

**UNIVERSITY OF OSLO**  
**Department of Informatics**

**Transport Layer  
challenges in  
hybrid military  
satellite networks**

Andreas Ramstad  
Urke

**May 3, 2011**





# Summary

On 26th of March 2010, the Norwegian Parliament approved the Norwegian Armed Forces project of acquiring a communications satellite. Combined with the increasing use of civilian technology like the Transmission Control Protocol (TCP) in military networks, research on TCP in military and satellite environments are required.

This thesis provides a thorough description of TCP, satellite environments, and related challenges in military networks containing both satellite and radio links. Geostationary satellites introduce an added delay of around 550 ms, while radio links are prone to high bit error rates. In such environments, TCP performance may suffer severely - the penalty depends on several factors, including the TCP flavor utilized at the sender. For Windows 7, the default TCP flavor is TCP NewReno, while Compound TCP is available. A wide variety of flavors are available for Linux, including the satellite-tailored variant TCP Hybla - the default flavor is CUBIC.

Evaluation and analysis through emulation has shown the mentioned TCP flavors to exhibit significant different performances at the sender side. The flavors available to Windows 7 perform poorly in lossy networks when compared to CUBIC and especially Hybla which outperforms the other flavors in such environments. CUBIC and Hybla are also seen to be more aggressive than Windows 7 flavors. If competing on the same bottleneck, this results in an unfair division of bandwidth - at the cost of Windows 7 flavors. Possible solutions include tuning of the TCP flavors and avoiding mixed-OS/TCP environments.

Another approach is the use of a Performance Enhancing Proxy at the sender side. The proxy evaluated, PEPsal, will intercept all TCP flows and forward traffic along a new TCP flow towards the intended receiver. This new TCP flow utilizes a tailored TCP flavor (e.g. Hybla) in order to increase performance regardless of the original TCP flavor at the sender. Analysis shows significant improved performance of Windows 7 flows in lossy networks. The unfair division of bandwidth is also decreased, but performance issues were observed. In addition, PEPsal break the end-to-end principle of TCP, and depends on a plaintext TCP header - causing challenges in encrypted military networks.

---

---

## Preface

This thesis concludes my Masters degree at the Department of Informatics at University of Oslo. The work has been carried out during the fall and spring of 2010 and 2011.

I wish to thank my supervisors for excellent counseling: Professor Knut Øvsthus at Bergen University College, and Dr. Lars Erling Bråten at Norwegian Defense Research Establishment (FFI, Forsvarets ForskningsInstitutt). A special thanks to both for providing a thesis within the topic of my wishes.

Thanks to FFI and their staff for equipment, counsel and hospitality during my lab work. And to Bergen University College for providing office space for me and my brilliant colleagues: S. K. Hammerseth, M. A. Lervåg, M. Ringkjøb, A. Skutle, A. Taranger and J.E. Vestbø. Especially, I would like to thank Jan Egil for being my 24-hour Linux support hotline.

Finally, a thanks to my parents, my brother and my wonderful wife, for their endless support throughout the entire Masters degree.

---

# Contents

Preface . . . . .	iii
<b>1 Introduction</b>	<b>1</b>
1.1 Background and motivation . . . . .	1
1.2 Scope . . . . .	2
1.3 Outline . . . . .	4
<b>2 Technical Background</b>	<b>5</b>
2.1 The OSI model . . . . .	5
2.2 The Transmission Control Protocol, TCP . . . . .	6
2.2.1 History . . . . .	7
2.2.2 Header . . . . .	8
2.2.3 Connection Establishment and Termination . . . . .	9
2.2.4 Data flow and flow control . . . . .	11
2.2.4.1 Retransmission and retransmission timeout . . . . .	13
2.2.5 Congestion Avoidance and Control . . . . .	15
2.3 Satellite Characteristics . . . . .	17
2.3.1 Delay . . . . .	17
2.3.1.1 Propagation . . . . .	17
2.3.1.2 Buffer . . . . .	18
2.3.2 Packet-loss and bit error . . . . .	19

2.3.3	Bandwidth Asymmetry . . . . .	19
<b>3</b>	<b>Challenges</b>	<b>21</b>
3.1	Bandwidth-delay product - Maximum Window Size . . . . .	21
3.2	Bandwidth-delay product - Slow Start duration . . . . .	22
3.3	Segment loss . . . . .	24
3.4	RTT Unfairness . . . . .	26
<b>4</b>	<b>Proposed Solutions</b>	<b>27</b>
4.1	TCP enhancements . . . . .	28
4.1.1	Enhancing mechanisms . . . . .	28
4.1.1.1	TCP Window Scale Option . . . . .	28
4.1.1.2	TCP Timestamps Option . . . . .	30
4.1.1.3	Increasing Initial Window (IW) . . . . .	32
4.1.1.4	Selective Acknowledgment . . . . .	33
4.2	TCP Flavors . . . . .	34
4.2.1	TCP NewReno . . . . .	35
4.2.2	CUBIC . . . . .	37
4.2.3	Compound TCP . . . . .	39
4.2.4	TCP Hybla . . . . .	41
4.2.5	Space Communications Protocol Specifications - Transport Protocol . . . . .	42
4.2.5.1	TCP Vegas . . . . .	44
4.3	Performance Enhancing Proxies (PEP) . . . . .	44
4.3.1	TCP Splitting . . . . .	46
4.3.1.1	Transparency and End-to-end argument . . . . .	47
4.3.1.2	Security . . . . .	47
4.3.2	PEPsal . . . . .	48



<b>5</b>	<b>Emulation</b>	<b>49</b>
5.1	Design . . . . .	49
5.2	Vyatta routers . . . . .	51
5.2.1	Configuration - IPsec . . . . .	51
5.2.2	Configuration - Delay and Packet Loss . . . . .	53
5.2.3	Configuration - Bandwidth limitation . . . . .	54
5.2.4	Packet Error Rate and Bit Error Rate . . . . .	55
5.3	FreeBSD with Dummynet . . . . .	56
5.3.1	Configuration - Bandwidth limitation . . . . .	57
5.4	TCP peers . . . . .	58
5.5	PEPsal and MultiTCP . . . . .	59
5.5.1	Configuration - PEPsal . . . . .	60
5.6	Tools . . . . .	60
5.6.1	Wireshark . . . . .	60
5.6.2	tcpdump . . . . .	61
5.6.3	TCP Probe . . . . .	61
5.6.4	Gnuplot . . . . .	62
5.6.5	tcptrace . . . . .	62
5.6.6	Iperf and Jperf . . . . .	63
5.7	Internet Protocol Security (IPsec) . . . . .	64
<b>6</b>	<b>Results and Analysis</b>	<b>69</b>
6.1	Emulation characteristics and confirmation . . . . .	70
6.1.1	On the performance of CTCP . . . . .	71
6.2	Results and analysis . . . . .	73
6.2.1	Impact of UDP . . . . .	73
6.2.2	Friendliness . . . . .	74
6.2.2.1	Implications . . . . .	77

6.2.3	Bit Error Rate (BER) . . . . .	79
6.2.3.1	Implications . . . . .	80
6.2.4	RTT and buffer . . . . .	81
6.2.4.1	Windows 7 Advertised Window problem . . . . .	81
6.2.4.2	Implications . . . . .	88
6.2.5	Webpages, HTTP, short transfers . . . . .	91
6.2.5.1	Implications . . . . .	93
<b>7</b>	<b>Conclusion</b>	<b>95</b>
7.1	Future work . . . . .	96
	<b>Appendices</b>	<b>103</b>
<b>A</b>	<b>Configuration of Vyatta routers</b>	<b>105</b>
A.1	Router D102 . . . . .	105
A.2	Router D103 . . . . .	108
<b>B</b>	<b>Gnuplot scripts</b>	<b>113</b>
B.1	tcpprint . . . . .	113
B.2	Selected script . . . . .	114
<b>C</b>	<b>PEPsal (iptables) script</b>	<b>115</b>
<b>D</b>	<b>Linux Traffic Control</b>	<b>117</b>
<b>E</b>	<b>Additional Results</b>	<b>119</b>

# List of Figures

2.1	Overview of OSI layer data structures and terminology [12]	7
2.2	Connection establishment/Three-way handshake	11
2.3	Connection termination/Four-way handshake	12
2.4	TCP flow with sliding window	13
2.5	Performance of Van Jacobson and Karels RTO algorithm, adopted from [22]	14
2.6	The slow start algorithm	16
2.7	The congestion avoidance algorithm	17
3.1	Duration, and data transfered, during slow start	23
3.2	Wireless link errors impact: goodput after 180 s of a GEO satellite connection (RTT = 600 ms) at different PER values (0 %, 0.1 % and 1 %). Adopted from [30].	25
3.3	RTT unfairness problem: goodput after 180 s of a satellite connection (variable RTT, PER = 0 %) in presence of 5 short RTT terrestrial background connections (Reno, RTT = 25 ms, PER = 0 %). Adopted from [30].	26
4.1	FTP transfer over high BDP link with TCP Window Scale Option disabled	30
4.2	FTP transfer over high BDP link with TCP Window Scale Option enabled	31
4.3	Improvement of throughput, compared to $IW = 1 \times MSS$ , for different values of IW. Adopted from [34]	32
4.4	Wireshark capture showing default Initial Window in Windows 7 (TCP NewReno)	33
4.5	Wireshark capture showing Initial Window in Ubuntu 10.10	33

4.6	Wireshark capture shows Selective Acknowledgment enabled by default in Windows 7. . . . .	35
4.7	The Window Growth Function of CUBIC [16] . . . . .	37
4.8	<i>cwnd</i> of CUBIC during FTP transfer over 500 kbps link with 528 ms RTT . . . .	38
4.9	Set of equations from Hybla presentation. [17] . . . . .	41
4.10	Integrated PEP scheme. . . . .	46
4.11	Distributed PEP scheme. . . . .	46
5.1	Logical design of emulation . . . . .	49
5.2	Physical design of emulation . . . . .	50
5.3	Physical design of emulation, with PEP . . . . .	50
5.4	Layer 3 design of emulation . . . . .	51
5.5	Protocol Operation for ESP [60] . . . . .	65
5.6	Wireshark capture of TCP/FTP traffic . . . . .	66
6.1	Wireshark capture showing ICMP answer from router to TCP peers Path MTU Discovery algorithm. . . . .	70
6.2	RTT of CTCP flow with 100 % buffer. . . . .	72
6.3	Goodput of single TCP flow with and without competing UDP flow. . . . .	73
6.4	Link utilization of single TCP flows (marked "1"), and friendliness between two competing (marked "2") TCP flows of different flavor. . . . .	74
6.5	Impact of competing CUBIC flow (introduced at approx. 70sec.) on Windows 7 NewReno flow. . . . .	75
6.6	Multiple Windows 7 Reno flows vs. one Linux CUBIC flow. All in steady state. .	76
6.7	Friendliness between CUBIC and Reno for different buffer sizes. . . . .	77
6.8	Friendliness between different flavors of TCP, all going through PEPsal utilizing Hybla. . . . .	78
6.9	Impact of Bit Errors on single TCP flows. . . . .	79
6.10	Average RTT for different flavors and buffer sizes. . . . .	81
6.11	RTT of NewReno flow between two Windows 7 hosts. . . . .	82

6.12	RTT of CTCP flow between two Windows 7 hosts. . . . .	82
6.13	RTT of CUBIC and Hybla flows with 1000% BDP buffer. . . . .	83
6.14	RTT and Receiver Window Size for NewReno flow between two Windows 7 hosts.	83
6.15	RTT and Receiver Window Size for CUBIC flow between two Linux hosts. . . . .	84
6.16	Maximum Advertised Receiver Window for different OS and buffer sizes. . . . .	85
6.17	Friendliness between competing flows, with 1000% of BDP buffer. Flow with Windows 7 receiver in steady-state, except *. . . . .	86
6.18	RTT of NewReno and CTCP flow from Windows 7 to Linux with 1000% BDP buffer. . . . .	87
6.19	RTT and goodput of NewReno flow between two Windows 7 hosts with 100% BDP buffer. . . . .	88
6.20	RTT and goodput of CUBIC flow between two Linux hosts with 1000% BDP buffer.	89
6.21	RTT and goodput of NewReno flow between two Win 7 hosts, 1000% BDP buffer.	90
6.22	Time-sequence graph showing first seconds of CUBIC and NewReno transfers. . .	92
E.1	RTT and goodput of NewReno flow between two Windows 7 hosts with 200% BDP Buffer. . . . .	119
E.2	RTT of Hybla flow between two Linux hosts. . . . .	120
E.3	RTT and goodput of Hybla flow between two Linux hosts with 100% BDP Buffer.	120
E.4	RTT and goodput of Hybla flow between two Linux hosts with 200% BDP Buffer.	121
E.5	RTT and goodput of Hybla flow between two Linux hosts with 1000% BDP Buffer.	121
E.6	RTT and goodput of CTCP flow between two Windows 7 hosts with 100% BDP Buffer. . . . .	122
E.7	RTT and goodput of CTCP flow between two Windows 7 hosts with 200% BDP Buffer. . . . .	122
E.8	RTT and goodput of CTCP flow between two Windows 7 hosts with 1000% BDP Buffer. . . . .	123
E.9	RTT of CUBIC flow between two Linux hosts. . . . .	123
E.10	RTT and goodput of CUBIC flow between two Linux hosts with 100% BDP Buffer.	124
E.11	RTT and goodput of CUBIC flow between two Linux hosts with 200% BDP Buffer.	124



# List of Tables

2.1	Overview and comparison of OSI and TCP/IP model . . . . .	6
2.2	TCP header . . . . .	8
4.1	Overview of TCP NewReno behavior . . . . .	36
5.1	Corresponding BER to configured PER in emulation. . . . .	56
5.2	Overview of security mechanisms in emulation . . . . .	64
5.3	Length of mandatory headers and trailers in ESP packet, tunnel mode . . . . .	65
5.4	Maximum goodput of TCP over different IPsec links. . . . .	67
6.1	Default parameters of emulation. . . . .	69
6.2	Penalty and properties of web page downloading for high RTT. . . . .	92
6.3	Average time to download various web pages for different RTT. . . . .	93





# Chapter 1

## Introduction

### 1.1 Background and motivation

On 4th of December 2009, the Norwegian Ministry of Defense, issued a proposition [1] to the Norwegian Parliament recommending an acquisition of a communications satellite for the Norwegian Armed Forces. The proposal evaluated several options, but concluded and suggested a joint acquisition in collaboration with Spain and the Spanish operator Hisdesat. On 26th of March 2010, the National Parliament adopted the proposal, and allowed the project to begin immediately within a 982 mill. NOK budget frame. The initiator of this thesis is the Norwegian Defense Research Establishment (FFI, Forsvarets ForskningsInstitutt).

There is a ongoing shift in military communication environments, from developing own proprietary equipment and protocols, to an extended use and modification of open civilian technologies. An example is the United States Department of Defense (DoD) which in 2007 was in the process of implementing a global Internet-like network for "warfighters", called the Global Information Grid [2]. This network will utilize Internet Protocol (IP) and Transmission Control Protocol (TCP), and be encrypted using a variant of IPsec. A similar approach is seen in the Norwegian Armed Forces. In an article in the FFI journal FFI-FOKUS [3], the Network based Defense (NbF) is described. The network will create a common, converged, Internet/IP-based platform in which all services may run. Emphasis is put on the use of civilian solutions, and modifications of these to meet military requirements. As a part of the NbF, the future Norwegian Modular Network soldier will communicate using IP-enabled handheld PDAs [4].

Another recent example is the April 18th 2011 US Army decision to use the Android operating system on their Joint Battle Command-Platform, or JBC-P Handheld [5]. The handheld devices (or smartphones) are used at the front lines identifying enemies, providing navigation

and communication etc. Third-party developers can offer applications and thereby expand the capabilities of the smartphones rapidly, not unlike the civilian/commercial realm. Illustrating this new paradigm: "All of the research dollars are out there in the commercial market. All of the best minds are at work in these companies to produce these smartphones and this software. We don't want to rehash that, we want to leverage it. We want to take advantage of it (...)" (Lt. Col. Mark Daniels, product manager for JBC-P, in [5]). As for the hardware, both commercial off-the-shelf and military smartphones are being evaluated.

It is evident that the usage of civilian commercial solutions are getting more common in military networks. At the transport layer, the dominant protocol in the Internet is the TCP. Consequently, any challenges or caveats of TCP met in the civilian realm may have to be addressed when employed in a military network. This renews and facilitates the need for research on the topics from a military perspective, and results of such efforts can be seen in numerous papers, e.g. [2], [6], [7] etc.

There is a widespread consensus that the TCP suffers when delay and bandwidth across the network increase. In addition to reduced throughput, it is difficult to achieve fair division of the bandwidth between flows with different properties. Combining long delay and high bandwidth with increased packet losses result in even further degradation. A network with a satellite and radio link would fit this description. In these networks, the widely deployed and utilized TCP is heavily penalized, and an alternative is needed. There exist already a wide variety of solutions ranging from TCP enhancements, new TCP flavors, entirely new protocols, to proxies. There is still not an universally accepted and standardized solution, and most likely there will never be. This thesis will explore this topic inside the scope given below.

## 1.2 Scope

The thesis aims to evaluate several of the solutions, but with the enormous amount of proposals, it is infeasible to evaluate all transport layer aspects. This section will define the boundaries of the thesis and how it separates itself from earlier research. The scenario is a network with a satellite leg introducing long delay (around 500 ms) and loss, but with only a moderate bandwidth (around 0.5 Mbps) available. In addition, a radio link is added to accommodate a last-hop radio distribution leg. Such hybrid networks are common in a military scenario. Given the mentioned background, all tests and evaluations will be conducted in a military-like network with an encrypted and plaintext portion. Also, a realistic [8] buffer scheme is utilized in the testbed - a parameter often omitted in modern TCP literature. The solutions should be currently available and easily or widely employed. With many users, and since it is infeasible to control all of them, it is especially interesting to evaluate the default schemes shipped with Linux and Windows 7

(expected to become widely deployed). Not just the performance of the systems themselves, but also how they interact with the proposed solutions, e.g. how will a Linux user affect a Windows user, and vice versa.

The performance of these protocols and proxies will be tested using an emulator. An elaborate discussion and description on different methods of evaluating TCP can be found in [9]. The authors presents the alternatives: Simulation, Emulation, and Live Internet Tests. Live Internet Test were quickly discarded since this requires an actual satellite link, which is both costly and limits the control of delay, buffers, and bandwidth. Emulations aim to model a selected piece of the network path between two hosts - in our scenario: A satellite link. The hosts are real computers with actual TCP implementations (as oppose to simulators) giving the benefit of (possibly) more accurate results. Emulations also offer control of buffers, queues etc. in intermediary network equipment like routers and switches. As with hosts, these are actual equipment and not an abstract implementation. A caveat of emulation is the emulator itself which may not be able to behave as the piece of the network it is trying to emulate. I.e. an actual satellite may produce more realistic results than the emulator. FFI utilize tactical routers in their networks, and it was aimed to utilize these in an evaluation of the TCP flavors. The difficulty of simulating these routers, and the aforementioned reasons, led to usage of an emulator in this thesis.

A testbed has been set up to emulate the described scenario and test properties of the solutions as the scenario, traffic and other parameters are adjusted. The thesis aims to identify any improvement, threat or solution which can affect the performance of the network.

Mainly, the assessed schemes are different TCP flavors. Each TCP flavor has different mechanisms designed to improve its performance. These are described in Section 4.2. The TCP flavors evaluated are:

- **TCP NewReno:** This is the default TCP implementation for Windows 7 and Windows Vista. NewReno is a slightly modified version of TCP Reno which earlier enjoyed widespread deployment, and is utilized in Windows XP. In most scenarios, their performance will be identical.
- **CUBIC:** Since kernel version 2.6.19, CUBIC has been the default TCP flavor for Linux. A similar variant, BIC, was used for kernel 2.6.18.
- **Compound TCP:** Compound TCP (CTCP) is a Microsoft patented TCP version, which is default in Windows Server 2008. It is also available in Windows Vista and Windows 7 (disabled by default). And via a downloadable patch for Server 2003 and 64-bit XP.
- **TCP Hybla:** Hybla is especially tailored for satellite environments with high packet loss and long delays. It is available as a module in the Linux kernel since version 2.6.13, making

it easily deployable.

In addition, a Performance Enhancing Proxy (PEP) has been evaluated. PEPs are placed in the network and intercept TCP flows. Dependent on the type of PEP, it perform different operations on the TCP flow with the aim to improve its performance. A thorough description of PEPs can be found in Section 4.3. The PEP evaluated in this thesis is:

- **PEPsal:** PEPsal is an open source, freely available PEP, compliant with existing standards. It was developed at the University of Bologna, Italy, in 2005, and to the authors knowledge is the only open source TCP PEP available. A description can be found in Section 4.3.2.

The thesis also presents several TCP mechanisms which are not necessary tied to a specific TCP flavor. These mechanisms are usual optional and used independent of the TCP flavor utilized, but all aim to improve TCP performance. In addition, a relevant transport layer protocol called Space Communications Protocol Specifications - Transport Protocol, is described and evaluated. It is mainly a modified version of TCP, but with some important and interesting differences.

## 1.3 Outline

The rest of this thesis is organized as follows: Chapter 2 presents a technical and theoretical background on TCP and satellites treating the relevant topics giving the unfamiliar reader a foundation on which to read the rest of the thesis. The following chapter identifies the problems - i.e. the challenges which arise and why they occur. Chapter 4 presents related research and proposed solutions to the challenges described in the previous chapter. Following is a presentation of the emulation testbed with detailed descriptions to allow reproduction. Chapter 6 presents and analyzes results from the mentioned emulation. Implications and solutions are also discussed. Chapter 7 concludes the thesis and presents future work.

## Chapter 2

# Technical Background

This section will explain relevant aspects of the Transmission Control Protocol (TCP) and the satellite characteristics which cause transport layer challenges over satellite links.

### 2.1 The OSI model

In 1977 the International Organization for Standardization (ISO) created a new subcommittee called SC16. They were tasked with creating standards leading up to Open Systems Interconnection (OSI). Up to this date each computer manufacturer developed their own "protocols" for interconnection between their own equipment - clearly showing a need for joint and open standards allowing interconnection. In 1978 the SC16 had its first meeting, and at the end of 1979 the Reference Model of Open Systems Interconnection, popularly called "OSI model", was adopted. After further revision the seven-layered OSI model (Table 2.1 ) was published in ISO 7498. For more about the history behind the OSI model, see [10].

The OSI model is an abstract reference model dividing the communication system into layers. Each layer has its own tasks and functions, and provides services to the layer above, and utilizes the services offered from the layer below. The OSI model is a reference model, meaning that actual communications systems do not strictly follow its structure. A second model was created using experiences from the ARPANET and culminated in the TCP/IP model (Table 2.1 ) described in IETF RFC-1122 and RFC-1123. One might argue that the TCP/IP model more accurately represents the real world since TCP in fact handles sessions.

The top three OSI model layers are loosely defined in the ISO 7498 standard [11] and will not be given much attention in this paper, with the exception of the session layer which TCP partly

Layer	OSI	TCP/IP
7	Application	Application
6	Presentation	
5	Session	
4	Transport	Transport
3	Network	Internet
2	Link	Link
1	Physical	Physical

Table 2.1: Overview and comparison of OSI and TCP/IP model

resides in. Also residing on the transport layer is the User Datagram Protocol (UDP), which is a connection-less (session-less) transport protocol. Together, TCP and UDP is the transport protocols on the Internet. Examples of protocols residing at layers above are can be identified in common services like World Wide Web (HTTP, HTTPS), E-Mail (SMTP,POP,IMAP), Remote Login (SSH,TELNET) etc. All of these protocols utilize services provided by TCP or UDP.

In the layer below the transport layer we find the network layer where Internet Protocol (IP) is the dominant protocol and provides routing and relaying of packets/datagrams (with TCP or UDP segments inside) as a service to the transport layer above. It is worth noting that the IP protocol is a best-effort protocol and does not guarantee delivery, nor that the order of segments received is consistent with the order in which they were sent. These responsibilities are left for the uppers layer. Layer 2 is the link layer, which handles transport of frames (with IP packets inside) on the link between network-entities. The dominant protocol is the Ethernet protocol, standardized as IEEE 802.3. The physical layer provides the actual carrying of frames from a link-entity to another, e.g. fiber, radio waves, copper etc. Figure 2.1 shows an overview of the terminology and how each layer manipulate the information received from the layer above before sending it to the layer below. For more detailed information on the different layers of the OSI model, see [11]

## 2.2 The Transmission Control Protocol, TCP

TCP is an enormous topic and this paper will only cover selected topics and aspects. The purpose of this introduction is to give the reader a basic understanding of the key components and workings of the protocol - with focus on elements which is relevant for the performance of TCP over networks with long delay and high packet loss and errors.

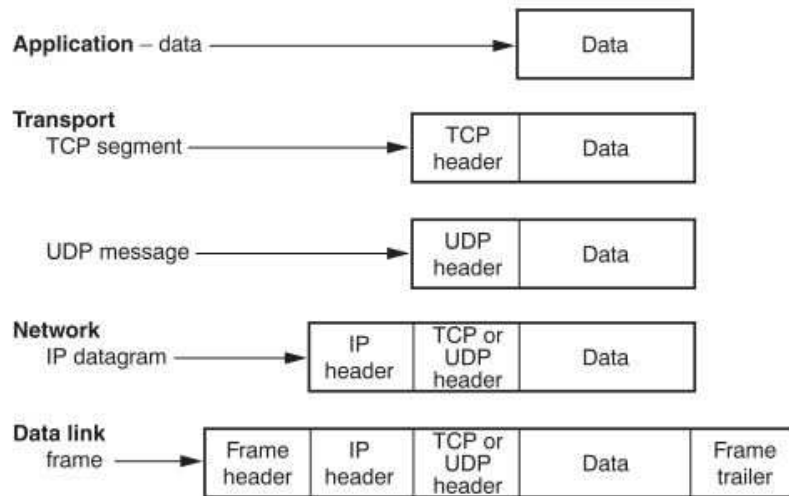


Figure 2.1: Overview of OSI layer data structures and terminology [12]

### 2.2.1 History

The concepts of TCP were first described by Vinton G. Cerf and Bob Kahn in their paper "A Protocol for Packet Network Intercommunication" which was published by IEEE in May 1974. Later in 1974, Vinton Cerf, Yogen Dalal and Carl Sunshine writes RFC-675, and it described an extensive protocol named Transmission Control Program with functions ranging over both layer 3 and 4. Through several revisions the protocol was split (IP becoming the latter part), changed name and functionality, and version 4 of the Transmission Control Protocol was adopted by U.S. Department of Defense (DoD) in February 1980 [13]. The protocol was then described in RFC-793 [14] in September 1981, and the introduction states:

"TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications."

Several extensions and modifications have been proposed since then. Some of them have become mandatory in TCP implementations, e.g. the Congestion Avoidance and Slow Start algorithm. Some extensions are optional, such as RFC-2018 "TCP Selective Acknowledgment Options". There also exist a wide assortment of TCP variants, e.g. TCP Vegas [15], TCP CUBIC [16], TCP Hybla [17] etc. Some of these variants and extensions are designed to increase performance in satellite networks and will be discussed thoroughly later in this paper.

### 2.2.2 Header

For future reference, the TCP header will be presented along with a short description of each field.

Bit:	0	4	8	16	31
Source Port			Destination Port		
Sequence Number					
Acknowledgment Number					
Head. Size	Reserved	Flags		Window	
Checksum			Urgent Pointer		
Options + Padding					

Table 2.2: TCP header

Each "line" in Table 2.2 is 32 bits, making the entire header at least 20 bytes long. Trailing in the header comes an optional variable length Options field, at last followed by the Application data - combining all these pieces results in a complete TCP segment. Presented below are an explanation of each field as they are standardized in RFC-793 [14], including proposed changes in RFC-3168 [18].

- **Source and destination port (16 bits):** These 16 bits identify a specific TCP-flow, and is coupled with the application utilizing it. Thus TCP can offer multiple simultaneous TCP-flows to the layers above, giving the user the possibility to run multiple applications utilizing TCP at the same time.
- **Sequence Number (32 bits):** 32-bits identifies the sequence number of the first byte of the application data in the TCP segment. If the SYN flag is set, this field identifies the initial sequence number (ISN), and the first byte of data would have sequence number ISN + 1.
- **Acknowledgment number (32 bits):** If the ACK flag is set, this number identifies the sequence number of first data byte in the next segment the sender is expecting to receive.
- **Offset (4 bits):** Since the TCP header has variable length(due to the optional variable length Options field), the offset field identifies number of 32-bit words in the header. In other words, this field identify where the application data begins.
- **Reserved (4 bits):** Reserved for future use. Must be set to zero.
- **Flags/Control Bits (8 bits):** RFC-793 originally specified 6 flags using 6 bits - however, RFC-3168 "The Addition of Explicit Congestion Notification (ECN) to IP" [18] specifies



the use of 2 new flags, using 2 bits from the Reserved field. These extensions are widely supported[19] and will thus be included here. From left to right:

- CWR: The data sender informs the data receiver that the congestion window has been reduced.
  - ECE: ECN-Echo. Used by the data receiver to inform the sender of a received Congestion Experienced (CE) packet.
  - URG: Urgent field is significant. Tells the receiver not to ignore the Urgent field.
  - ACK: Acknowledgment field is significant. Tells the receiver not to ignore the Acknowledgment field.
  - PSH: Informs the receiver that the Push operation (causes TCP to immediately forward all data received by the user) has been invoked.
  - RST: Reset the TCP connection
  - SYN: Synchronize sequence numbers; start of a new TCP connection.
  - FIN: No more data from sender.
- **Window (16 bits):** The number of data bytes, beginning with the one indicated in the acknowledgment field, which the sender of this segment is willing to accept (available buffer space). This field is very important for long-delay TCP transmissions and will be covered extensively throughout the paper.
  - **Checksum (16 bits):** Computed over the entire TCP segment, plus a pseudo header(outside the scope of this paper).
  - **Urgent Pointer (16 bits):** This field is read only when the URG-flag is set. It indicates that this segment is carrying data that needs urgent priority and the segment should be processed before any other packet in the buffer. The Urgent Pointer points to the sequence number of the last byte of the urgent data.
  - **Options (variable length):** An example of option is the 16-bit Maximum Segment Size (MSS) setting the MSS for the transmission at initiation.

### 2.2.3 Connection Establishment and Termination

The TCP protocol is a connection-oriented protocol, providing reliable and in-order delivery end-to-end from source to destination. These services are made possible by the use of sequence numbers and acknowledgments which implies that each peer has synchronized sequence numbers. Before any data is transferred, two TCP-peers establish a connection where they synchronize

values for sequence number, and other options like Maximum Segment Size (MSS). The process of setting up a connection is often called three-way handshaking, and if successful, the data transfers may commence. When both peers are finished sending data, the connection is terminated by a process called four-way handshaking. The following sections will explain the three-way and four-way handshake as described in [13].

**Three-way handshake** As illustrated in Figure 2.2, the connection establishment procedure consists of three steps:

1. The initiator or client starts the process by sending a TCP segment to a specified port at the receiver. E.g. a request to a web-server, resulting in port 80 as destination port and an arbitrary, random port as source.

The segment will have the SYN-flag set; indicating that this is a new TCP connection and the Sequence Number field contains the client's Initial Sequence Number (ISN).

2. The receiver or server responds with a TCP segment with e.g. source port 80, and destination port set to the random source port from the first segment received. Thus, the two peers have sent segments which can uniquely be separated from other TCP flows.

The segment will have both the SYN and ACK-flag set. The Sequence number field will contain the server's ISN. The Acknowledgment number field will contain the client's ISN+1, acknowledging the received data and indicating that the next byte of data the server expects to receive has sequence number ISN+1.

3. Finally, the client will send a TCP segment in return with the ACK-flag set and the Acknowledgment number field set to the server's ISN+1. This concludes the handshake - the connection is established and data may be exchanged between the peers.

**Four-way handshake** Illustrated in Figure 2.3, the TCP connection termination procedure consist of four steps:

1. The peer, e.g. client, who are finished sending data and wish to terminate the TCP connection sends a segment with the FIN-flag set, indicating it wants to terminate the connection.
2. The recipient, e.g. server, confirms the termination with a segment with the ACK-flag set. The traffic in the direction from client to server is now stopped. Note, the other direction is still open and data may still be transferred.
3. The server then sends (when it is ready) a segment with the FIN-flag set.

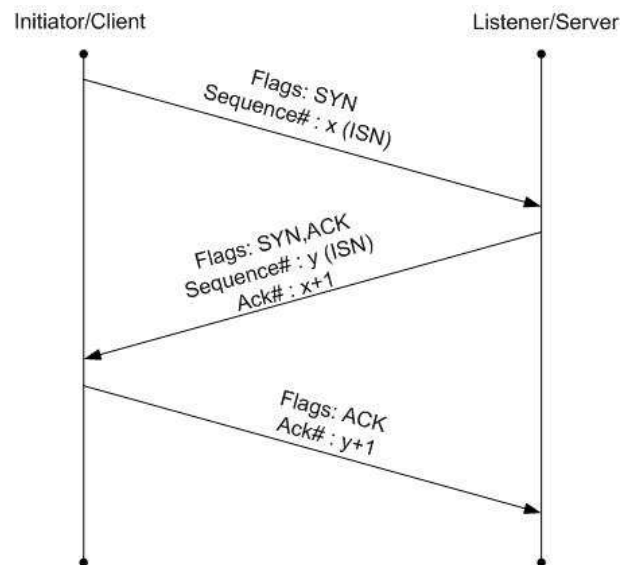


Figure 2.2: Connection establishment/Three-way handshake

4. The client responds with a segment with the ACK-flag set. This closes the communication in the server-to-client direction, thus closing the connection completely.

A TCP connection is uniquely identified by the port number, making each peer capable of separating this flow of data from other TCP connections. Applications may request and utilize one or several TCP connections simultaneously, e.g. a browser downloading a web page is likely to use multiple TCP connections. A TCP connection is also full duplex, allowing both peers to send and receive data. Note that the client/server concept may be somewhat confusing in a full-duplex context - the term TCP peer is often utilized.

### 2.2.4 Data flow and flow control

The following two sections will focus on the mechanism which is in play during a TCP data transfer, i.e. the connection has already been established. These mechanisms are described in RFC-793 [14], RFC-2488 [20], RFC-5681 [21] and "Congestion Avoidance and Control" by Van Jacobson and M.J. Karels [22].

As mentioned earlier, TCP puts data (received from the upper layer) into segments, giving each byte a sequence number and putting the sequence number of the first byte into the header. However, it is the receiver who governs how much data it is willing to receive and thus how much the sender can transmit. The receiver sets these limits in the Window field and is given as the number of bytes the receiver is willing to accept. During a transfer the receiver may adjust the

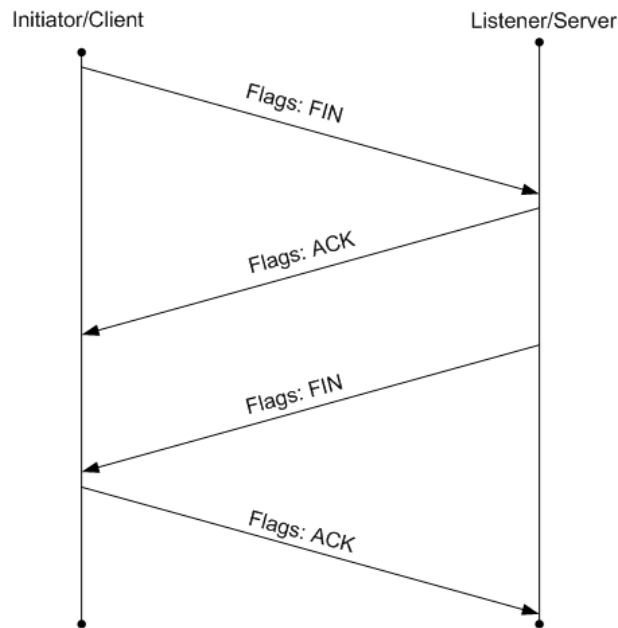


Figure 2.3: Connection termination/Four-way handshake

window size by setting a new value in the Window field. The assumption is that the window size reflects the available buffer size at the receiver. If the sender ignores the window, i.e. transmits above the limits imposed by the window, the overflow data should be discarded.

The size of each segment is governed by the amount of data to be transferred, up to the limits imposed by the Maximum Segment Size (MSS). The MSS can be set in multiple ways: By the Path MTU (Maximum Transmission Unit) Discovery algorithm described in RFC-1191. By the receiver via the MSS option in the TCP-header. Or it could be defaulted to 536 bytes as specified in RFC-1122.

For future reference we will use some of the definitions found in RFC-5681: The last Advertised Window by the receiver is called Receiver Window (*rwnd*). Flight Size is the amount of data that has been sent but has not been acknowledged.

The window scheme used in TCP is called sliding window. This implies that the window "slides" forward consecutively to the next data which is ready to be transmitted as the sender receive acknowledgments, see Figure 2.4. The *rwnd* is set to 1500, implying that the maximum flight size is 1500 bytes. After sending three segments, each with 500 bytes, the window is "full", forcing the sender to stop transmission while waiting for an acknowledgment. When a segment arrives with  $Ack\# = 2000$ , the receiver has acknowledged the reception of all 1000 bytes from  $Seq\# 1000$  to 1999 and is expecting the next byte to have sequence number 2000. This means that the flight size has decreased by 1000 bytes, the window slides forward, and opens up for 1000 more

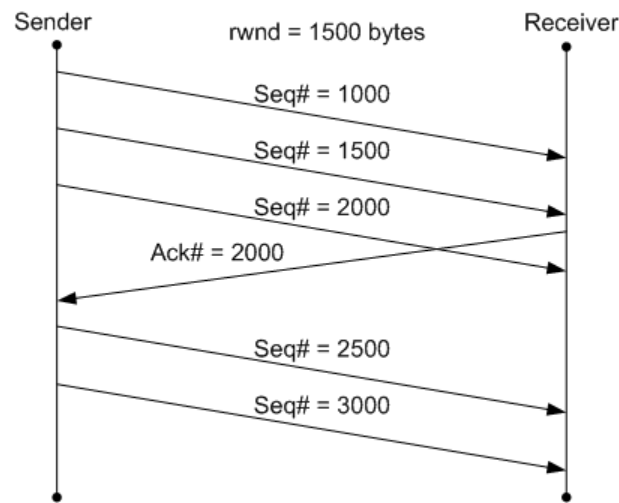


Figure 2.4: TCP flow with sliding window

bytes to be transferred.

Note however that there exist algorithms which causes the receiver not to reply with an ACK<sup>1</sup> immediately. RFC-1122 describes "Delayed ACK", which *recommends* that a receiver wait (maximum 500ms) before sending an ACK. It aims to reduce ACK traffic, since the receiver could potentially acknowledge several segments using one ACK-segment.

#### 2.2.4.1 Retransmission and retransmission timeout

If an acknowledgment has not been received for a given sequence number/segment within a retransmission timeout (RTO), a timeout will occur, and the segment will be re-transmitted. Setting a correct RTO is a very important and difficult task. It should not be too short, as this would cause the sender to prematurely timeout and retransmit a correctly received segment. Yet it should not be too long as this would decrease throughput since the sender will have to wait for the RTO to expire before retransmitting. The difficulty of this task is caused by the varying Round-Trip Time (RTT), which tells how long a segment needs to travel across the network, and back. In other words, how long time a segment with data needs to reach the receiver, and how long it takes for the ACK to return (not accounting for other delays like processing time). If RTT was a constant value, one could roughly set  $RTO = RTT$ . However, this is not the case, and there are several proposals on how to measure RTT and how to set the RTO based on these measurements. These methods will be discussed later, as they are an important mechanism in several of the TCP flavors targeted for enhancing TCP over satellite. RFC-793 only *suggest* one

<sup>1</sup>A segment with the ACK-flag set is usually referred to as an ACK or an ACK segment.

method, but this was later improved by Van Jacobson and Karels [22]. This method is used in one of the earlier, more popular TCP flavors: TCP Reno. A presentation of the entire algorithm is outside the scope of this paper, but Figure 2.5 shows the measured performance. The dotted line represents actual segments and their RTT. The solid line represents the RTO as set by the algorithm. As discussed earlier, we can see that the RTO is "always" greater than RTT, but not by too much.

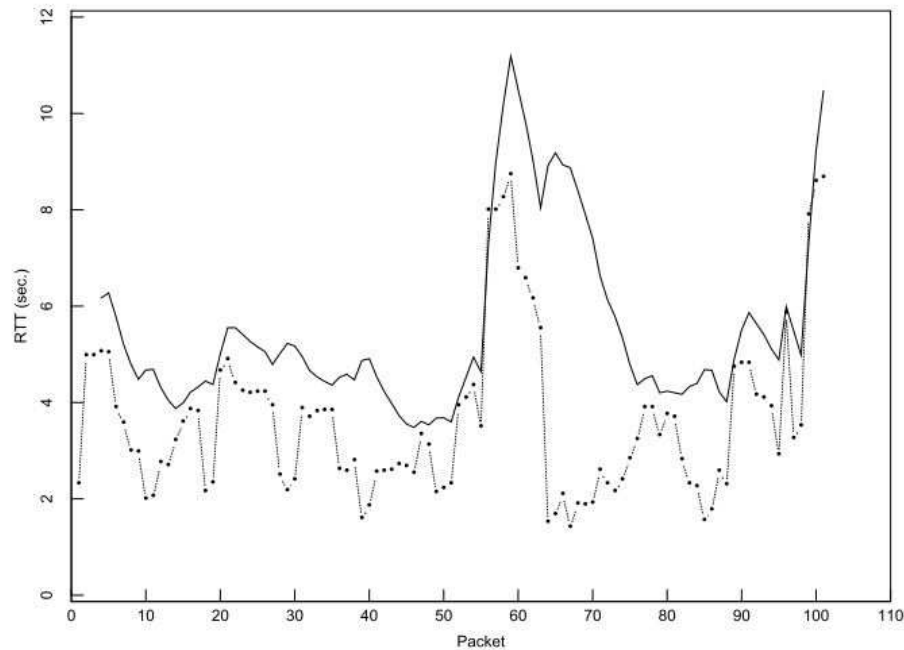


Figure 2.5: Performance of Van Jacobson and Karels RTO algorithm, adopted from [22]

For future reference, it should be noted that there is another way to reach timeout. If a segment is lost in transit but the preceding segments arrive, the TCP receiver will buffer the incoming segments, but generate an ACK which is a duplicate of the previous ACK sent. RFC-793 states that this ACK should be discarded when arriving at the sender, but other flavors e.g. Reno and Tahoe uses these duplicates: Three duplicate ACKs indicate a lost packet.

When recovering from lost or disordered segment, the receiver will sort them according to sequence number before the data is delivered to the upper layers. The sorting and retransmission is the two mechanisms which enable TCP to deliver reliable and in-order transmission of data.

### 2.2.5 Congestion Avoidance and Control

The previous section described mechanism which makes sure a receiver is not swamped with incoming traffic beyond what its buffer can handle. However, the receivers adjustment of *rwnd* does not take congestion throughout the network into consideration. Without any mechanism which limits the TCP flows, they will easily overwhelm the network. As Van Jacobson and Karels states in the introduction to their paper "Congestion Avoidance and Control" [22], there were early realizations of this problem:

"In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP<sup>2</sup> hops) dropped from 32 kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad."

Their, and other investigations resulted in several new mechanisms. The *Slow Start* and *Congestion Avoidance* were made mandatory, while *Fast Recovery* and *Fast Retransmit* are recommended, as stated in RFC-1122. This section will only describe the two first. The Slow Start and Congestion Avoidance mechanisms added two new variables called congestion window (*cwnd*) and slow start threshold (*ssthresh*) to the TCP flows. Where it earlier only was the *rwnd* who governed the maximum allowed flight size, it is now set to the minimum of *cwnd* and *rwnd*. The *ssthresh* determines whether to use the Slow Start or Congestion Avoidance algorithm to govern the flow. RFC-1122 specifies the algorithms to be used as the ones described in "Congestion Avoidance and Control" from 1988. However, several other algorithms has been proposed, and the difference between flavors of TCP (Reno, NewReno, BIC, CUBIC, Hybla etc.) are often in their congestion avoidance and control algorithms. This section will describe the algorithms described in "Congestion Avoidance and Control" for reference (used in TCP Tahoe, and form the foundation for several other flavors), as other flavors and algorithms will be described later.

**Slow Start Threshold** The *ssthresh* governs which congestion control algorithm should be used. The slow start algorithm is used when  $cwnd < ssthresh$ , or until congestion is experienced, while congestion avoidance is used otherwise. *ssthresh* should initially be set arbitrarily high, e.g. to the size of the largest possible advertised window. This is to make sure that it is the network capacity which limits the increase of *cwnd*, and not the *ssthresh*. When experiencing congestion the *ssthresh* must be reduced to half the current *cwnd*.

---

<sup>2</sup>Interface Message Processor, an equivalent of today's routers. Connecting the different networks of the ARPANET together.

**Slow Start** The slow start algorithm is used in the beginning of a transmission or in the restart after loss. The purpose is to probe the "unknown" network to discover its capacity by slowly increasing the *cwnd*. It is important that this increase is not too slow or too fast, as this would make the probing ineffective. The size of the congestion window after the three-way handshake is called Initial Window (IW) and is set to one segment (proposed to be up to four segments in RFC-5681).

After the IW has been set, the *cwnd* is increased by 1 segment for each segment that has been acknowledged, as depicted in Figure 2.6. Thus, *cwnd* increases exponentially.

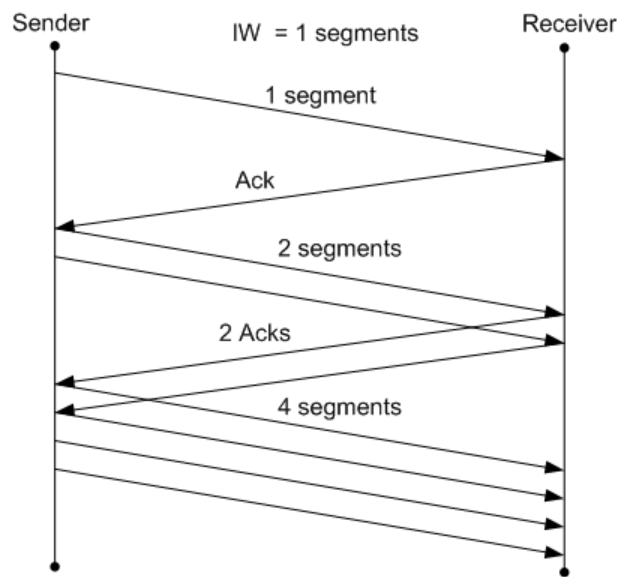


Figure 2.6: The slow start algorithm

**Congestion Avoidance** During congestion avoidance, the *cwnd* should be increased by  $1/cwnd$  (in segments) per received ack as seen in Figure 2.7. This results in an increase of 1 segment per RTT, and thus it increases linearly, until congestion is experienced. If a timeout (packet loss/error) occurs, the *ssthresh* is set to half the current *cwnd*, and the *cwnd* is set to 1 segment which forces slow start. The slow start will then exponentially increase the *cwnd* until the *ssthresh* is reached (which is set to half of the last window size where we "got in trouble") - after which the congestion avoidance algorithm is used.



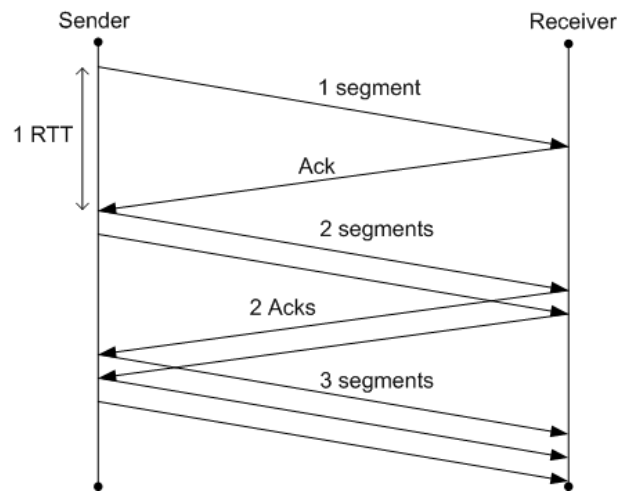


Figure 2.7: The congestion avoidance algorithm

## 2.3 Satellite Characteristics

This section briefly describes the characteristics of a satellite link which are relevant to TCP traffic and who cause some of the challenges which will be discussed later.

### 2.3.1 Delay

The RTT in a transfer consist mainly of two components: Propagation delay, and serialization delay. One may also experience processing delay at the end-hosts and intermediary network equipment. Serialization delay is the time it takes for a sender to realize the bits onto the medium, i.e. the packet size divided by the bandwidth. However, these delays are very small when compared to the propagation delay across satellite links. Another source of delay appears when traffic overflows the available bandwidth. The overflow traffic is buffered in front of the bottleneck while waiting to be transmitted. Depending on the size of the buffer, this will cause additional delay.

#### 2.3.1.1 Propagation

A satellite link is, because of the distance between the satellite and the earth, prone to propagation delay of various length depending on the altitude of the satellite orbit and the position of the Earth station. Communication satellite orbits are usually categorized into Low-Earth-Orbit (LEO), Medium-Earth-Orbit (MEO) and Geostationary orbit (GEO). In a GEO orbit the

satellite stays stationary relative to the Earth's surface, and these satellites are the focus in this paper.

GEO satellites also have the highest altitude of the three categories; 35768km.

The high altitude consequently forces a signal to propagate over long distances before reaching the satellite, and back to earth. This propagation delay can easily be calculated:

$$RTT = 4 \times \frac{d}{c}$$

where  $d$  is the distance from the earth station to the satellite and  $c$  is the speed of light. Following is a short example calculating the RTT between Ålesund, Norway (62°N, 6°E), and one of the several Norwegian THOR satellites in GEO orbit at 1°West:

$$d = \sqrt{R_e^2 + r^2 - 2R_e r \cos(\gamma)}$$

where  $R_e$  is the radius of the earth,  $r$  is the distance from the earth center to the satellite, and  $\cos(\gamma)$  is calculated as follows:

$$\cos(\gamma) = \cos(L_e) \times \cos(l_e - l_s)$$

where  $L_e$  is the latitude of the earth station,  $l_e$  and  $l_s$  are the longitude of the earth station and satellite, respectively. This yields:

$$\cos(\gamma) = \cos(62) \times \cos(6 - (-1)) = 0,47$$

$$d = \sqrt{6378^2 km + 42164^2 km - 2 \times 6378 km \times 42164 km \times 0,47} = 39603 km$$

$$RTT = 4 \times \frac{3960.3000 m}{3 \times 10^8 m/s} = 528 ms \tag{2.1}$$

This shows that the RTT between two peers in Ålesund, connected via a GEO satellite, should be at least 528ms. This does not take account any delay on board the satellite or delay induced by ground equipment like a satellite modem.

### 2.3.1.2 Buffer

A satellite modem may contain a buffer of substantial size to store overflow traffic trying to enter the satellite link. There may be several reasons for using such a buffer - one is the delay before bandwidth becomes available, i.e. the time it takes for resources/bandwidth to be allocated to

the modem. In this period, the modem may want to store the incoming traffic. In addition, some modems use the buffer as a measure of bandwidth requirements. When the buffer utilization gets close to its maximum capacity, this is interpreted as a sign of too little bandwidth available, and the modem request more resources. And vice versa when the buffer is emptying.

Packets residing in the buffer cause increased delay, and hence increased RTT. The additional delay can easily be calculated as data in buffer divided by the rate at which the buffer empties, or for worst case:  $\frac{BufferSize}{DataRate}$ . In [8], where several modems and equipment are evaluated, buffers of up to 400 kB are mentioned. Delay caused by buffer where for some systems measured as high as 5.5 seconds, giving a total of 6 seconds RTT. In addition this delay can be, in contrary to propagation delay, very varying as the buffer fills and empties, giving the transport layer and TCP difficulties, for example in determining appropriate retransmission timeouts (RTO).

### 2.3.2 Packet-loss and bit error

Due to channel degradation caused by fading, interference, noise etc., a satellite link may have a higher average Bit Error Rate (BER) compared to e.g. links utilizing copper and fiber cables. The BER typically lie between  $10^{-4}$  and  $10^{-12}$ , which are much higher than wired links [13], [23], [24], [25], [26], [27]. Packet loss might also occur due to imperfect and variable bandwidth allocation and control in a satellite. If a link is allocated insufficient bandwidth, or the allocation is too delayed, this could lead to overflowing queue/buffer at a satellite modem.

### 2.3.3 Bandwidth Asymmetry

Bandwidth in the forward direction, i.e. from the satellite and down to the earth station, is usually much higher than in the reverse direction. This asymmetry can be in the order of 10:1 or more [13]. Some systems also have no uplink at all, in which return traffic must use a terrestrial network - this is often referred to as a hybrid solution. However, it is important to note that this is not a physical limitation valid for all satellites; it is merely a satellite design decision. The design requirements might also vary between civilian/commercial and military use, where a typical civilian customer usually download more data than he uploads (this conception has been challenged by the increasing use of peer-to-peer technologies like the torrent protocol). It is fair to assume that this characteristic is not present in a military setting, and a symmetric link is more common. Consequently, this characteristic will not be assessed further in this thesis.



## Chapter 3

# Challenges

This section will explain the challenges which arise due to the previous explained characteristics of satellites and TCP. Note that the challenges presented might be de facto solved, or there exist proposed solution. These solutions are presented in Section 4. The challenges can be summarized into two main problems:

- TCP transmits a certain amount of data, restricted by *cwnd* or *rwnd*, and then has to wait for an acknowledgment before continuing. As the delay over a link increases, the later these acknowledgments will arrive - causing the TCP throughput to increase more slowly. This under-utilization gets worse for high bandwidth links as the sender can fill the flight size faster and thus wait longer. This penalty is defined by the product of bandwidth and RTT of the link.
- TCP interprets any lost segment as a sign of congestion and reacts accordingly. For all TCP variants not especially designed for lossy networks this causes the sender to back off and reduce its *cwnd*. This reaction is erroneous if the segment was lost over a lossy (i.e. channel fading) satellite link and will result in under-utilizing of the bandwidth and reduced throughput for the user.

### 3.1 Bandwidth-delay product - Maximum Window Size

The Bandwidth-Delay Product (BDP) tells how much data can be "in-flight", i.e. the flight size, over a link. When this product becomes so large that TCP performance is degraded, the link is called a long fat pipe, and a network containing such a link is called LFN (Long Fat Network). The BDP in such a network sets the minimum size of the buffer at each peer if the peer wants

to "fill the pipe", i.e. maximize throughput. From earlier sections we have seen that the buffer limits the amount of unacknowledged data each peer can handle, consequently limiting the flight size.

A link with a RTT of 528 ms (from previous section) and a bandwidth of 3662 kBps results in a BDP of 1933 kB. This means that the flight size and hence the *cwnd* should be at least 1933 kB. However, if a receiver has such a buffer available, could he signal it to the sender? In Section 2.2.2 we saw that the Window field was 16 bits long, giving the receiver a possibility to signal a window of maximum 65535 bytes - not nearly enough to "fill the pipe". The sender can consequently transmit 65535 bytes, but then has to wait 528 ms (from the first segment was sent) before transmitting again. The resulting (and maximum) TCP throughput over this link can easily be calculated:

$$throughput_{max} = \frac{window\ size}{RTT} = \frac{65kB}{528ms} = 124kbps \quad (3.1)$$

Clearly, this is a serious under-utilization of the available bandwidth, and presents a challenge to achieving effective TCP transmission. Note however, that this is the limit of a single TCP flow and one may assume that a situation where there is only one TCP connection over a satellite link is fairly rare. Several simultaneous TCP connections might still be able to utilize the entire bandwidth.

## 3.2 Bandwidth-delay product - Slow Start duration

As described in Section 2.2.5 the slow start algorithm is used at the start of a transmission, and the IW is set to 1 segment. The *cwnd* will then increase exponentially by 1 segment for each acknowledgment received until a segment is lost. However this has been shown to be too slow and inefficient on high BDP links - especially for transmissions which are short compared to the BDP [28]. For reference, the BDP in the previous section was calculated to 1933kB. Any transmission which transmit less than 1933kB, e.g. HTTP or e-mail transfers, are certain to be inefficiently transferred.

Our goal is to use Congestion Avoidance as soon as possible, since this implicitly means that the links is using close to the maximum bandwidth available (a segment has been lost). As depicted in [25], the time a connection spends using the slow start algorithm, and the average throughput, can respectively be calculated ("roughly") as:

$$SlowStartTime = RTT * \left( 1 + \log_2 \left( \frac{BDP}{MSS} \right) \right) \quad (3.2)$$

$$SlowStartThroughput = \frac{2 \times B - \frac{MSS}{RTT}}{\log_2\left(\frac{BDP}{MSS}\right)} \quad (3.3)$$

Where MSS is the maximum segment size and  $B$  is the bandwidth of the link. Using values from previous sections and 1460 byte<sup>1</sup> TSS results in a 6 second SlowStartTime, a 0.7 MBps average throughput and a total of 4.2 MB transferred during slow start. Consequently, in our example, shorter transfers than 4.2 MB would in fact never use Congestion Avoidance and never utilize the entire bandwidth. Figure 3.1a shows the duration of the slow start algorithm (Eq. 3.2 ) as a function of bandwidth, while Figure 3.1b shows data transmitted (Eq. 3.2 multiplied with Eq. 3.3 ). And the latter can be used as a reference to find the smallest transfer which can potentially utilize the given bandwidth. The difference between high and low BDP links are clearly shown, and it is evident that a transfer using LAN will almost certain exit slow start (utilize most of the bandwidth) and use congestion avoidance. However, one could assume that a single TCP transmission over satellite would rarely get access to as much as 18 MBps bandwidth, since satellite links usually are divided among several users, making the upper values of bandwidth for GEO possibly irrelevant.

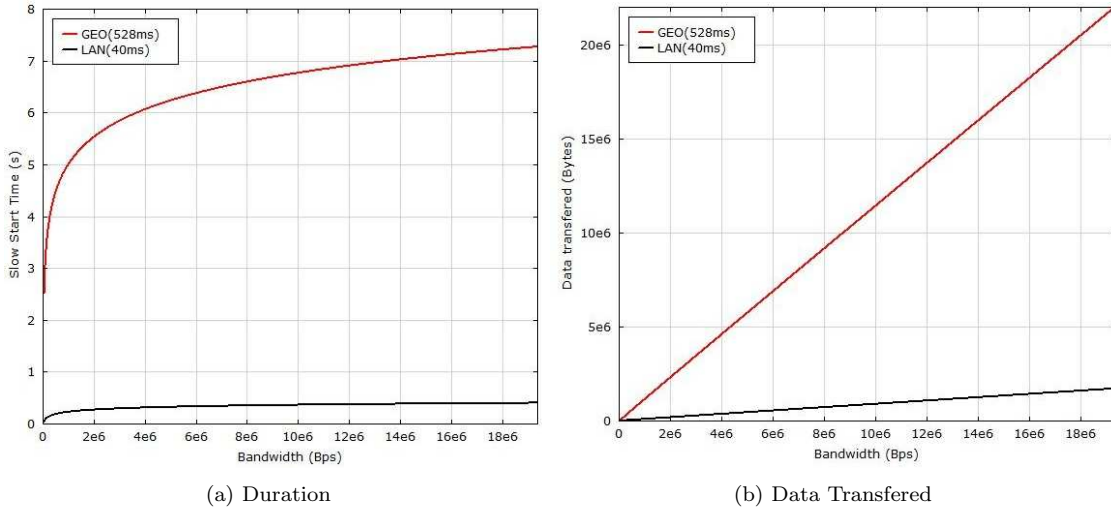


Figure 3.1: Duration, and data transferred, during slow start

The results would get even worse if the transmission utilized Delayed ACK, described in Section 2.2.4. This causes the receiver to wait for maximum 500 ms (waiting for a new segment) before

<sup>1</sup>1500 byte is the Maximum Transmission Unit for Ethernet. IP and TCP headers are minimum 20 bytes each, giving a maximum of 1460 bytes Segment Size without fragmentation, jumbo frames, or similar techniques.

sending an acknowledgment. The result is even longer slow start duration:

$$SlowStartTime = RTT * \left( 1 + \log_{1.5} \left( \frac{BDP}{MSS} \right) \right)$$

### 3.3 Segment loss

The TCP protocol was designed to be used in networks with low bit error. Thus it was fair by the designers to assume that when a segment did not arrive, it was caused by a congested router somewhere in the network which dropped the packet. Especially older TCP flavors, like Tahoe and Reno, penalize the *cwnd* harshly when a segment is lost. The algorithms always interpret the loss as if it stems from congestion and conservatively backs off to avoid network congestion. Many newer flavors make the same assumptions, but reduce *cwnd* in a different manner - these will be described later. For reference, we describe Tahoe and Reno:

- **TCP Tahoe:** Implemented the scheme described in "Congestion Avoidance and Control". When a segment is lost (a timeout of 1-2 seconds has passed), the *ssthresh* is set to half the current *cwnd*, and the *cwnd* itself is set to 1 segment. Since  $cwnd \leq ssthresh$  this causes TCP to enter slow start. As described in Section 3.2, this will penalize the throughput. Three duplicate ACKs will trigger Fast Retransmit which cause the sender to immediately retransmit the lost segment before reducing *cwnd*, *ssthresh* and entering slow start.
- **TCP Reno:** As with TCP Tahoe, if a timeout occurs the *cwnd* is set to 1, *ssthresh* is set to  $cwnd/2$  and slow start initiated. However, when receiving three duplicate ACKs, Reno utilize an additional algorithm called Fast Recovery which is initiated after Fast Retransmit. With Fast Retransmit the *cwnd* is "only" halved, and *ssthresh* is set to  $cwnd/2$ , such that  $cwnd = ssthresh$ . In addition, *cwnd* is increased for each duplicate ACK received. Consequently, this causes the transmission not to enter slow start but remain in congestion avoidance.

If the lost segment is caused by network congestion, the actions above are rational. The throughput is penalized, yet it avoids swarming the network with data and causing situations like the 32kbps-to-40bps throughput drop between LLC and UC Berkeley in 1986, observed by Van Jacobson and Karels. However, when a segment is lost due to bit errors, there would be no need to back off. The result is an unnecessary drop in throughput. Corrupted packets does not cause serious problems in most wired network since they *a)* have a very low BER, and *b)* have low latency. The low latency allow them to quickly increase the *cwnd* after a loss (Figure 3.1a ), thus decreasing impact on the performance.

The situation over a satellite is quite opposite: both BER and latency is significantly higher. In other words, the throughput drops occur more frequently and last for a longer period. Tahoe



halves the *ssthresh*, sets *cwnd* to 1 segment and initiates slow start. The performance would then be equal to a brand new connection, as seen in Figure 3.1. In addition, the *ssthresh* is at least half the BDP (maximum), and this lowered *ssthresh* will cause the connection to use congestion avoidance on a earlier state, thus causing even slower increase in *cwnd*.

A worse scenario is a packet loss during an initial slow start. This would cause TCP to enter congestion avoidance, and increase the *cwnd* even slower as described in Section 2.2.5. The result would be a TCP connection with very poor throughput which slowly improves.

Shown earlier, TCP Reno is an improvement compared to Tahoe, since its back-off is much more liberal than the "*cwnd* = 1"-behavior of Tahoe (with respect to performance over long fat pipes). But Reno and even further developments like NewReno, Vegas, Westwood etc. are prone to reduced performance on lossy links. Research and numerous simulations shows that even a "small" ( $\sim 0.1\%$ ) Packet Error Rate (PER) could have a huge effect on their throughput - some select results may be seen in [29], [30], [31] and [32]. Note the difference between BER and PER: A PER of 1% *could* result from a BER as low as  $8.3e-7$ : 1 bit error in every hundred packet, where each packet is 1500bytes, i.e. 12000bits.

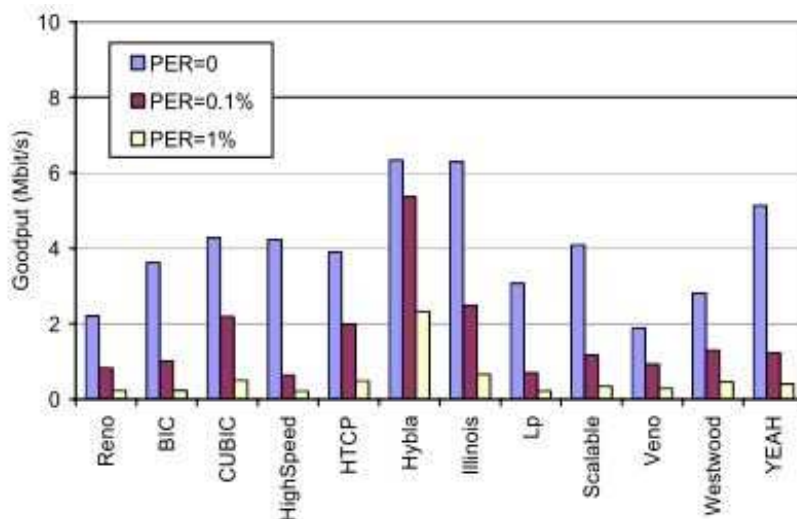


Figure 3.2: Wireless link errors impact: goodput after 180 s of a GEO satellite connection (RTT = 600 ms) at different PER values (0 %, 0.1 % and 1 %). Adopted from [30].

Figure 3.2 shows measurements from [30] for different PER values. RTT was 600ms, bandwidth was 10 Mbps, and the measurements were taken after 180 s. The results show the devastating impact on throughput induced by PER. For all of the TCP flavors (except Hybla) the drop was in a factor of 2 or more, even at PER = 0.1%. And at PER = 1% the performance was drastically reduced to only a fraction of the available bandwidth. The authors of [30] also explain this by the slow *cwnd* reopening, caused by the long RTT.

### 3.4 RTT Unfairness

Measurements have shown that TCP connections with high RTT is heavy penalized in heterogeneous networks [30]. E.g. two TCP senders in the same subnet where both are sending data to the same receiver. The path used by one of the sender contains a satellite link, while the other path is only wired. In such a scenario the connection using the low-RTT link will get favored in terms of bandwidth. Figure 3.3 shows measurements from [30] illustrating this effect for a wide range of TCP flavors. The study used 5 short-RTT connections competing with a 6th long-RTT (300 or 600ms). An ideal situation would be a fair division of bandwidth between all 6 connections - about 1.67 Mbps each.

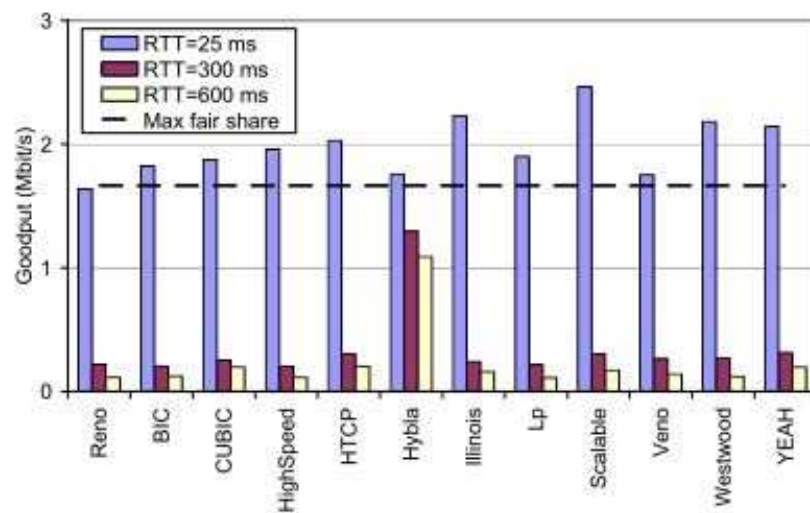


Figure 3.3: RTT unfairness problem: goodput after 180 s of a satellite connection (variable RTT, PER = 0 %) in presence of 5 short RTT terrestrial background connections (Reno, RTT = 25 ms, PER = 0 %). Adopted from [30].

## Chapter 4

# Proposed Solutions

This section will describe existing solutions to the challenges mentioned above. The solutions can be divided into two main categories:

- **TCP enhancements:** Changes made to the standardized TCP to improve its performance, this results in new mechanisms or entirely new TCP flavors. These solutions only require changes on the end hosts, and maintain the end-to-end principle of TCP. Some mechanisms require sender-side modifications only, while other requires both sides. Examples are Hybla, CUBIC, Westwood, Selective ACK etc. Also optimizing of TCP parameters fall into this category, e.g. increasing initial window, Window Scale Option etc.
- **Performance Enhancing Proxies (PEP):** PEPs are intermediary nodes in the network which intercept the traffic and does "something" to increase the traffic performance. There are several different categories of PEPs which all have different mechanism for improving the performance, these will be discussed later. To illustrate the general concept, consider a simplified example with a PEP before and after a satellite link. The PEPs would intercept the un-encrypted TCP traffic, and transmit it over the satellite utilizing a satellite-friendly protocol (TCP flavor or entirely different transport protocol). The receiving PEP would then forward the traffic using TCP. The TCP peers are usually unaware of any presence of PEPs and their operation. Consequently, this approach violates the end-to-end principle of TCP. In addition, the PEP is dependent on TCP header information, which may be encrypted, e.g. if IPsec is utilized. Still, PEPs are the most widely adopted solution [30].

## 4.1 TCP enhancements

Enhancements to TCP can be very difficult to implement because of the interoperability requirement with existing standardized TCP implementations. It would do no good to invent a brilliant TCP version and install it on a server if there are no clients which can understand its TCP "dialect". One way to achieve wide implementation for a new TCP flavor is following a standardization path - however this can be assumed to be very time consuming. This can be avoided if one has control over all TCP peers, e.g. military, corporate or diplomatic networks, and install a new TCP flavor on all hosts and servers. In other words, *if we control all peers communicating through the satellite, we could implement which ever TCP flavor we want*. Sadly, this is rarely the situation. In addition, it would make the use of commercial solutions in a network difficult since it would require modifications to be interoperable.

If the changes made to TCP are interoperable with existing standards, implementation becomes much more agile. One example is TCP CUBIC (not standardized by the IETF) which is the default implementation in the Linux operating system past kernel version 2.6.19 [7]. Another example is several of the TCP options, which is obviously optional at each end-host. They will be utilized if both peers support and agree to use them during the three-way handshake. Hence we can divide the TCP enhancements into two categories:

- Enhancing mechanisms
- TCP Flavors

Each category will be explained and presented in the following sections.

### 4.1.1 Enhancing mechanisms

This section describes mechanisms which aim to enhance the performance of TCP in various ways. These mechanisms are standalone, in the sense that they in theory may be applied to any TCP flavor. Utilizing them in an existing TCP implementation usually does not "result" in a new flavor. Such scenarios are often referred to as for example: "TCP Reno with Window Scale Option enabled". A new TCP flavor however, are often characterized by significant changes from existing implementations, and may or may not utilize one or several of these standalone mechanisms.

#### 4.1.1.1 TCP Window Scale Option

In Section 3.1 we saw that the receiver can signal a window of maximum of 65535 bytes. For a large BDP network this result in a significant underutilization of the available bandwidth as

seen in Equation 3.1. This limit appears due to the mere 16 bits available in the TCP header to signal window size. A proposed solution is the TCP Window Scale Option which is described in Section 2 of RFC-1323, "TCP Extensions for High Performance" [33].

To solve this problem, one might be tempted to expand the Windows Size field in the TCP header. However it is infeasible to implement a new TCP header, so the TCP Window Scale Option is communicated between TCP peers using the options field in the TCP header, see Section 2.2.2. In a SYN-packet, the sender inserts 3 bytes in the options field, stating it is willing to use window scaling on both incoming and outgoing segments. The first two bytes identifies the option ("3" for Window Scale Option), and the length ("3" (bytes)). The last byte identifies a multiplier which is a number between 0 and 14 and specifies how many bits the Window Size should be left-shifted. Practically, this expands the Window Size to 30 bits. If the Window Size is 65535 bytes, and the multiplier is 14, one would multiply 65535 with  $2^{14}$ , resulting in a maximum window of 1073725440 bytes - larger than 1 Gigabyte. Inserting this new value into Equation 3.1 yields:

$$throughput_{max} = \frac{window\ size}{RTT} = \frac{1073MB}{528ms} \sim 16Gbps \quad (4.1)$$

This is a major improvement compared to the previous limit of 124kBps. However, as the bandwidth in future satellites increases, this limit may prove too small in some special scenarios with very few users. I.e. a single flow has access to more than 16 Gbps over the satellite link.

Using the emulation setup described in Chapter 5, we can measure the effect of the mechanism. Figure 4.1 and 4.2 shows result from a FTP transfer between two computers running Windows 7, and are measured over a 2 Mbps link, with 528 ms Round-Trip Time (RTT). The maximum goodput<sup>1</sup> of the link is 1.877 Mbps, or 235 kBps. (see Section 5.7 for details). The link has a Bandwidth-Delay Product (BDP) of 132 kB, i.e. the Receiver Window (*rwnd*) should be at least 132kB for the transmitter to be able to "fill the pipe". A receiver is not able to signal such a large *rwnd* without using scaling. Figure 4.1 shows the effect of this limit as the receiver signal a maximum *rwnd* of 65520 bytes (Windows 7). The sender ends up waiting for acknowledgments, which greatly penalizes the goodput.

Figure 4.2 shows an identical transfer, with Windows Scale option enabled. It is evident that the limit on goodput is imposed by the link bandwidth and not by the *rwnd*, as the goodput reaches the theoretical maximum goodput of the link.

This option is enabled by default on all the most popular operating systems i.e. Linux (since

---

<sup>1</sup>Goodput is defined as the rate of correct data delivered to the Application Layer, i.e. the correctly received amount of TCP segment payload per time unit.

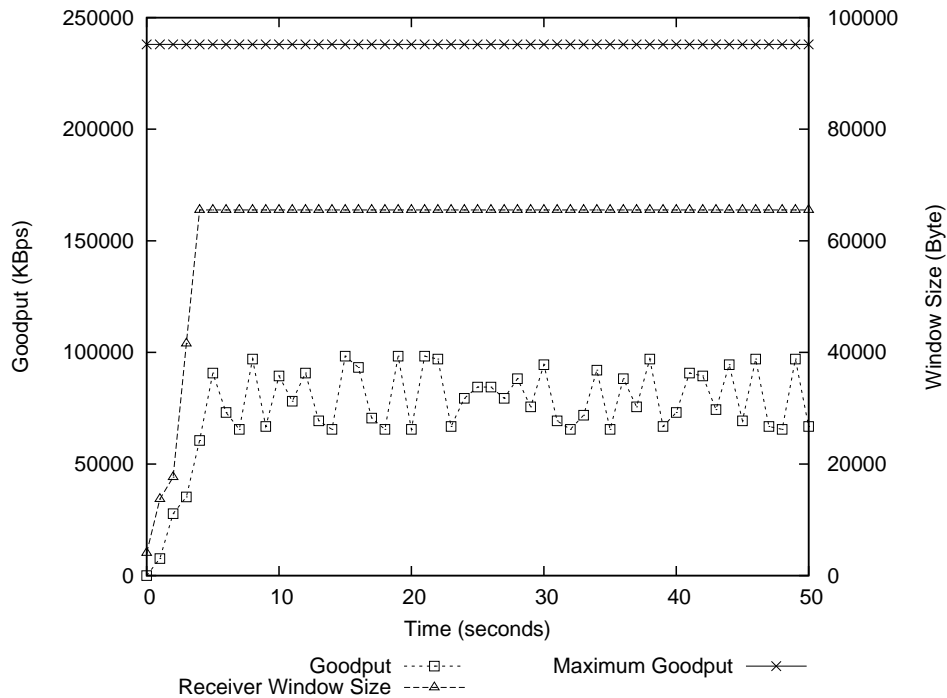


Figure 4.1: FTP transfer over high BDP link with TCP Window Scale Option disabled

version 2.2), Windows (for Windows 7, see Figure 4.6 ) and Mac OS X. It is also *recommended* in RFC-2488: "Enhancing TCP Over Satellite Channels using Standard Mechanisms" [20] - which has a status of Best Current Practice.

#### 4.1.1.2 TCP Timestamps Option

Timestamps is used for two mechanisms: Round-Trip Time Measurement (RTTM) and Protect Against Wrapped Sequences (PAWS). These mechanisms are often considered in conjunction with the TCP Window Scale Option, especially PAWS. They are both *recommended* in RFC-2488, which calls them "companion algorithms" to the Window Scale Option. Timestamps, RTTM and PAWS are (as Window Scale Option) described in RFC-1323.

The TCP Timestamp Option (kind: "8") lets the sender put a 4-byte timestamp in the TCP header option field. The timestamp is the current value of the timestamp clock at the transmission of the sender. Note that this is not the actual time-of-day, but a timestamp clock which values are proportional to a real clock. The receiver will then echo back this timestamp in the ACK segments.

RTTM subtracts the current value from the timestamps clock from the received/echoed times-

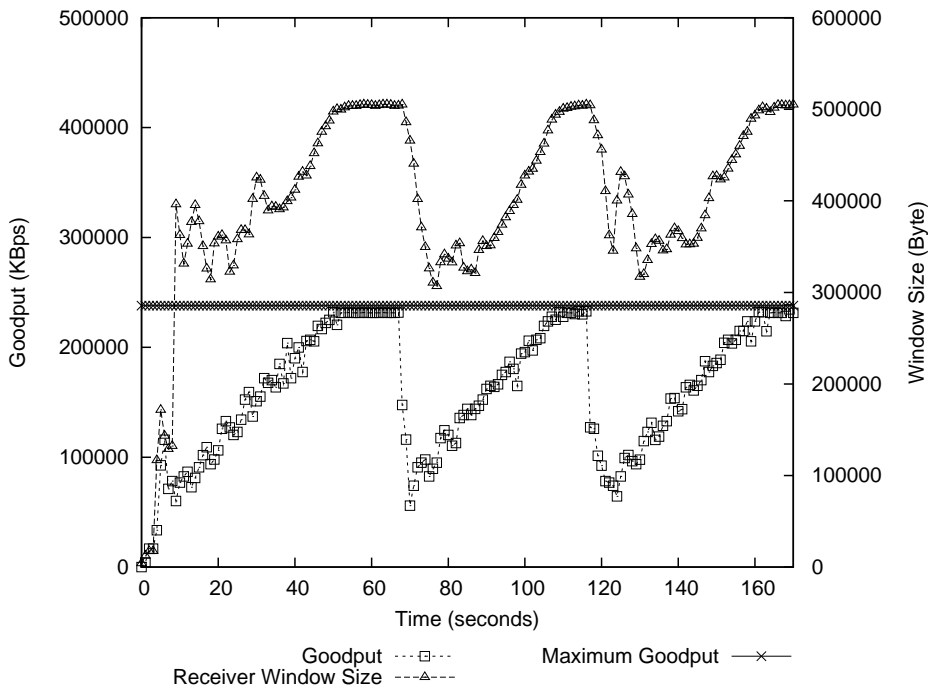


Figure 4.2: FTP transfer over high BDP link with TCP Window Scale Option enabled

tamp, and thus have calculated an accurate RTT. This can be done for every segment, also retransmissions. For most TCP flavors, RTT measurements is done for one segment per window, by measuring the time it takes between a segment is sent until its ACK is received. This is sufficient when the window is small. However, RFC-1323 states that a higher frequency of sampling is needed to assure correct values for RTT when the window is large.

PAWS use the timestamp for a completely different challenge. A fundamental assumption in TCP is that the sequence numbers uniquely identify a byte, i.e. there will never exist two bytes with the same sequence number in the network at the same time (for our flow). The network layer "guarantees" that a IP packet (containing a segment) will stay in transit no longer than two minutes [14]. Consequently, if a TCP peer sends a byte, it cannot send a byte with the same sequence number for at least two minutes. This limits the maximum data rate to 286 MBps or 2.2 Gbps [23]. As a larger window facilitates higher throughput, this limit needs to be addressed: PAWS uses the timestamp along with the sequence number to uniquely identify and distinguish bytes with the same segment number.

Windows supports timestamps, RTTM and PAWS, but has different policies on utilizing it. For example, in Windows Server 2003 it is enabled by default, while in Windows 7 it is disabled. In Linux, it is enabled by default since version 2.2. Newer versions of Mac OS X also use it by default.

### 4.1.1.3 Increasing Initial Window (IW)

As discussed in Section 3.2 the slow start algorithm often increases the Congestion Window (*cwnd*) too slow, resulting in under-utilizing of the available bandwidth. This is especially apparent when transfers are short compared to the BDP. One approach to reduce the duration of the slow start algorithm is to increase the initial value of the *cwnd*, known as Initial Window (IW). This would cause more segments to be sent during the first RTT of transmission. Consequently, more ACKs will be returned to the sender, and this results in a more rapid increase of *cwnd*. The value of IW was initially set to  $1 \times$  Maximum Segment Size (MSS). However, research has shown major throughput improvements on TCP transfer over satellite links using larger IWs, see for example [34] and [35].

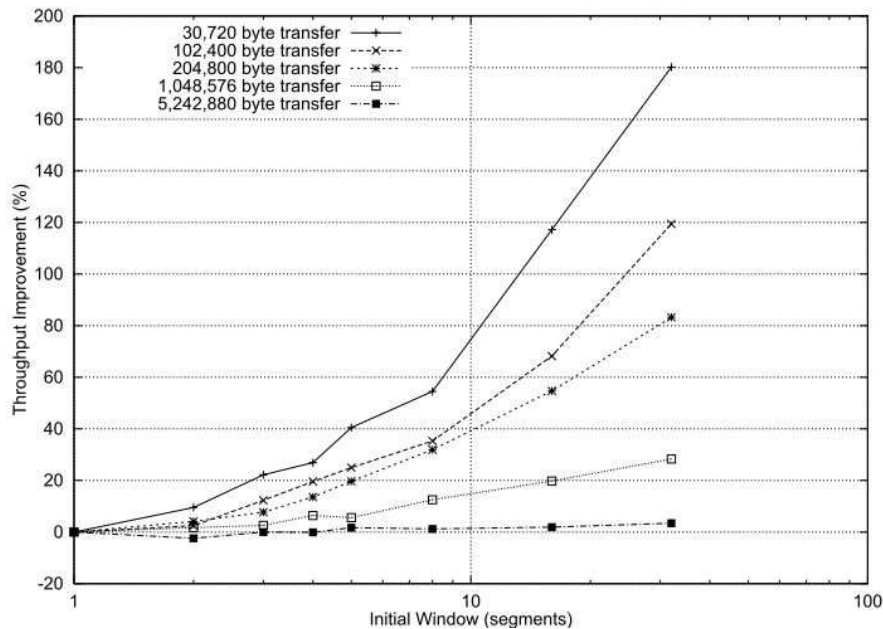


Figure 4.3: Improvement of throughput, compared to  $IW = 1 \times$  MSS, for different values of IW. Adopted from [34]

Figure 4.3 shows the benefits of increased IW across a 560 ms delayed link. It is evident that even small increases to IW results in significant improvement. Note the difference between short and long transfer, and the fact that for longer transfers, a larger IW does not results in any significant improvement, as expected.

There are several proposals to a standardized value available, and currently the most up-to-date (October 2002) RFC is RFC-3390: "Increasing TCP's Initial Window". It proposes an increase



of IW up to  $4 \times MSS^2$  [36], which is widely used today [37].

Using Wireshark<sup>3</sup> one may easily find the Initial Window (IW) behavior of any operating system. Figure 4.4 and 4.5 shows Wireshark captures of the first segments exchanged in a FTP transfer. Common for both is the first three segments which constitutes the three-way handshake, i.e. setting up the TCP connection. Following the setup is the first transfer of actual data - the amount of segments sent is governed by the IW. Figure 4.4 shows the default IW of Windows 7 (tests shows same IW for Windows 7 with Compound TCP enabled) being 2, while Figure 4.5 shows a value of 3 segments for Ubuntu 10.10. The value for Ubuntu is in accordance with the values given in RFC-3390, and the Ubuntu manual<sup>4</sup> actually states it uses the algorithm in RFC-2414, which has been obsoleted by RFC-3390 (but the upper limit for Initial Window remained the same).

No.	Time	Source	Destination	Protocol	Info
2	*REF*	10.0.0.3	192.168.0.30	TCP	ftp-data > 63530 [SYN] Seq=0 win=8192 Len=0 MSS=1460 WS=8 SACK_PERM=1
4	0.532646	192.168.0.30	10.0.0.3	TCP	63530 > ftp-data [SYN, ACK] Seq=0 Ack=1 win=8192 Len=0 MSS=1460 WS=8 SACK_PERM=1
5	0.532725	10.0.0.3	192.168.0.30	TCP	ftp-data > 63530 [ACK] Seq=1 Ack=1 win=16640 Len=0
8	0.543351	10.0.0.3	192.168.0.30	FTP-DATA	FTP Data: 1398 bytes
9	0.543357	10.0.0.3	192.168.0.30	FTP-DATA	FTP Data: 1398 bytes
10	1.089083	192.168.0.30	10.0.0.3	TCP	63530 > ftp-data [ACK] Seq=1 Ack=2797 win=17408 Len=0

Figure 4.4: Wireshark capture showing default Initial Window in Windows 7 (TCP NewReno)

No.	Time	Source	Destination	Protocol	Info
333	*REF*	192.168.0.2	10.0.0.3	TCP	ftp-data > 49224 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=113159936 TSER=0 WS=6
335	0.531538	10.0.0.3	192.168.0.2	TCP	49224 > ftp-data [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=8 TSV=45310255 TSER=113159936
336	0.531558	192.168.0.2	10.0.0.3	TCP	ftp-data > 49224 [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=113160069 TSER=45310255
338	0.531979	192.168.0.2	10.0.0.3	FTP-DATA	FTP Data: 1386 bytes
339	0.531992	192.168.0.2	10.0.0.3	FTP-DATA	FTP Data: 1386 bytes
340	0.532004	192.168.0.2	10.0.0.3	FTP-DATA	FTP Data: 1386 bytes
348	1.076511	10.0.0.3	192.168.0.2	TCP	49224 > ftp-data [ACK] Seq=1 Ack=2773 Win=17152 Len=0 TSV=45310310 TSER=113160069

Figure 4.5: Wireshark capture showing Initial Window in Ubuntu 10.10

There is also an ongoing work, initiated by Google Inc., aiming to increase the IW to  $10 \times MSS$ . This is documented in an Internet draft: "Increasing TCP's Initial Window" [38], and in a corresponding article "An Argument for Increasing TCP's Initial Congestion Window" [37].

#### 4.1.1.4 Selective Acknowledgment

The Selective Acknowledgment option is described in RFC-2018 [39]. As described in Section 2.2.4, TCP acknowledges a received segment with an ACK segment containing the sequence

<sup>2</sup>The actual algorithm reads:  $\min(4 * MSS, \max(2 * MSS, 4380 \text{ bytes}))$  For Ethernet with MTU at 1500 bytes and MSS at 1460 bytes, this will result in an IW of  $3 \times MSS$ .

<sup>3</sup>Wireshark is the world's foremost network protocol analyzer. It lets you capture and interactively browse the traffic running on a computer network. It is the de facto (and often de jure) standard across many industries and educational institutions." www.wireshark.org. Further described in Section 5.6

<sup>4</sup><http://manpages.ubuntu.com>

number of the next segment it anticipates to receive. However, the receiver will only send the ACK if the received segment is at the left (the next segment number) of the receiver window, i.e. it was the segment it was expecting to receive. Consequently, if the expected segment was lost, but the preceding segments arrive, the receiver will send a duplicate of the previous ACK. The sender will retransmit the lost segment, and either continues with the next segments (which may have already arrived) or wait for an ACK to see how many segments actually arrived. Either way, TCP uses one RTT to recover for each lost segment in a window. If multiple segments were lost in the same window, the procedure above would have to be repeated, causing TCP to use several RTTs before recovering. Since this restricts the flow of ACKs, it would penalize the expansion of *cwnd* and the overall throughput of the transfer. Especially for a long-RTT satellite link, this would be devastating. In addition, TCP over satellite may utilize TCP Window Scale Option described in Section 4.1.1.1; larger windows increase the possibility of more lost segments existing within the same window.

A proposed solution is Selective Acknowledgments (SACK). Once the traffic flow has begun; if the receiver experiences lost segments in a window, it will send an ACK containing (in the Option field) a list of all received segments, regardless of their position in the receiver window. Consequently, this identifies all segments which have not been received. The sender can then retransmit all the lost segments and continue transmitting at the correct segment number, resulting in a period of 1 RTT of penalty for up to 4 missing segments<sup>5</sup> in the same window. SACK utilizes the Option field in the TCP header. During connection establishment, the TCP sender will send a SYN segment with the SACK-Permitted Option (kind: "4") set in the Option Field - telling the receiver it may use SACK.

The Selective Acknowledgment option is widely accepted and is *recommended* in RFC-2488. The option is enabled by default in Linux (since kernel 2.2) and in newer versions of Mac OS X. Wireshark captures (Figure 4.6) shows this is enabled by default in Windows 7 as well.

## 4.2 TCP Flavors

This section will present the TCP flavors which are selected for evaluation and for testing through emulation. It is important to note that each variant implicitly uses most of the mechanisms explained throughout several of the previous sections. This section will focus on the relevant and unique characteristics of the flavors, which usually is the congestion algorithm. All flavors are sender-side modifications only, and are compatible with existing TCP implementations. The four selected TCP variants are:

---

<sup>5</sup>This limitation is due to the maximum 40 byte large Option field. If other options are in use, this amount may decrease.

```

Frame 3067 (66 bytes on wire, 66 bytes captured)
Ethernet II, Src: usi_98:d2:61 (00:21:86:98:d2:61), Dst: HewlettP_a0:f3:8d (00:1c:c4:a0:f3:8d)
Internet Protocol, Src: 158.37.91.31 (158.37.91.31), Dst: 158.37.91.30 (158.37.91.30)
Transmission Control Protocol, Src Port: 57450 (57450), Dst Port: http (80), Seq: 0, Len: 0
  Source port: 57450 (57450)
  Destination port: http (80)
  [Stream index: 77]
  Sequence number: 0 (relative sequence number)
  Header length: 32 bytes
  Flags: 0x02 (SYN)
  window size: 8192
  Checksum: 0xdc09 [validation disabled]
  Options: (12 bytes)
    Maximum segment size: 1260 bytes
    NOP
    window scale: 2 (multiply by 4)
    NOP
    NOP
    SACK permitted

```

Figure 4.6: Wireshark capture shows Selective Acknowledgment enabled by default in Windows 7.

- TCP NewReno
- CUBIC
- Compound TCP (CTCP)
- TCP Hybla

In addition, Space Communications Protocol Specifications (SCPS) - Transport Protocol (SCPS-TP), will be presented for reference and comparison. This is a set of specifications which makes modifications to TCP and utilize several of the available options. However, in most literature this is depicted as in a different category than TCP, i.e. not a TCP flavor but a new protocol. The SCPS-TP is tailored for space and satellite environments, and was earlier standardized by the USA Department of Defense (DoD), but is today a ISO standard. The focus on space communication and its military background and standardization makes this protocol a relevant topic.

#### 4.2.1 TCP NewReno

TCP Reno is by some called the reference TCP flavor, and it has enjoyed wide employment since it was released. A few changes have been proposed to its Fast Recovery algorithm, and Reno with these modifications in place are called TCP NewReno. NewReno is described in RFC-2581 [40] and RFC-3782 [41]. As mentioned in Section 3.3, TCP Reno introduced a new mechanism called Fast Recovery, and also utilizes Fast Retransmit seen in TCP Tahoe.

*Fast Retransmit* utilizes the duplicate ACKs which a TCP receiver will send if it receives a segment with a higher sequence number than it expects, i.e. a segment has most likely been lost.

If the TCP sender receives three duplicate ACKs, it assumes the segment is lost, and immediately retransmits the lost segment, without waiting for the segment timeout timer to expire. With TCP NewReno (and Reno), the sender then enters Fast Recovery.

*Fast Recovery* will not (as with TCP Tahoe) set  $cwnd = 1$  and initiate slow start. It will keep the connection in Congestion Avoidance, set  $ssthresh$  to  $cwnd / 2$ , and cut  $cwnd$  in half. In addition, for each duplicate ACK it receives, Fast Recovery increases the  $cwnd$  by one segment. This will diminish the penalty on the throughput since the sender allows itself to send more segments in spite of the fact that no acknowledgments have been received. Fast Recovery ends when an ACK for all retransmitted segments has been received - in which the sender will stay in Congestion Avoidance and  $cwnd$  set to  $ssthresh$  (which was set to  $cwnd / 2$  when three duplicate ACKs was received).

When multiple segments are lost in the same window of data, the sender can receive *partial ACKs*, an ACK that acknowledge some, but not all of the retransmitted data. Using the old Fast Recovery (used in TCP Reno), this would cause the sender to exit Fast Recovery. But duplicate ACKs would still be received, causing a re-entry to Fast Recovery (and halving of  $cwnd$ ), which eventually would lead to very small  $cwnd$  and a timeout. NewReno interprets this as a loss of the segment sent immediately after the segment acknowledged in the partial ACK. It thus retransmits this segment and stays in Fast Recovery until all lost data is acknowledged.

It should be noted that the Fast Recovery behavior when receiving duplicate ACKs is not necessary when utilizing Selective Acknowledgments (see Section 4.1.1.4), since the sender will know which segments is lost, and may immediately retransmit them. Selective Acknowledgments are default in most newer operating systems, but since the mechanisms acquire both sides to support the option, scenarios may often arise where the NewReno Fast Recovery is needed.

The increasing of  $cwnd$  during Slow Start and Congestion Avoidance in TCP NewReno follows the scheme outlined in Van Jacobson and Karels "Congestion Avoidance and Control", i.e. it is as "slow" as TCP Tahoe (and as depicted in Section 2.2.5) to increase it  $cwnd$ . The behavior is summarized in Table 4.1.

State	Behavior
Slow start	$cwnd = cwnd + 1$ segment each ACK
Congestion Avoidance	$cwnd = cwnd + 1$ segment each RTT
3 dup. ACK	Fast Retransmit
Timeout	$ssthresh = cwnd / 2$ , $cwnd = 1$ segment
Fast Retransmit	Immediate retransmit lost seg. - $j$ , Fast Recovery
Fast Recovery	$ssthresh = cwnd / 2$ , $cwnd = (cwnd / 2) + 1$ seg. per dup. ACK.

Table 4.1: Overview of TCP NewReno behavior

It is difficult to learn details about TCP implementations in Microsoft operating systems, since the source code is not open like in Linux. Most information is therefore obtained via official and unofficial web pages, and published papers. Windows Vista utilize the brand new (in 2006) designed TCP/IP stack (which included NewReno and CTCP), and the newer Windows Server 2008 also utilize the stack [42]. Consequently, Vista uses NewReno, and it is fair to assume that it is used in Windows 7.

### 4.2.2 CUBIC

CUBIC is described in [16] and since kernel 2.6.19 it has been the default TCP flavor in Linux. It is an improvement over TCP BIC (which was default in kernel 2.6.18), and the authors claims it "(...) simplifies the BIC window control and improves its TCP-friendliness and RTT-fairness."

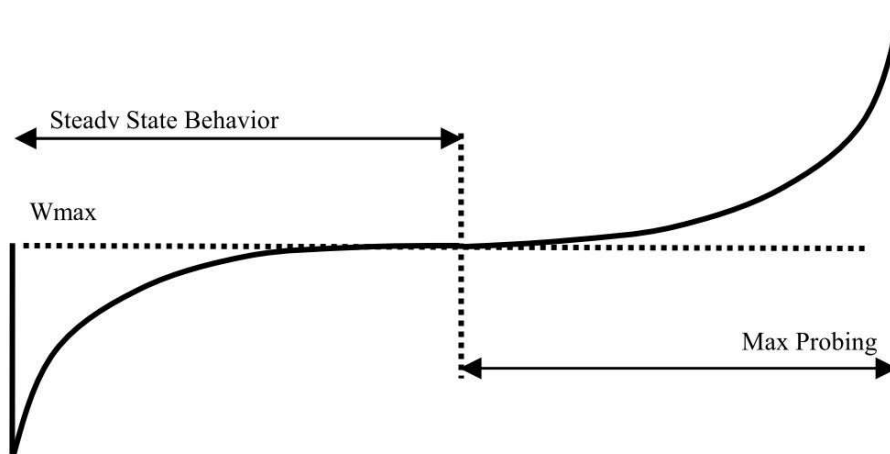


Figure 4.7: The Window Growth Function of CUBIC [16]

The behavior of the Congestion Window ( $cwnd$ ) when loss occurs is significantly different than TCP Reno. One of the most fundamental differences is its independence of the RTT. As discussed,  $cwnd$  in Reno is dependent on ACKs being received, which again is dependent on the RTT. CUBIC on the other hand, increases its  $cwnd$  based on the time elapsed since last a segment was lost.

The  $cwnd$  behavior can be seen in Figure 4.7. When a segment is lost,  $W_{max}$  is set to the value of the  $cwnd$  when the loss occurs.  $W_{min}$  is set by a multiplicative decrease  $\beta$  of  $W_{max}$ . The  $cwnd$  is then set to the midpoint between  $W_{min}$  and  $W_{max}$ . CUBIC assumes that the "correct" window lies somewhere between its current  $cwnd$ , but before  $W_{max}$  (where it "got into trouble" the last time), so it starts increasing the  $cwnd$  rapidly, before slowing down as it gets closer to  $W_{max}$ .

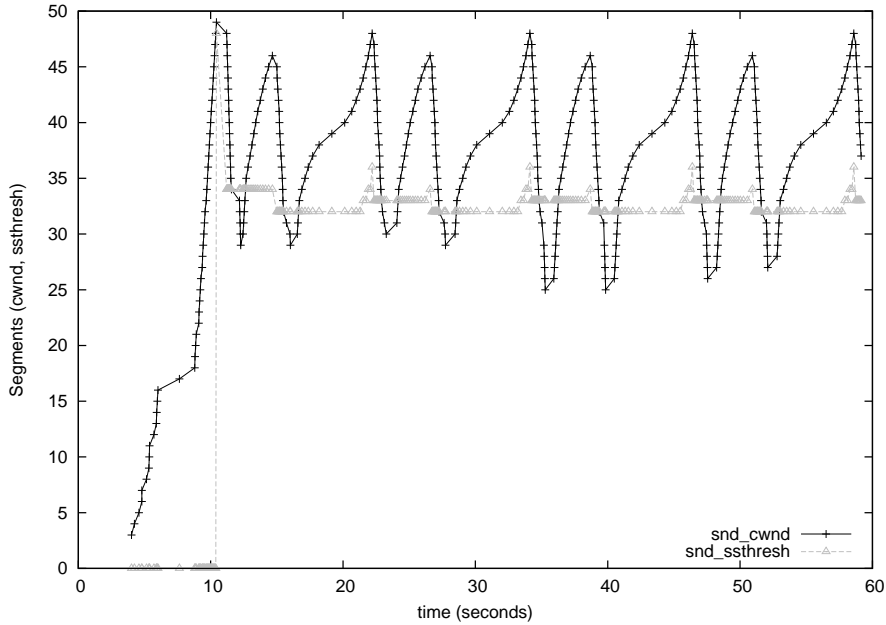


Figure 4.8: *cwnd* of CUBIC during FTP transfer over 500 kbps link with 528 ms RTT

When it reaches  $W_{max}$ , CUBIC assumes the correct maximum is further up and therefore begin to exponentially increase the *cwnd*. This behavior is governed by the following equation:

$$cwnd = C(t - K)^3 + W_{max} \quad (4.2)$$

Where  $C$  is a scaling factor,  $t$  is the elapsed time since last a segment was lost, and  $K$  is:

$$K = \sqrt[3]{W_{max} \times \frac{\beta}{C}} \quad (4.3)$$

Evidentially, RTT is not a part of the equations, making *cwnd* a function of the time since last loss, the  $W_{max}$ , and various constants. The creators point out that for short RTTs, Reno actually increase its *cwnd* more aggressively than CUBIC.

Figure 4.8 shows the actual *cwnd* measured at a Linux TCP sender using CUBIC. *snd\_cwnd* denotes the senders *cwnd* while *snd\_ssthresh* is the slow start threshold described in Section 2.2.5. One may note that every second "period" the *cwnd* seems to be heading for a higher plateau, which should be placed at  $W_{max}$ . This is due to a modification (details in [43]) of the original algorithm when implemented in Linux. The plateau is adjusted based on whether the last  $W_{max}$  is increasing or declining when compared to the previous  $W_{max}$ . It should also be noted that CUBIC has an limitation on how fast the *cwnd* is allowed to grow. When  $W_{max}$  is far away,

or unknown, CUBIC expands *cwnd* aggressively, but the limit will constraint the growth to be linear.

The following are presumably not governed by CUBIC, but is mentioned to correct any misconception. CUBIC regards three duplicate ACKs as a lost segment and immediately retransmit, identical to Fast Retransmit. It also utilize the Slow Start algorithm at the start of a transmission, but has a initial window of 3 segment, compared to 2 segments in Windows 7.

### 4.2.3 Compound TCP

Compound TCP (CTCP), is developed and patented by Microsoft, and is presented in [44] and [45]. The protocol is interesting because it is deployed in several operating systems released by Microsoft. It is enabled by default in Windows Server 2008, while available (but disabled by default) in Windows 7 and Vista. Also, a hotfix is available which adds CTCP capability to Windows Server 2003 and 64-bit Windows XP. A Linux implementation (research purposes only, due to patents) was made by California Institute of Technology (caltech), but due to changes in kernel 2.6.17 it malfunctioned, and support has been discontinued<sup>6</sup>.

CTCP aims to improve performance in high BDP networks, while retaining friendliness to other TCP flavors, and intra-protocol fairness to flows with different RTT. The authors divide efforts to improve TCP into two categories: Loss-based and delay-based. Loss-based TCP schemes use loss to indicate congestion, like TCP NewReno, Cubic, etc. On the other hand, delay-based schemes use variation in RTT to estimate congestion. CTCP proposes to add a delay-based component on top of a loss-based flavor, e.g. NewReno, to create a Compound TCP. The key idea is to aggressively expand the window if excess bandwidth is available, and back-off when the link is getting congested, allowing all flows a fair share. The creators suggest Reno (or other loss-based flavors) does not achieve this sufficiently by itself.

CTCP introduces a new variable called Delay Window, or *dwnd*. This is set by the delay-based component of CTCP. The "old" *cwnd* will remain as usual and is set by the loss-based component, e.g. NewReno, more or less unchanged. The actual window which governs the amount of unacknowledged data in flight is called *win* and is set accordingly:

$$win = \min(cwnd + dwnd, awnd) \tag{4.4}$$

Where *awnd* is the advertised receiver window (denoted earlier as *rwnd*). Initially, *dwnd* is set to 0, which results in CTCP using the regular Slow Start algorithm described in several sections. The delay-based component (i.e. *dwnd* is not enabled until the transmission enters the Congestion Avoidance phase and *win* exceeds a threshold  $W_{low}$ .

---

<sup>6</sup><http://netlab.caltech.edu/lachlan/ctcp/>

The delay-based component measures congestion, i.e. data being buffered using the following equations:

$$ExpectedThroughput = \frac{win}{baseRTT} \quad (4.5)$$

$$ActualThroughput = \frac{win}{sRTT} \quad (4.6)$$

$$Diff = (Expected - Actual) \times baseRTT \quad (4.7)$$

$sRTT$  is the current measured RTT.  $baseRTT$  is the minimal RTT seen this far, up to the connection was started. This is assumed to be the actual RTT of the connection, i.e. no buffer or similar is delaying the data. Dividing the actual congestion window  $win$  with the  $baseRTT$  gives an expected throughput without congesting the network. Actual throughput is calculated using  $sRTT$  which is approximately the current RTT.  $Diff$  is the number of bytes which has entered the network and presumably fills up a buffer. A congestion is detected if  $Diff \geq \gamma$ , where  $\gamma$  is a given threshold. The setting of  $\gamma$  is important, since this decides how many packets should be allowed to be in the buffer. A too low value would cause unnecessary back-off, while a too high would cause unfairness and possible congestion. In [44] this value was set to 30 packets, while in [45] a dynamic algorithm is suggested. It is not known which scheme Windows 7 utilizes, and if dynamic, it is difficult to calculate since it utilizes parameters which may differ for each implementation. For details on the dynamic adjustment of  $\gamma$ , see Section 6.1.1 and [45].

A summary of the behavior of the congestion window  $win$  during CTCP can be seen below. When no delay or loss is experienced, i.e. none of the components detects congestion, the  $win$  obeys the following:

$$win(t+1) = win(t) + \alpha \times win(t)^k \quad (4.8)$$

Where  $\alpha$  and  $\beta$  are tunable parameters.  $win$  is expanded by a factored and powered value of the previous  $win$ . Evidently the expansion of  $wnd$  depends on the implementors choice of parameter values. And when a loss is experienced:

$$win(t+1) = win(t) \times (1 - \beta) \quad (4.9)$$

and  $k$  are all tunable parameters. Note the multiplicative decrease in case of a packet loss, which differ (if not set to 0.5) from the behavior of TCP NewReno. One may also express the behavior of the delay-based component; when  $\gamma$  is larger than  $diff$ , i.e. excess bandwidth is available:

$$dwnd(t+1) = dwnd(t) + (\alpha \times win(t)^k - 1)^+, \text{ if } diff < \gamma \quad (4.10)$$

And when increased delay and congestion has been sensed:

$$dwnd(t+1) = (dwnd(t) - \zeta \times diff)^+, \text{ if } diff < \gamma \quad (4.11)$$



$\zeta$  is a tunable parameter governing how rapid *dwnd* should decrease. The authors point out that *dwnd* never is negative, which results in the loss-based flavor being a lower bound of the window. Due to the extended use of tunable parameters, it is difficult to outline a specific behavior of CTCP. Generally, the protocol should be more aggressive than the loss-based flavor it is paired with, but still retain the loss-based fairness and friendliness.

#### 4.2.4 TCP Hybla

TCP Hybla is presented and described in [17] from 2004 - making it a fairly young TCP flavor. It aims to provide increased performance in high RTT and lossy networks, e.g. networks with a satellite leg. In addition, it tries to solve unfairness between TCP flows with different RTT.

Hybla proposes new Slow Start and Congestion Avoidance algorithms, and similar to CUBIC they are independent of the RTT. A thorough presentation can be found in [17], and Figure 4.9 present a set of equations from this paper where  $\gamma = ssthresh$ ,  $t_\gamma = RTT \times \log_2 \gamma$ ,  $W$  is *cwnd*,  $B$  is throughput, and superscript "H" denotes Hybla.

$$W(t) = \begin{cases} 2^{t/RTT}, & 0 \leq t < t_\gamma, \quad \text{SS} \\ \frac{t - t_\gamma}{RTT} + \gamma, & t \geq t_\gamma, \quad \text{CA} \end{cases} \quad (2)$$

$$B(t) = W(t)/RTT \quad (3)$$

$$\rho = RTT/RTT_0 \quad (5)$$

$$W^H(t) = \begin{cases} \rho 2^{\rho t/RTT}, & 0 \leq t < t_{\gamma,0}, \quad \text{SS} \\ \rho \left[ \rho \frac{t - t_{\gamma,0}}{RTT} + \gamma \right], & t \geq t_{\gamma,0}, \quad \text{CA} \end{cases} \quad (6)$$

$$B^H(t) = \begin{cases} \frac{2^{t/RTT_0}}{RTT_0}, & 0 \leq t < t_{\gamma,0}, \quad \text{SS} \\ \frac{1}{RTT_0} \left[ \frac{t - t_{\gamma,0}}{RTT_0} + \gamma \right], & t \geq t_{\gamma,0}, \quad \text{CA} \end{cases} \quad (7)$$

$$W_{i+1}^H = \begin{cases} W_i^H + 2^\rho - 1, & \text{SS} \\ W_i^H + \rho^2/W_i^H, & \text{CA} \end{cases} \quad (9)$$

Figure 4.9: Set of equations from Hybla presentation. [17]

Equations marked 2 and 3 is the *cwnd* and throughput of Van Jacobson and Karels Slow Start (SS) and Congestion Avoidance (CA) algorithms used in e.g. NewReno, described in Section 2.2.5. They depend directly on the RTT, as seen in the equations. To remove this dependence, the creators of Hybla introduce  $\rho$  (equation marked 5). This variable is constantly calculated during a transmission as the measured RTT is divided by the RTT ( $RTT_0$ ) of a link in which

we want Hybla to equalize in performance. A reasonable value of  $RTT_0$  would be the RTT of a transmission which include a satellite link, but subtracted the satellite delay penalty. I.e. if the RTT of a path is 600 ms, and 550 ms of this is caused by the satellite link,  $RTT_0$  could be set to 50 ms. If the value is set too low, it will cause Hybla to penalize TCP flows with higher RTT. The normalized  $RTT_0$  will be multiplied with time and the  $cwnd$ , and result in equation marked 6 and 7. The  $cwnd$  and throughput will consequently be independent of time. The final equation (9) shows the resulting function which governs the update of  $cwnd$  as ACKs are received. It should be noted when the  $cwnd$  reaches the estimated BDP of the path, Hybla reverts back to using Congestion Avoidance in equation marked 2.

Hybla uses some of the mechanisms described earlier - timestamps are recommended and Selective Acknowledgment are mandatory. Not earlier mentioned, *Hoe's channel bandwidth estimate* [46] tries to calculate the BDP of a path based on the delay of received ACKs, and set  $ssthresh$  based on this value. This mechanism is also recommended. Lastly, packet spacing (also called TCP pacing) tries to eliminate large bursts of segments which can lead degraded performance, by spacing them out over the RTT [17]. Packet spacing is also recommended, however, most of the evaluations of Hybla presented by the creators in [17] is done without spacing enabled.

#### 4.2.5 Space Communications Protocol Specifications - Transport Protocol

The Space Communications Protocol Specifications<sup>7</sup> (SCPS) was originated by the US DoD and NASA, and further developed by The Consultative Committee for Space Data Systems<sup>8</sup>. It is divided into several parts, and the Transport Protocol (SCPS-TP) provides Transport Layer type service. Initially, the SCPS-TP was a military standard in MIL-STD-2045-44000, but today the specification is standardized in ISO 15893. A end-host and proxy reference implementation is available at <http://www.openchannelsoftware.com/projects/SCPS>.

SCPS-TP is a collection of protocols that provide full reliable service (TCP), best-effort reliability service (TCP with modification), and minimal reliability service (UDP). The protocols are described in [47]. Although the reliable service is based on TCP, it utilizes a wide range of enhancements.

Depending on "mission requirements", the capabilities of the network and the characteristic of the links, hosts *should* and *may* support a wide range of options. A list containing most of these are provided below to give an idea of the difference between TCP and SCPS-TP:

- Window Scale Option

---

<sup>7</sup><http://www.scps.org>, was not available during Spring 2011

<sup>8</sup><http://www.ccsds.org>

- Timestamp Option
- Selective Acknowledgment Option
- Explicit Congestion Notification
- Header Compression Option
- TCP for Transactions Options

Also, SCPS-TP hosts *must* implement the Delayed ACK algorithm. Consequently, not every segment is separately acknowledged, but are sent periodically based on the RTT [27].

In addition to the enhancements above, SCPS-TP adds major improvement to the congestion control. As described in Section 3.3, TCP interprets all losses as congestion. In fact, there are no certain ways of separating losses due to congestion from other losses, e.g. channel fading. How to interpret and react to such unknown segment losses is one of the native TCP challenges. SCPS-TP aims to solve this, by identifying the cause behind a loss, or a potential lossy situation. Roughly, it identifies three scenarios:

- Congestion
- Corruption
- Link outage

When a loss is experienced, and no evidence is present which suggest a specific cause, congestion is assumed. The standard specifies that one of the following congestion control algorithms may be used:

- Van Jacobson's Slow Start and Congestion Avoidance (Section 2.2.5)
- TCP Vegas [15]

In addition to congestion control, SCPS-TP tries to identify losses caused by corruption, as oppose to congestion. This can be identified via "inter-layer signaling, MIB<sup>9</sup>-query, or other mechanism.". When a connection experience loss due to corruption, the sender must send a segment with the Corruption Experienced Option to the receiver. The sender will then take appropriate action in accordance to a corruption (opposed to TCP where loss can be either congestion or corruption (or something else)).

SCPS-TP also identifies link outages as a separates cause for packet loss. This can also be identified via inter-layer signaling or MIB-queries. The sender will then take appropriate action (amongst other, no adjustment to *cwnd*), until the link has been restored.

---

<sup>9</sup>Management Information Base. Database containing a wide array of information on the state of the peer. See [47].

#### 4.2.5.1 TCP Vegas

This section will provide a short introduction to the Vegas congestion algorithm as described in [15]. It consists of three fundamental improvements compared to Reno:

- Faster retransmission
- Anticipate congestion
- Improved slow start

Vegas will use the RTT of duplicate ACKs as an indication if a segment has been lost in transit. If the RTT fulfills the criteria, Vegas regard the segment as lost, and the segment is retransmitted immediately without waiting for three duplicate ACKs.

An inherit problem with Reno is that congestion is not detected until a segment is lost, i.e. the link is already congested. Vegas aim to detect an oncoming congestion and act accordingly before a congestion occurs and packets are lost. The solution is to measure the increased delay of packets as they are forced to hold in buffers when the link is overwhelmed. The algorithm is almost identical to the one described in Section 4.2.3 on CTCP (CTCP is derived from Vegas).

The slow start algorithm works similarly to the congestion avoidance, i.e. it backs off after congestion has happened, which in turn usually result in multiple losses causing reduction of the *cwnd* [15]. The mentioned delay-based solution is applied at the slow start phase (including a few breaks in the expanding of *cwnd* to accommodate valid measurements), backing off as congestion is building up and is detected.

## 4.3 Performance Enhancing Proxies (PEP)

A PEP is an intermediary node in the network path which aims to improve the performance of the traffic, usually by intercepting traffic flow and modifying or adding behavior. PEPs can operate over several OSI layers, including lower ones like the link layer. This paper will focus on proxies operating at the transport layer, often called TCP PEPs because they interact with TCP.

PEPs are usually non-standardized and proprietary, and several commercial solutions are available [48]. RFC-3135 [49] is a survey of the PEP landscape, categorizing and identifying the different types that exist - with a focus on PEPs that operates with TCP. It is classified as informational, i.e. it makes no recommendations on the use of PEP, in fact the authors point out that the use of PEPs should be regarded as an option only when end-to-end approaches are unavailable.

The mechanisms used by PEPs are summarized in [49], and in the section below. Note that an actual proxy may utilize several of these mechanisms to boost performance:

- **TCP ACK Handling:** This lets the PEP manipulate the flow of ACKs between sender and receiver. The behavior can be subdivided into several mechanisms. TCP ACK Spacing aims to reduce the burstiness of ACKs, while Local TCP Acknowledgments lets the proxy transmit premature ACKs to the sender. The latter is used by many implementations, and will be discussed later. Local TCP Retransmission allows the proxy to cache and locally retransmit lost segments. TCP ACK Filtering and Reconstruction lets a PEP create ACKs when an asymmetric link has too small bandwidth in the return-direction to accommodate the flow of ACKs.
- **Tunneling:** Tunneling lets the proxy encapsulate the packet before transmitting across the link. Another proxy at the other end then decapsulates before forwarding the packet.
- **Compression:** This is sometimes the only performance enhancing mechanism included in the proxy. There are several different compression schemes available, and they may be applied at several headers and payloads. Depending on the scheme used, security in form of encryption must be taken into account, and may limit the compression.
- **Connection maintenance:** The proxy might mitigate periods of link disconnection by using a cache or an intermediary protocol which keeps the TCP connection between peers (or between PEP and peer) open. Longer periods of outage will without such a mechanism cause a timeout and closing of *cwnd* with all its drawbacks.
- **Priority-based Multiplexing:** The proxy may perform Quality of Service enforcement, boosting the performance of selected and prioritized traffic flows. This might be done natively, i.e. cache non-priority traffic and transmit high-priority, or it could be done in conjunction with a split TCP scheme.
- **Protocol Booster Mechanisms:** These are different ways to improve specific protocols performance, e.g. adding a checksum to UDP segments to improve error detection. Another example is additional timestamp to measure and control jitter, to assure correct order of delivery.

The mechanism most relevant in this thesis is the Local TCP Acknowledgments which is an important part of the TCP splitting approach. TCP splitting is interesting due to its promising results [31], widespread treatment in literature and research, and the availability of a free, open-source implementation.

### 4.3.1 TCP Splitting

A split connection PEP terminates an initiated TCP connection from a TCP sender at the proxy, and establishes a new TCP session with the intended receiver. Depending on the distribution of proxies, a second proxy may be placed at the receiver side, dividing the TCP connection into three parts. These distributions are called integrated and distributed, respectively, and are illustrated in Figures 4.10 and 4.11.

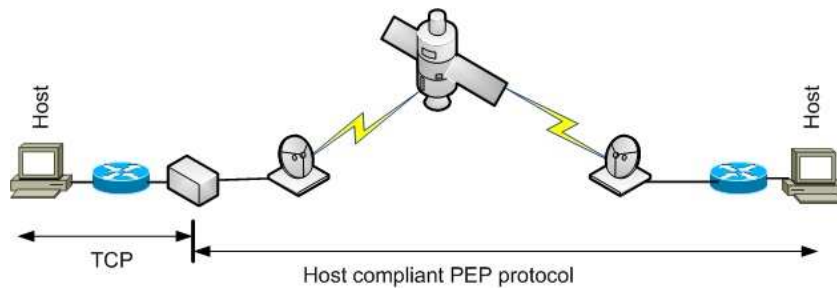


Figure 4.10: Integrated PEP scheme.

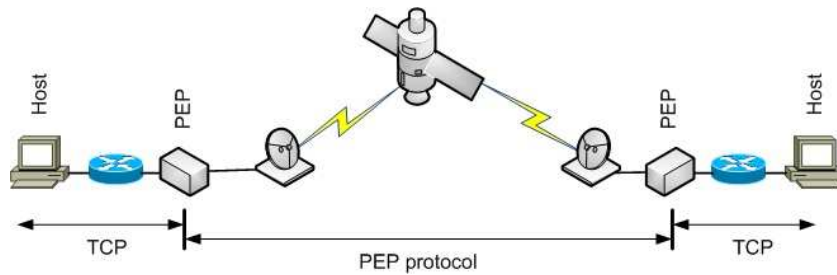


Figure 4.11: Distributed PEP scheme.

The rationale behind TCP splitting is to replace the sub-optimal TCP variant with a protocol tailored to perform optimal over satellite links. Using the distributed scheme allows this protocol to be independent of the abilities and support of the end host, while an integrated scheme must use a protocol supported by the end host. The latter invites to the usage of sender-side modifications of TCP, e.g. the TCP flavors described in previous section.

In conjunction with TCP splitting, spoofed ACKs may be generated. This enables the proxy to send "fake" ACKs, making the sender increase its *cwnd* more rapidly. This again will increase the rate at which the PEP and PEP protocol is fed data. Consequently, the slow opening of *cwnd* for some TCP flavors have been mitigated.

#### 4.3.1.1 Transparency and End-to-end argument

Some proxies are completely transparent, i.e. the hosts are unaware of its existence and operation. While other may require signaling, or modification on one or both hosts. These are regarded as PEPs with some degree of transparency, and are usually harder to implement since they require control of end hosts.

The end-to-end to principle<sup>10</sup> of the Internet states that certain required end-to-end functions can only be correctly performed by the en systems themselves, i.e. the intelligence should be at the end hosts, not hidden in the network. A TCP splitting approach breaks this principle, which is one of the main reasons why PEPs are not recommended for general use [49].

#### 4.3.1.2 Security

It is evident that the TCP Splitting requires the PEP to be able to read the TCP header. This would normally not be a problem; however there are several security schemes which encrypt the TCP header, e.g. IPsec in both tunnel (used in emulation) and transport mode. In such scenarios, TCP Splitting is impossible to employ without any sort of workaround, and is another argument against the use of PEPs.

There exist several solutions, and maybe the simplest and most obvious is to move the PEP in front of the cryptographic device, i.e. out of the black portion of the network. This allows the PEP to inspect the plaintext TCP header before forwarding it to be encrypted. However, to accomplish this, one must either move the black boundary closer to the satellite gateway, or move the PEP farther away. The first option is often unacceptable for security reasons. The second option could lead to sub-optimal performance since the protocol tailored for the satellite link must be used across large portions of the network [6]. Due to its simplicity and ease of deployment this is a feasible solution. Surveys on the performance of the PEP protocol across the wired network portion should be done prior to deployment - this is however outside the scope of this thesis. For networks containing a radio distribution leg, the sender and black boundary are often close to the radio link. A PEP aimed to increase performance in lossy networks could consequently be placed on the red side but still close to the radio link. This could reduce the length of which the tailored TCP flavor must traverse the network and thus maintain the PEPs performance.

Another solution is to terminate the IPsec tunnels at the PEP, creating "red enclaves" inside the black network [2]. This would allow the PEP to be sited close to the satellite gateway, while still keeping the broad black boundaries. However, the solution has several challenges. It breaks the end-to-end security associations, which in turn forces each end host to create security

---

<sup>10</sup>RFC-1958: Architectural Principles of the Internet

associations with the PEP, with all related drawbacks of key management, processing at PEP, etc. In addition the transparency would be reduced since each end host would have to be aware and interact with the PEP. Finally, the end host would have to trust the PEP since it would be able to decrypt the data, and the PEP itself would be an attractive point of attack [50].

There also exists a solution called Split PEP Enabler, which utilizes three PEPs at each side of the satellite. As the others, this solution has several challenges, and the interested reader is referred to [2].

Other solutions include modifications of IPsec to make TCP headers available [50] and a multi-layer IPsec [2] which achieves the same goal. To the best of my knowledge, no available implementation exist.

#### 4.3.2 PEPsal

PEPsal is an open source PEP implementation freely available for download at <http://sourceforge.net/projects/pepsal/>. It is presented, described and tested in [31]. Referring to the sections above, it is a Transport Layer (TCP) PEP, which is transparent and to be distributed following an integrated scheme. It utilizes the TCP Splitting approach and spoof ACKs towards the sender to mitigate TCP penalties at the sender. Since the implementation is based on the Linux operating system, any available Linux implementation of TCP can be used. The authors recommend TCP Hybla (which also is created by the creators of PEPsal) described in Section 4.2.4, which is tailored for satellite environments. The authors also recommend installing the MultiTCP patch for Linux which includes the complete implementation of Hybla.

PEPsal operates as described in Section 4.3.1 on TCP Splitting. It intercepts the incoming SYN packet (segment with SYN flag set), acknowledges it, and establishes a new TCP session using e.g. Hybla with the intended receiver. The payload is then copied and forwarded using the new "tailored" session. PEPsal will keep acknowledging data from the sender at a high rate such that the Hybla session will not be limited by the original senders rate.



# Chapter 5

## Emulation

This section will describe the design, equipment, and tools utilized during the emulation. Configurations of equipment have been described to allow easy reproduction (requested by FFI), and is therefore quite elaborate.

### 5.1 Design

The logical design chosen for the emulation can be seen in Figure 5.1. The topology is a quite simple, and common when testing TCP performance. However, it has two characteristics worth mentioning. Since this evaluation is done with a military scope, the network contains an encryption device. This separates the network into a red and black side to create a more realistic, secure, environment. In addition, a radio link has been added to accommodate a military hybrid network with a radio distribution leg, which is a common scenario. Two PCs (i.e. TCP peers), with different operating systems, is placed at each side of a satellite link. In front of the satellite modem, a cryptographic device sets up a IPsec VPN tunnel, creating the black side of the network.

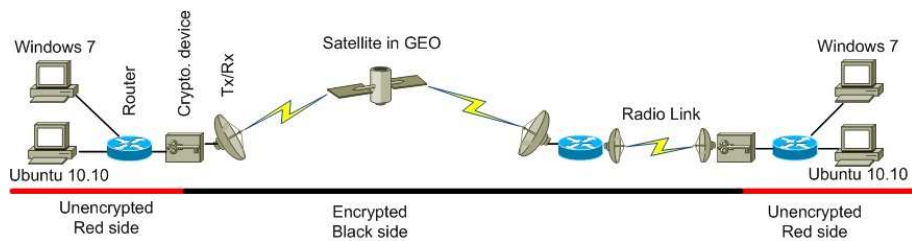


Figure 5.1: Logical design of emulation

Figure 5.2 shows the actual physical implementation of the logical design. The equipment mentioned below, its configuration, and software used, will be presented in following sections. On each end, a tactical router running the Vyatta routing software is placed. The tactical routers are provided, modified, and used by FFI, making the emulation more closely resemble a military network. The routers obviously routes traffic, but also provide the delay and loss using Linux Traffic Control and the built-in network emulator *netem*. A PC between the routers, running FreeBSD and the network emulator Dummynet, limits the bandwidth to 500 kbps. Together, these devices constitute the satellite. In addition, the routers set up the IPsec VPN tunnel (included in Vyatta software) between themselves. Each peer is running out-of-the-box versions of their respective operating systems denoted in the figure. No major software or configuration has been changed or added which can affect their networking performance. All links has been set to 10 Mbps, full duplex (except, obviously, the rate limitation itself), thus creating a "10 Mbps to 0.5 Mbps" bottleneck.

Figure 5.3 shows the design when a Performance Enhancing Proxy (PEP), i.e. PEPsal, is utilized. Note that the PEP must be in the red side of the network since it interacts with the TCP header. Ideally it should be placed as close to the satellite link since it utilize a TCP flavor tailored for satellite links.

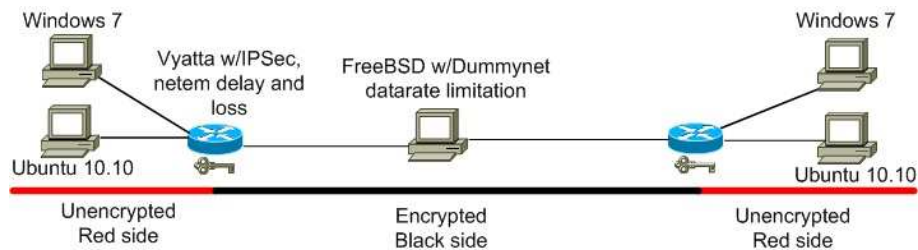


Figure 5.2: Physical design of emulation

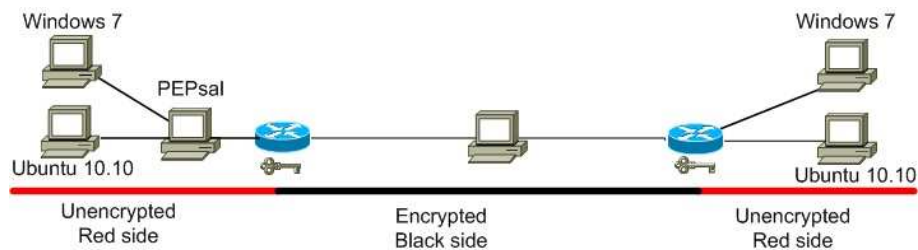


Figure 5.3: Physical design of emulation, with PEP

## 5.2 Vyatta routers

On each end of the "satellite link" there is a router which both routes and encrypt/decrypt data, ref. Figure 5.2. The routers are tactical routers provided by FFI (Forsvarets ForskningsInstitutt/Norwegian Defense Research Establishment), running Vyatta<sup>1</sup> routing software.

Vyatta is an open source network operating system based on Linux. It is freely available for download online, but there also exist a series of commercial products available for purchase. The Vyatta software provides a wide range of basic and advanced services, e.g. routing, firewall, IPsec VPN, DHCP, NAT etc. In the emulation, routing and IPsec VPN was utilized - as well as the Linux network emulator *netem*.

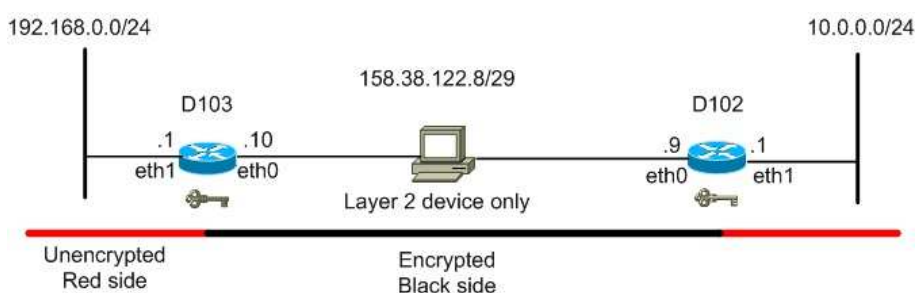


Figure 5.4: Layer 3 design of emulation

### 5.2.1 Configuration - IPsec

The complete configuration for routers D102 and D103 can be found in Appendix A. As mentioned, the routers utilize the IPsec feature and the routing service. The routing configuration is quite trivial and consists basically of static routes giving routing capabilities between the three subnets in Figure 5.4. Documentation and commands used for this configuration can be found in the Vyatta Basic Routing Reference Guide [51]. The resulting routing table (for D102) can be seen below.

```
vyatta@D102:~$ show ip route
```

```
Codes: K - kernel route, C - connected, S - static, R - RIP, O/M - OSPF(MT),
       I - ISIS, B - BGP, > - selected route, * - FIB route, L - OLSR
```

```
S>* 0.0.0.0/0 [1/0] via 158.38.122.10, eth0
```

```
C>* 10.0.0.0/24 is directly connected, eth1
```

```
C>* 127.0.0.0/8 is directly connected, lo
```

<sup>1</sup><http://www.vyatta.org>, <http://www.vyatta.com>

```
C>* 158.38.122.8/29 is directly connected, eth0
S>* 192.168.0.0/24 [1/0] via 158.38.122.10, eth0
```

Between the two routers a VPN tunnel is set up, providing a more realistic network scenario. The tunnel is an IPsec Site-to-Site VPN in tunnel mode. Configuration of this service is not as trivial as routing, however, the Vyatta VPN Reference Guide [52] proves an excellent documentation. The rationale behind the configuration used on the routers can mostly be found in Chapter 2 of [52], section "Configuring a Basic Site-to-Site Connection".

For reproduction, a short walk-through on the configuration of router D102 of the IPsec tunnel will be presented. The first step is to enable VPN on the appropriate interface (see Figure 5.4 for interface notation):

```
set vpn ipsec ipsec-interfaces interface eth0
```

There are two protocols needed to be configured to enable IPsec. The first is Internet Key Exchange (IKE), which establish the IPsec tunnel. The peers will initially try to establish IKE Security Associations (SA). One needs to specify the hash and encryption algorithm used during the first IKE phase, these will be used to establish the IKE SA. When the IKE SA has been established, the Encapsulating Security Protocol (ESP) SA negotiation may commence. As for the IKE SA, the ESP SA needs a encryption and hash algorithm. Obviously, the settings for the SAs must match on both routers. The emulation uses 256-bit AES for encryption and SHA-1 for hashing.

```
set vpn ipsec ike-group IKE-D102 proposal 1
set vpn ipsec ike-group IKE-D102 proposal 1 encryption aes256
set vpn ipsec ike-group IKE-D102 proposal 1 hash sha1
```

```
set vpn ipsec esp-group ESP-D102 proposal 1
set vpn ipsec esp-group ESP-D102 proposal 1 encryption aes256
set vpn ipsec esp-group ESP-D102 proposal 1 hash sha1
```

Next we need to specify the authentication mode (Pre-Shared Key "asd" used), which SA groups should be used (obviously the ones we just created), as well as remote and local IP addresses (see Figure 5.4 ).

```
set vpn ipsec site-to-site peer 158.38.122.10 authentication mode pre-shared-secret
edit vpn ipsec site-to-site peer 158.38.122.10 /(navigates to node)
set authentication pre-shared-secret asd
set ike-group IKE-D102
set local-ip 158.38.122.9
set tunnel 1 local-subnet 10.0.0.0/24
```

```
set tunnel 1 remote-subnet 192.168.0.0/24
set tunnel 1 esp-group ESP-D102
top      /(navigates out of node)
commit  /(enables configuration)
```

As mentioned, the steps above need to be "duplicated" at the other router/end of the tunnel. The configuration should result in an active IPsec tunnel, and two Security Associations, as shown in the output below from D102:

```
vyatta@D102:~$ show vpn ipsec status
IPSec Process Running PID: 3451
```

```
1 Active IPsec Tunnels
```

```
IPsec Interfaces :
    eth0    (158.38.122.9)
```

```
vyatta@D102:~$ show vpn ipsec sa
Peer          Tunnel# Dir SPI      Encrypt   Hash      NAT-T A-Time L-Time
-----
158.38.122.10 1      in  be67de76 aes256   sha1      No    1261 3600
158.38.122.10 1      out 5621d630 aes256   sha1      No    1262 3600
```

The output shows the IPsec tunnel being active and enabled on interface *eth0* with IP *158.38.122.9*. *PID* is the identifier of the IPsec process, as set by the Linux operating system. The second output shows the list of Security Associations, one for each direction of the tunnel. Listed are all properties which were configured. *A-Time* is the time (in seconds) since the association was negotiated. *L-Time* is the maximum lifetime of the association (default 3600 seconds).

All traffic between the two subnets, 10.0.0.0/24 and 192.168.0.0/24 should consequently be encrypted. A capture of the actual traffic, captured "mid-tunnel" at the intermediary delay/rate limitation device can be seen in Figure 5.6b, hence proving the functionality of the configuration and the tunnel.

## 5.2.2 Configuration - Delay and Packet Loss

Since the Vyatta routers run Linux, they also include Traffic Control (*tc*) and *netem* which is a Network Emulation utility controlled by *tc*. *netem* documentation can be found in [53], while *tc* has its own manual. With *tc*, one configures queues or queuing disciplines (*qdisc*) for interfaces.

When the kernel wants to send something out an interface, it inserts the frame into the qdisc "in front" of the interface. At this point, *tc* may perform a wide variety of functions like shaping, different queuing schemes, QoS etc. One may also engage the *netem* utility. The *netem* network emulator can introduce delay, loss, duplication, re-ordering etc. to the segments residing in the qdisc. In the emulation, delay and loss is used. Referring to Figure 5.4, the following commands (example at 1 % PER) are issued at both routers D102 and D103:

```
tc qdisc add dev eth0 root netem delay 264ms loss 1%
tc qdisc add dev eth1 root netem loss 1%
```

Resulting in a total of 528 ms RTT, as according to the design, and 1% of randomly distributed packet loss each direction.

### 5.2.3 Configuration - Bandwidth limitation

It should be noted that large efforts were made by the author to utilize *tc* for bandwidth limitation. *tc* has been utilized successfully at FFI for the purpose of rate limitation. Therefore I intended initially to use *tc* in this emulation as well. However, challenges were met during testing and it was concluded that *tc* would not fulfill the requirements needed for this specific emulation. Dummynet were finally chosen for rate limitation. The purpose of this section is to describe the challenges and experiences drawn from these efforts.

It is not possible to apply both *netem* and the rate limitation of *tc* simultaneously to a interface. I.e. delay and rate limitation cannot be applied simultaneously without some kind of workaround. To solve this, one has to divert traffic into a tree. The tree need to be configured with at least two leafs, since either leafs may be configured with either delay or rate limitation. The traffic will then be filtered such that all traffic enters both leafs. On each leaf, separate rules for delay and rate limitation are applied, and consequently both rules will be applied to all traffic in the tree. An example of the scripts used to create the tree and apply the rules can be found in Appendix D.

However, it was experienced that when using these trees, one lost control of the buffer, i.e. the buffer became very large although the buffer size was specified at the rate limiter. Troubleshooting was first focused on tuning of parameters at the rate limiter, which was Token Bucket Filter (TBF - has its own manual) and Hierarchical Token Bucket (HTB - has its own manual). But it was later discovered, when the rate limiter was applied directly to the interface, that the tree was to blame. Using *tc -s qdisc show dev eth0 root* shows a value *backlog* which is the number of packets currently in the buffer. When a tree was in place, this value would climb to hundreds of packets although the buffer was set to be small at the rate limiter. A solution to control and decrease this buffer was not found.

As can be read in the TBF manual<sup>2</sup> the kernel timer frequency sets limits on how "perfect" the rate limiter can operate. This is especially important if one wants to control the maximum burst, i.e. not allow burst above the configured limit. Summarized, to achieve "perfect" limits the kernel should only transmit one packet per tick of the clock. If the limit is configured higher than the timer "allows", this would force the kernel to treat several packages each tick, and reduce the accuracy. Therefore, efforts were made to increase the timer frequency from 250 HZ (default on Ubuntu) to 1000 HZ. On Linux this requires changes to compiler directives, and re-compiling the kernel. Although successful, it is a time-consuming task.

Finally, unexplained rate limits were experienced. At smaller, but sufficient buffer sizes (according to calculation and manual), the traffic flows were not able to reach the configured rate limit. The rate could however be achieved by first setting the buffer large, and then reduce it during a transfer. An explanation or an error in the rationale or calculation could not be found. Eventually, a choice was made to utilize FreeBSD and the Dummynet network emulator for rate limitation.

#### 5.2.4 Packet Error Rate and Bit Error Rate

As mentioned, *netem* was used to add packet loss to the emulation. However, *netem* only accept the loss-parameter as a Packet Error Rate (PER), while a satellite link operates with Bit Error Rate (BER). Consequently, we must calculate the resulting BER from the PER input to get values which may be applied and compared in a satellite paradigm. The conversion from BER to PER is easily done:

$$BER = 1 - (1 - PER)^{(1/N)} \quad (5.1)$$

Where  $N$  is the length of the packet in bits. In Figure 5.1, the only path for traffic to flow is via the radio and satellite links. Long segments containing data flows one way, and short ACKs flow the other way, both being impacted by the packet loss. Since PER is a function of packet size, we must calculate a separate BER for each direction.

We regard the event of a packet loss in one direction to be independent of the event that a packet is lost in the opposing direction, which allows us to simply summarize the two BERs. Note however that this is an assumption made only for the emulation, and in a real-life scenario this would most likely not be the case. Table 5.1 shows the configured PER and the corresponding BER used in the emulation. Values for  $N$  was 12176 and 1248 bits, or 1522 bytes for data and 158 bytes for ACKs. The rationale behind these values can be found in Section 6.1.

---

<sup>2</sup>man TBF, or <http://linux.die.net/man/8/tc-tbf>

Configured PER	BER
0.01 %	8.835E-08
0.1 %	8.839E-07
1 %	8.879E-06
2 %	1.785E-05
5 %	4.531E-05
8 %	7.366E-05

Table 5.1: Corresponding BER to configured PER in emulation.

### 5.3 FreeBSD with Dummynet

FreeBSD<sup>3</sup> is a open source UNIX-like operating system. It was chosen for this task due to its simple network emulator, Dummynet, and the ability to easily adjust the frequency of the kernel clock. In the emulation it was installed on a PC with two Ethernet interfaces, and was used for rate limitation and analysis of packets inside the IPsec tunnel. FreeBSD built-in bridging capabilities were also used, allowing traffic to flow through the PC at Layer 2.

A higher kernel timer frequency allows increased resolution of all operations of Dummynet, i.e. it allows Dummynet to more precisely limit the bandwidth. In the emulation, the frequency was set to 40000 HZ, in accordance with suggestions in [54]. This is easily done by adding

```
kern.hz = 40000
```

to */boot/loader.conf*, and then reboot the computer.

Dummynet is an application which allows the user to shape traffic, impose delay, loss, bandwidth etc. It is used in conjunction with the firewall *ipfw* and both are included in the FreeBSD operating system. In fact, Dummynet only has a small manual itself, the complete manual can be found in the Traffic Shaper section of the *ipfw* manual. *ipfw* inspects and identify packets/frames, and based on the given rules, forwards them to Dummynet which shapes the traffic before they leave the bridge.

---

<sup>3</sup><http://www.freebsd.org>



The following is taken from the *ipfw* manual:

```

      ^      to upper layers      V
      |                                |
      +----->-----+
      ^                                V
[ip(6)_input]          [ip(6)_output]    net.inet(6).ip(6).fw.enable=1
      |                                |
      ^                                V
[ether_demux]         [ether_output_frame] net.link.ether.ipfw=1
      |                                |
      +--->---[bdg_forward]--->---+    net.link.bridge.ipfw=1
      ^                                V
      |      to devices              |
```

It shows at which point *ipfw* inspects and apply action to the packets. In the emulation this is done at `bdg_forward`, since `net.link.bridge.ipfw` is set to 1 (see next section). Consequently, the traffic shaping is done before any processing is done at Layer 2 or above. This is an important fact when calculating maximum goodput.

### 5.3.1 Configuration - Bandwidth limitation

To enable Layer 2 (Ethernet) bridging on FreeBSD, the following commands are necessary. Initially, the interfaces are set to 10 Mbps/Full duplex, then the bridge itself is created and its member interfaces assigned (make sure they are up):

```
ifconfig sis0 media 10baseT/UTP mediaopt full-duplex
ifconfig sk0 media 10baseT/UTP mediaopt full-duplex
ifconfig bridge create
ifconfig bridge0 addm sk0 addm sis0 up
```

Limiting the bandwidth of the traffic crossing the bridge requires the configuration of *ipfw* and *Dummynet*. *Dummynet* and *ipfw* is included in FreeBSD as modules, which must be loaded before one is able to utilize them. The following commands loads the *ipfw* and *Dummynet* modules, and enable *ipfw* to inspect packet at Layer 2:

```
kldload ipfw.ko
kldload dummynet.ko
sysctl -w net.link.bridge.ipfw=1
```

The following commands create filters which match the traffic flowing between the peers - these are identical to firewall filters used on FreeBSD. However, the action to be taken if traffic is matched is not to drop or allow, but forward to pipe 1 or 2. Note that the actual IP payload and head is encrypted, making only the new IP header (appended by IPsec) visible for *ipfw*. Consequently the filters must match the IPs at which the IPsec tunnel terminates, ref. Figure 5.4.

```
ipfw add 9 pipe 2 all from 158.38.122.10 to 158.38.122.9
ipfw add 8 pipe 1 all from 158.38.122.9 to 158.38.122.10
```

Lastly, the pipes themselves are created. These are a part of Dummynet and enforce the rate limitation. The exemplars below limit the bandwidth to 500 kbps, as according to the design, and with a buffer at 100% of BDP.

```
ipfw pipe 1 config bw 500 Kbit/s queue 33KB noerror
ipfw pipe 2 config bw 500 Kbit/s queue 33KB noerror
```

## 5.4 TCP peers

The TCP peers are running Ubuntu 10.10 and Windows 7 Professional. Both have no installed software or configuration changes which may affect their networking performance.

On both operating systems, all interfaces were set to 10 Mbps and full duplex, according to the design. In addition, Large Segment Offloading was disabled. If enabled, the interface will signal a very large Maximum Segment Size to the application layer. These large segments will obviously be fragmented at the interface before being transmitted. However, Wireshark inspect these segments before they are fragmented, creating incorrect captures. In Windows these configurations are done via built-in tools, while for Linux *ethtool*<sup>4</sup> was utilized.

The built-in FTP client was utilized for both systems. On Linux, ProFTPD<sup>5</sup> FTP server was utilized, with default settings. On Windows, the built-in FTP server (must be enabled) was used, also with default settings.

On Linux, the TCP flavor was viewed and changed using the following commands:

```
sysctl net.ipv4.tcp_congestion_control
sysctl -w net.ipv4.tcp_congestion_control=cubic
```

Natively, only Cubic and Reno is available. To add, for example Hybla, one has to add the module. The following commands add the module, and list the available flavors:

---

<sup>4</sup>man ethtool

<sup>5</sup><http://www.proftpd.org>

```
modprobe tcp_hybla
sysctl net.ipv4.tcp_available_congestion_control
```

All modules available in kernel 2.6.35 are: BIC, HTCP, Illinois, Probe , Vegas, Westwood, Highspeed, Hybla, Lp, Scalable, Veno and Yeah.

As stated in the introduction of this thesis, the aim was (among others) to evaluate accessible and easily deployable solutions to the challenges discussed. The current Hybla implementation in Linux lacks two algorithms which the creators recommend, namely Packet Spacing and Hoe's channel bandwidth estimate [46]. Adding support for these algorithms would require exchange of the Linux kernel (new kernel with MultiTCP patch) on all peers, and is therefore not regarded as an easily deployed solution. Consequently, the Linux built-in implementation of Hybla is evaluated on the peers, without the additional algorithms.

## 5.5 PEPsal and MultiTCP

PEPsal is described in [31] and in Section 4.3.2, while MultiTCP is described in [55]. Since both are open source, they can be downloaded at <http://sourceforge.net/projects/pepsal/> and <http://sourceforge.net/projects/multitcp/>, respectively.

While PEPsal is an open source implementation of a Performance Enhancing Proxy (PEP), MultiTCP is a Linux patch for experimental evaluation of TCP enhancements. The patch provides several TCP flavors and mechanism like Packet Spacing and Hoe's channel bandwidth estimate [46]. In addition it can output information on a variety of TCP variables and parameters throughout a transmission. The interest in the patch is due to the *complete* implementation of TCP Hybla. Although Hybla is included in Linux, it does not incorporate two algorithms which are recommended by the creators, namely Hoe's channel bandwidth estimate Set-Up and Packet Spacing. As mentioned, this is acceptable at the peers, but should be included at the PEP since this requires minimal effort to deploy.

However, I have not been able to produce a working Linux image patched with MultiTCP. Large efforts have been made to accomplish this. Numerous images were created utilizing several different computers, kernel versions, compiler directives and compilers. But sadly, a bootable image has not been possible to produce. However, there is no doubt that the MultiTCP patch actually can work, and the problem lies in aforementioned efforts - not in the patch itself. Hybla in PEPsal is therefore utilized without the recommended algorithms, and performance may be improved using the complete implementation.

### 5.5.1 Configuration - PEPsal

The emulation has been set up according to Figure 5.3. After being compiled (*make*) and installed (*make install*), some firewall rules (in Linux: *iptables*) must be added. An example script is provided by the creators, and this has been used, with a few modifications. The modified script can be seen in Appendix C. Before running the script, the chains which the script populate and utilize must be created:

```
iptables -N TCP_OPTIMIZATION -t nat
iptables -N TCP_OPTIMIZATION -t mangle
```

To start PEPsal, the following command, with parameters according to script is executed:

```
pepsal -q 9 -v -p 6009
```

## 5.6 Tools

This section will describe the different tools used, tested, and considered in conjunction with the emulation. In addition, simple explanations on how to use, and specific usage in the emulation are included. Where appropriate, experiences and challenges met during testing will be described.

### 5.6.1 Wireshark

Wireshark is an open-source packet capture and analyzer software. Capturing is done using the *pcap* API for Linux, and the WinPcap for Windows (installs with Wireshark installer). Wireshark is openly available for download on the official web site: <http://www.wireshark.org/>, and via the APT package handling utility for Ubuntu/Debian. Wireshark will capture and log all frames leaving or entering a selected interface. In addition, it provides a variety of tools to analyze the captured frames, e.g. RTT, throughput, TCP flow graph, etc. Also, it will inspect content in TCP segments (among other) and automatically recognize and mark duplicate ACKs, retransmissions, resets, and similar, which simplifies analysis. Because of this, Wireshark is the preferred tool throughout the emulation to for most measurements.

It is outside the scope of this section to explain all uses of Wireshark, but the use is mostly tied to the "IO graphs" found in the "Statistics" menu. Through this tool a major part of the results has been generated. It should be noted that the tool offers few ways to combine graphs or customize their graphical appearance, so the results are put into gnuplot or Microsoft Excel.

## 5.6.2 tcpdump

*tcpdump* is a command-line packet capturer and analyzer. It was used when analyzing packets, especially Layer 2 (ethernet) information on size and bytes on the wire, as they traverse the IPsec tunnel at the satellite emulator (FreeBSD PC with DummyNet). This was done to calculate overhead and maximum throughput through the IPsec tunnel. It is natively included in FreeBSD and usage was done with:

```
man tcpdump
tcpdump -v
```

Which displays (on the command-line) Layer 2 information on each packet traversing the bridge.

## 5.6.3 TCP Probe

TCP probe is a Linux module which allows "real-time" values of, amongst other, *cwnd* and *ssthresh* be recorded during a TCP session. A more detailed explanation may be found on The Linux Foundation TCP Probe website: "It works by inserting a hook into the `tcp_recv` processing path using `kprobe` so that the congestion window and sequence number can be captured." [56]. TCP probe gives an unique opportunity to study the behavior of different TCP congestion algorithms, e.g. Figure 4.8, and may allow a further understanding of phenomena seen in the throughput. Regrettably, no Microsoft Windows counterpart has been found, and it is fair to assume none exist due to the restrictive nature of the Windows kernel.

TCP Probe is very simple to use, examples may be found on the TCP Probe website. Some scripts can also be found on the TCP Testing website ([57]), but were not used for the emulation. The module is included in Ubuntu 10.10, and to load it for FTP (port 20) type:

```
modprobe tcp_probe port=20
```

To capture the output from TCP probe into file (TCPreno.out):

```
cat /proc/net/tcpprobe >TCPreno.out & pid=$!
```

For each segment sent, TCPreno.out will now contain a line with 10 values, including the current *cwnd* and *ssthresh*. The website mentions and explain only 9 fields, leaving the 10th unknown.

To stop the capture:

```
kill $pid
```

It has been experienced during testing that TCP probe writes to file in chunks of around 32 kB, i.e. if it has not captured enough to fill 32 kB, it will not show up in output. Consequently, a

transfer can not contain fewer segments than TCP probe needs to generate 32 kB, or else no output will be seen at all.

In addition, some instability was experienced. At random intervals, TCP Probe will not output anything at all - no solutions or explanations have been found.

An alternative was considered - the MultiTCP patch described in section 5.5. But TCP probe was chosen due to its simplicity and ease of use.

### 5.6.4 Gnuplot

Gnuplot<sup>6</sup> is a command-line drive graphing utility. It supports multiple operating systems, but has been tested in Linux only. On Ubuntu, it installs easily via the APT package handling utility, and the command:

```
apt-get install gnuplot
```

In the emulation, Gnuplot has been used to draw graphs of data retrieved from both tcpprobe (mainly *cwnd* and *ssthresh*), and Wireshark. It was preferred due to its simplicity, and ability to directly generate Encapsulated PostScript (graphical file format, .eps) which is convenient if writing in L<sup>A</sup>T<sub>E</sub>X. No other software was considered for this use.

Scripts used to generate the *cwnd* and *ssthresh* graphs were slightly modified (colors and output) versions of scripts (tcpview and tcpprint) found on The Linux Foundation web site [57]. For all other, different scripts were written for each graph. There are too many to include in this paper, but a selected script can be found in Appendix B along with the tcpprint script.

### 5.6.5 tcptrace

*tcptrace* is a tool which analyses TCP dump files. It can easily be downloaded via the official web site: <http://www.tcptrace.org/>, or via the APT package handling utility for Ubuntu/Debian. It can analyze dump-files from several capture-programs, e.g. tcpdump or Wireshark. The latter has been used for this thesis. It is enormously powerful and versatile, and analyzes a wide variety of parameters, and provides flexible output. A quick peak of the capabilities can be seen by reading the help:

```
tcptrace -hargs
```

Or read the manual available at the web site. The usage has been mainly to provide RTT calculations of a transmission. This can be done using the following command:

---

<sup>6</sup><http://www.gnuplot.info/>

```
tcptrace -R -z WiresharkCapture.pcap
```

The `-R` option will generate RTT graphs, while `-z` will set the time (x-axis) to start at 0 instead of clock-time at capture. The output will be in form of XPLOT graphs. XPLOT is a simple graphing program, similar to gnuplot, and installs automatically when installing `tcptrace`. For conformity (and reduced workload, not having to learn a new program), a program called XPL2GPL was used to convert the XPLOT graphs to the gnuplot format. This produces a separate file for the actual data, called `WiresharkCapture.datasets`. The data in these datasets was then used to draw the gnuplot graphs using scripts mentioned in the gnuplot-section.

One important note should be made on the RTT calculations of `tcptrace`. It utilizes Karn's algorithm [58] for measurement. The algorithm states that a measurement taken on a segment which has been sent more than once, should be discarded. For some emulation results, this will cause gaps in the measured RTT as retransmission might be common, and will affect the calculation of average RTT.

### 5.6.6 Iperf and Jperf

Iperf is a TCP and UDP traffic generator. It also reports, both on the server and client side, various parameters like throughput, loss, jitter, etc. One may also influence the TCP and UDP sockets at each side, e.g. buffer size, window size, maximum segment size, and many more. It can easily be downloaded at their web site: <http://sourceforge.net/projects/iperf/> or via the APT package handling utility for Ubuntu/Debian. Natively, it is only supported for Linux, but there are several Windows compilation of the source code found online. In addition, one might use Jperf (also available on the Iperf web site), which is a Java-based graphical extension to Iperf. Obviously, this runs on any operating system, and offers a graphical interface to the other command-line driven Iperf. It is important to note that Jperf actually runs Iperf, and no extra functionality (except real-time drawing of graphs based on output from Iperf) is offered.

In the emulation, Iperf is used, amongst other, to test if a specific bandwidth is obtainable. This task may be accomplished with the following two commands, for client and server respectively:

```
iperf -c 192.168.0.30 -u -i 1 -l 1400B -t 30 -b 1000K
iperf -s -u -i 1
```

This will result in a 1 Mbps UDP stream from the client towards 192.168.0.30, with a segment size of 1400 bytes, lasting in 30 seconds, and giving a report to output each second. A complete explanation of all available parameters can be found by displaying the Iperf help:

```
iperf -h
```

One important phenomena has been experienced on all Windows compilations tested (Jperf 2.0.2 and Iperf 2.0.4). For example, over a 2 Mbps link with 528 ms RTT, a maximum throughput of 393 kbps on TCP traffic is seen. This can be avoided if one adjusts the "Window Size" parameter manually. Consequently, it seems that Window Size governs the maximum value of *cwnd* in Windows (as Receiver Window on the sender can not affect throughput). The default value can be calculated to be approx. 26 kB, i.e. the sender is not able to expand its *cwnd* beyond 26 kB. This can explain the arbitrary "roof" met at 393 kbps.

Another minor detail is the misspelling of the bandwidth configured which mislead the user to think he configures the bandwidth in bytes. It says kBytes/s and Mbytes/s, yet the transfer itself is done in kbits/s, Mbits/s.

## 5.7 Internet Protocol Security (IPsec)

As the emulation utilizes a IPsec VPN tunnel, a short introduction on IPsec will be given here, along with the implications on performance due to overhead.

IPsec is a protocol suite which provides authentication, encryption and data integrity checks for communication at the IP layer. It can be used to secure end-to-end between two hosts, between networks and host, or between two networks/routers as in our emulation. IPsec was first described as early as 1995 in RFC-1825 and RFC-1829, but today the suite is described in a series of RFCs, most notably in RFC 4301 (Architecture), 4302 (Authentication Header (AH)), 4303 (Encapsulating Security Header (ESP)) and 4306 (Internet Key Exchange (IKEv2)). The support of IPsec is mandatory in IPv6, but optional in IPv4. The architecture of the suite is modular, which allows different mechanisms be used to achieve each security goals. Vyattas implementation of IPsec can be studied in the Vyatta documentation "VPN Reference guide" [52]. The usage of Vyattas IPsec in the emulation is summarized in Table 5.2.

Security	Algorithm used in emulation	Protocol used in emulation
Confidentiality	256-bits AES <sup>7</sup>	ESP
Authentication	Pre-Shared Key and SHA <sup>8</sup> -1	ESP and IKE
Data Integrity	SHA-1	ESP
Key Management	Pre-Shared Key	IKE

Table 5.2: Overview of security mechanisms in emulation

As mentioned, IPsec resides on the network layer. It has two modes, tunnel and transport

---

<sup>7</sup>Advanced Encryption Standard

<sup>8</sup>Secure Hash Algorithm



mode, as depicted in Figure 5.5. In the emulation, transport mode is used, and this provides the strictest security. It is the Encapsulation Security Payload (ESP) [59] protocol which is used on the actual traffic of data. When IPsec is run in transport mode, ESP will encrypt the entire TCP segment, i.e. data and header, as well as the IP header. It will then append an own ESP header, a new IP header, an ESP trailer, and an optional ESP Authentication Data Field, as shown in Figure 5.5.

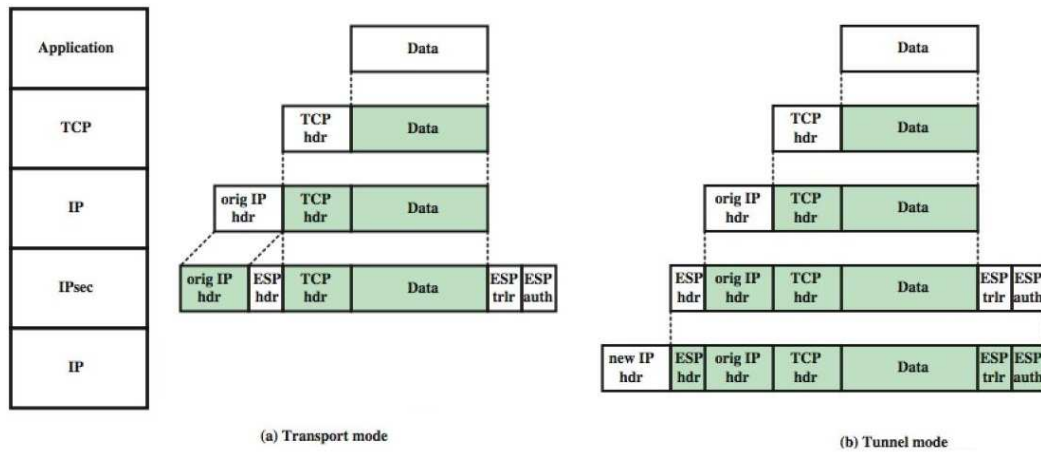


Figure 5.5: Protocol Operation for ESP [60]

These additional headers and trailers impacts the throughput of TCP and hence the results of the emulation. It is therefore important to describe and take into account this additional overhead to assure valid results.

Type	Bytes
New IP header	20
ESP header	8
Original IP header	20
TCP header	20
ESP trailer	Variable, $\geq 2$
$\Sigma$ IPsec headers	$\geq 30$
$\Sigma$ All headers	$\geq 70$

Table 5.3: Length of mandatory headers and trailers in ESP packet, tunnel mode

Table 5.3 shows the overhead accompanied with IPsec running in tunnel mode. In total, the packet will have at least 70 bytes of overhead, 30 of which is introduced by ESP. However, the ESP trailer has a variable size, due to the integrity check and the padding field. The padding

field can get as large as 255 bytes depending on the requirements of the encryption algorithm, the optional hiding of actual payload length, and the fact that the trailer must align with the right end of the header [60].

No. .	Packet size	Time	Source	Destination	Protocol	No. .	Packet size	Time	Source	Destination	Protocol
1	66	0.000000	192.168.0.30	10.0.0.3	TCP	1	134	0.000000	158.38.122.10	158.38.122.9	ESP
2	66	0.002084	10.0.0.3	192.168.0.30	TCP	2	134	0.000962	158.38.122.9	158.38.122.10	ESP
3	54	0.002165	192.168.0.30	10.0.0.3	TCP	3	118	0.002035	158.38.122.10	158.38.122.9	ESP
4	81	0.003977	10.0.0.3	192.168.0.30	FTP	4	150	0.002889	158.38.122.9	158.38.122.10	ESP
5	64	0.004058	192.168.0.30	10.0.0.3	FTP	5	134	0.003925	158.38.122.10	158.38.122.9	ESP
6	86	0.005499	10.0.0.3	192.168.0.30	FTP	6	150	0.004407	158.38.122.9	158.38.122.10	ESP
7	67	0.005577	192.168.0.30	10.0.0.3	FTP	7	134	0.005444	158.38.122.10	158.38.122.9	ESP
8	75	0.007587	10.0.0.3	192.168.0.30	FTP	8	134	0.006498	158.38.122.9	158.38.122.10	ESP
9	68	0.007683	192.168.0.30	10.0.0.3	FTP	9	134	0.007558	158.38.122.10	158.38.122.9	ESP
10	112	0.009142	10.0.0.3	192.168.0.30	FTP	10	182	0.008051	158.38.122.9	158.38.122.10	ESP
11	59	0.009218	192.168.0.30	10.0.0.3	FTP	11	118	0.009089	158.38.122.10	158.38.122.9	ESP
12	85	0.010702	10.0.0.3	192.168.0.30	FTP	12	150	0.009550	158.38.122.9	158.38.122.10	ESP
13	61	0.010777	192.168.0.30	10.0.0.3	FTP	13	134	0.010644	158.38.122.10	158.38.122.9	ESP
14	83	0.012400	10.0.0.3	192.168.0.30	FTP	14	150	0.011283	158.38.122.9	158.38.122.10	ESP
15	62	0.012525	192.168.0.30	10.0.0.3	FTP	15	134	0.012397	158.38.122.10	158.38.122.9	ESP
16	74	0.013956	10.0.0.3	192.168.0.30	FTP	16	134	0.012872	158.38.122.9	158.38.122.10	ESP
17	60	0.014168	192.168.0.30	10.0.0.3	FTP	17	118	0.014032	158.38.122.10	158.38.122.9	ESP
18	99	0.015843	10.0.0.3	192.168.0.30	FTP	18	166	0.014751	158.38.122.9	158.38.122.10	ESP
19	66	0.015992	192.168.0.30	10.0.0.3	TCP	19	134	0.015865	158.38.122.10	158.38.122.9	ESP
20	66	0.017510	10.0.0.3	192.168.0.30	TCP	20	134	0.016427	158.38.122.9	158.38.122.10	ESP
21	54	0.017557	192.168.0.30	10.0.0.3	TCP	21	118	0.017421	158.38.122.10	158.38.122.9	ESP
22	60	0.017577	192.168.0.30	10.0.0.3	FTP	22	118	0.017451	158.38.122.10	158.38.122.9	ESP
23	108	0.019440	10.0.0.3	192.168.0.30	FTP	23	166	0.018272	158.38.122.9	158.38.122.10	ESP
24	124	0.019443	10.0.0.3	192.168.0.30	FTP-DATA	24	182	0.018311	158.38.122.9	158.38.122.10	ESP
25	60	0.019445	10.0.0.3	192.168.0.30	TCP	25	118	0.018336	158.38.122.9	158.38.122.10	ESP
26	78	0.019447	10.0.0.3	192.168.0.30	FTP	26	150	0.018374	158.38.122.9	158.38.122.10	ESP
27	54	0.019520	192.168.0.30	10.0.0.3	TCP	27	118	0.019360	158.38.122.10	158.38.122.9	ESP
28	54	0.019555	192.168.0.30	10.0.0.3	TCP	28	118	0.019391	158.38.122.10	158.38.122.9	ESP
29	54	0.019567	192.168.0.30	10.0.0.3	TCP	29	118	0.019421	158.38.122.10	158.38.122.9	ESP
30	60	0.020819	10.0.0.3	192.168.0.30	TCP	30	118	0.019826	158.38.122.9	158.38.122.10	ESP

(a) FTP traffic captured at FTP client

(b) FTP/ESP traffic captured at intermediate router

Figure 5.6: Wireshark capture of TCP/FTP traffic

Figure 5.6 shows a Wireshark capture of actual traffic between a FTP client and a FTP server. Notice the first and second column which shows the packet number and size in bytes. Figure 5.6a shows the packets as they are leaving the interface of the FTP client and Figure 5.6b shows the packets at an intermediary router which is located at the black side of the network, between the client and server. Comparing the values of Packet Size between the two figures illustrates the added overhead of IPsec and ESP. As expected they are very variable, but never drop beneath 30 bytes. Evidentially, it is very hard to precisely determine the amount of extra overhead on a per flow basis. One might however calculate a upper bound of the possible throughput (there are so many optional and variable fields making a lower bound difficult (or pointless) to calculate). The actual bytes on the wire for a Ethernet frame is 1526 bytes<sup>10</sup> which with TCP through IPsec in tunnel mode will result in maximum 1430 bytes of payload. For a 2Mbps link, this yields:

$$TCPthroughput_{max} = \frac{2000000bps}{1526bytes} \times 1430bytes = 1.874Mbps \quad (5.2)$$

Consequently, the absolute maximum throughput of TCP traffic (or goodput) on a 2Mbps link is 1.876Mbps. As shown above, the overhead can be assumed to be even higher. Some further values can be found in Table 5.4. In addition, if smaller frames with smaller maximum payload

<sup>10</sup>Ethernet Frame consists of: 8 bytes of Preamble + 12 bytes of MAC addresses + 2 bytes of Type + 1500 bytes of payload (MTU) + 4 bytes of Integrity Check (FCS/CRC) = 1526 bytes on the wire.

than 1500 bytes are used, the relative amount of overhead (when compared to data) will increase, resulting in further reduction of TCP throughput.

<b>Bandwidth</b>	<b>Goodput</b>
500 kbps	469 kbps
1000 kbps	937 kbps
2000 kbps	1874 kbps

Table 5.4: Maximum goodput of TCP over different IPsec links.



## Chapter 6

# Results and Analysis

If not otherwise specified, the following setup has been used:

Bandwidth across satellite	500 kbps (each direction)
Bandwidth in terrestrial network	10 Mbps (each direction)
Round-Trip Time (RTT)	528 ms
Buffer before satellite links	100 % of BDP, i.e. 33 kB (each direction)
Loss	0 %
IPsec	Enabled, Tunnel mode

Table 6.1: Default parameters of emulation.

In the default setup, the receiver of a Linux or Windows TCP flow will be an identical host, i.e. Linux to Linux, Windows to Windows. Buffers at all other equipment has been left at default, yet there are no other bottlenecks besides the satellite, leaving the buffers (in practice) unused. Each section specifies the details of how measurements were conducted.

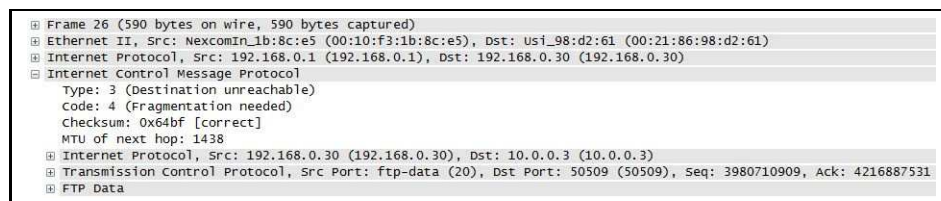
Buffer ahead of the rate limiter (Dummynet in Figure 5.2) will in certain runs of the emulation be adjusted between 85 % and up to 1000 % of BDP, i.e. between 28 kB and 330 kB. These larger values represent realistic buffer sizes, as described in [8], where several satellite modems and vendors are evaluated. The report mentions buffers of up to 400 kB, and buffer delay of up to 5.5 seconds. It appears that such large buffers are seldom utilized when TCP is evaluated in research, making the topic both relevant and interesting.

This thesis focuses on TCP flavors which are readily available and widely deployed, i.e. flavors available on Windows 7 and Linux. Based on that, NewReno (default in Windows 7), Compound TCP (available in Windows 7 and default on Windows Server 2008) and CUBIC (default in Linux

since kernel 2.6.19) has been chosen. In addition, TCP Hybla for Linux has also been tested since this protocol shows promising results ([30] and [29]), and is specially tailored for satellite environments. It should be noted that for TCP Hybla, Hoe's channel bandwidth estimate, and packet spacing, has not been enabled since this requires the MultiTCP patch and recompiling of Linux, as discussed in Section 5.5.

## 6.1 Emulation characteristics and confirmation

Each host utilizes the Path MTU (Maximum Transmission Unit) Discovery algorithm, starting a transmission with a IP packet where the "Don't Fragment" flag is set. Any router on the path which has a too low MTU, i.e. the packet must be fragmented, will respond with an ICMP<sup>1</sup> packet suggesting a lower MTU. Figure 6.1 shows the response from one of the routers in the emulation (D103, see Figure 5.4). Because of the IPsec tunnel, the router advertises a MTU of 1438, which causes the senders to utilize IP packets of max 1438 bytes (including header). Consequently, the maximum TCP payload available is 1398 bytes, assuming 20 bytes of TCP header, i.e. no option fields are utilized.



```

# Frame 26 (590 bytes on wire, 590 bytes captured)
# Ethernet II, Src: NexcomIn_1b:8c:e5 (00:10:f3:1b:8c:e5), Dst: Us1_98:d2:61 (00:21:86:98:d2:61)
# Internet Protocol, Src: 192.168.0.1 (192.168.0.1), Dst: 192.168.0.30 (192.168.0.30)
# Internet Control Message Protocol
  Type: 3 (Destination unreachable)
  Code: 4 (Fragmentation needed)
  Checksum: 0x64bf [correct]
  MTU of next hop: 1438
# Internet Protocol, Src: 192.168.0.30 (192.168.0.30), Dst: 10.0.0.3 (10.0.0.3)
# Transmission Control Protocol, Src Port: ftp-data (20), Dst Port: 50509 (50509), Seq: 3980710909, Ack: 4216887531
# FTP Data

```

Figure 6.1: Wireshark capture showing ICMP answer from router to TCP peers Path MTU Discovery algorithm.

*tcpdump* on the FreeBSD satellite emulator, shows Layer 2 (Ethernet) information on the traffic as it is "inside" the IPsec tunnel. It reports the largest frame to be 1510 bytes, and 156 bytes for ACKs. This does not take account the 8 bytes of preamble and 4 bytes of integrity check which has been stripped of the ethernet frame (presumably by the interface hardware). Total bytes traversing the wire is therefore 1522.

From Figure 6.1 we see 1438 bytes of maximum IP packet size at the TCP peer. Adding 26 bytes of Ethernet headers and trailers gives a total of 1464 bytes on the wire from the TCP sender. The difference between bytes on the wire outside compared to inside the tunnel is thus 58 bytes, which is the added overhead by IPsec. Compared to Table 5.3 (30 bytes overhead), this seems reasonable. The extra overhead can be explained by the 20 bytes of SHA-1 integrity check, up to

---

<sup>1</sup>Internet Control Message Protocol

15 bytes of encryption block padding (AES uses a 16 bytes block size), and possibly alignment padding which ensures the fields ends at the correct place in the header.

To calculate the maximum possible throughput across the emulator, one can use Equation 5.2. As mentioned in Section 5.3.1, Dummynet will measure throughput based on the Ethernet frame, excluding the preamble and integrity check. 1510 bytes on the wire (perceived by Dummynet) and 1398 bytes of TCP payload (no options utilized) yields:

$$TCPthroughput_{max} = \frac{500000bps}{1510bytes} \times 1398bytes = 462.914kbps \quad (6.1)$$

And with UDP (8 byte header) instead of TCP:

$$UDPthroughput_{max} = \frac{500000bps}{1510bytes} \times 1410bytes = 466.887kbps \quad (6.2)$$

Initial tests to confirm the bandwidth was done using Iperf. A 1 Mbps stream of UDP was sent across the emulator for 30 second. Measurements at the receiver showed a total UDP throughput of 466.791 kBps, confirming the capabilities and performance of the emulator. Note that UDP did not utilize the MTU path discovery, thus sending packets with 1500 byte size. This causes fragmentation at the Vyatta routers when IPsec headers and trailers where to be appended, and after a few seconds all traffic stopped - it is believed to be caused by drainage of fragmentation buffer at the receiver side (the receiver stores fragments waiting for the remaining fragments). Forcing Iperf to utilize a 1438 bytes MTU solved the problem.

### 6.1.1 On the performance of CTCP

Throughout the emulation, CTCP seems to have a very similar performance when compared to TCP NewReno. This is understandable based on the description in Section 4.2.3. Because CTCPs behavior is heavily affected by tunable parameters which may vary between implementations, hypothesis and predictions of behavior has been very difficult. Based on Figure 6.9, it seems the multiplicative decrease  $\beta$  (refer Section 4.2.3) is quite similar to 0.5 found in NewReno, and/or  $\alpha$  and  $k$  (which governs the increase of  $wnd$  ( $cwnd$ )) has been set conservatively.

Another important parameter is the  $W_{low}$  which is the threshold of how large the  $cwnd$  must be before the delay-based component is enabled. It is possible that the BDP of the emulation is not large enough to trigger the use of the component. The BDP of the link is  $512 \text{ kbps} \times 512 \text{ ms}$ , totaling to 33 kB, which is about 22 segments. With a 100% of BDP buffer, which is the smallest used in the emulation: 66 kB, approx. 44 segments. Or with a buffer at 1000% of BDP, the total BDP would be 363 kB, approx. 242 segments. The  $W_{low}$  used by the creators in [45]

was 41 segments. If Microsoft uses the same limit, the delay-based component would probably not be used for 100% buffer.

As described in Section 4.2.3,  $\gamma$  governs how many packets can be buffered before CTCP regards it as a congestion. If Microsoft uses the static value proposed in [44], this limit would be 30 packets. I.e. 30 packets must be residing in the buffer before CTCP regards the link as congested. However, the authors of CTCP also suggested a dynamic scheme to govern  $\gamma$  [45]. The dynamic algorithm is seen below:

$$\gamma = \max(\gamma_{min}, W_{low} \times \frac{\kappa}{1 + \kappa}) \quad (6.3)$$

$$\kappa = \frac{R_{max} - R_{min}}{R_{min}} \quad (6.4)$$

Assuming Microsoft uses the proposed values in [45], with  $W_{low}$  at 41 packets and  $\gamma_{min}$  at 3 packet. This would result in  $\gamma$  at approximately 20 packets for 100% buffer, and approximately 37 segments for 1000% buffer. The buffers can contain around, respectively, 22 and 242 segments. The consequences, for 100% buffer, would be that the delay-based component never could detect a congestion.

A comparison of Figure 6.2 and the results observed in Figure 6.18 suggest the assumptions above to be correct. An anticipated, linear, NewReno-like curve is seen at 100 % buffer. And a faster inclining, CTCP curve is seen with 1000 % buffer.

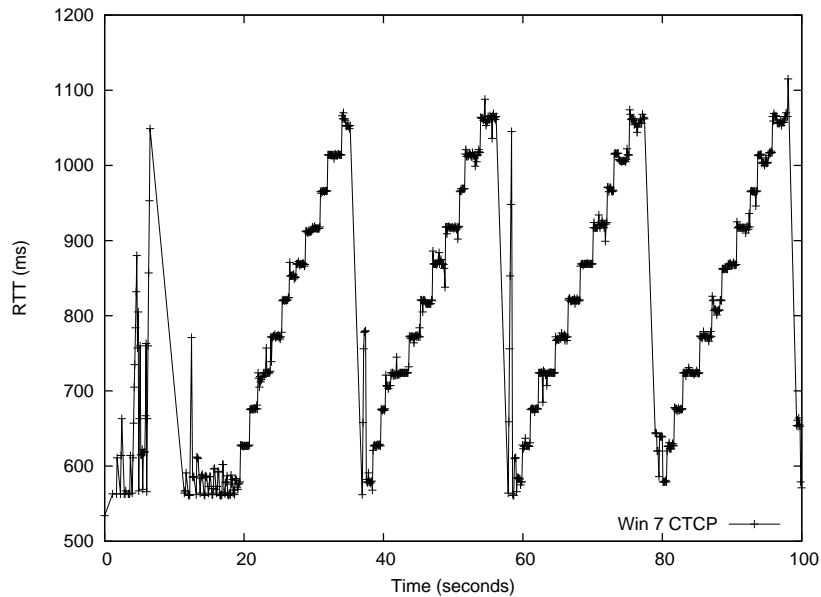


Figure 6.2: RTT of CTCP flow with 100 % buffer.



## 6.2 Results and analysis

### 6.2.1 Impact of UDP

Figure 6.3 shows the impact of a simultaneous UDP flow on different TCP flavors. Since UDP has no congestion avoidance, back-off mechanism, or similar, the sender will always submit traffic to the network at its desired speed. The effect is seen as an effective bandwidth limiter where TCP acts as if the available bandwidth equals actual bandwidth minus UDP traffic. The maximum goodput is around 463 Kbps (Equation 6.1).

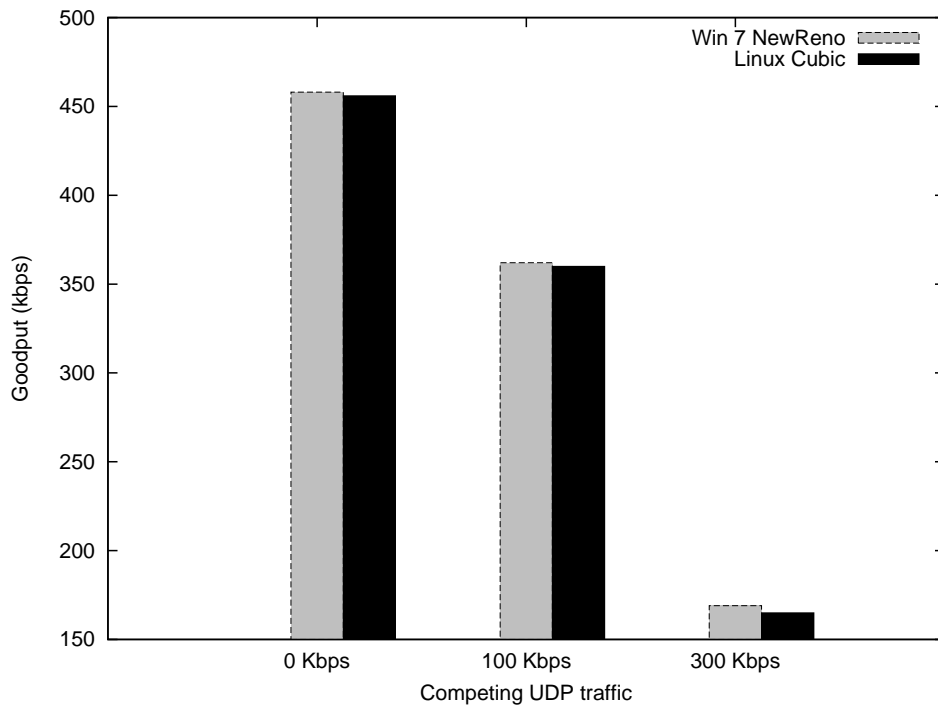


Figure 6.3: Goodput of single TCP flow with and without competing UDP flow.

### 6.2.2 Friendliness

Friendliness of a TCP protocol is defined as "its capacity to assure a fair band subdivision among competing connections that use different protocol variants." [29]. Fairness considers the same property, but among protocols of the same version. The measurements have been done as follows: First, one (or several where applicable) traffic flow is started. The stream is allowed to reach steady-state, i.e. the sender has exited Slow Start and had several opportunities to expand the *cwnd* to its maximum. At this point the competing flow is started, and is equally allowed to reach its steady-state. Around three minutes into the test, the actual measurement is taken for a minimum of three minutes and up to six minutes depending on the buffer size. This procedure is repeated twice to mitigate any random effects.

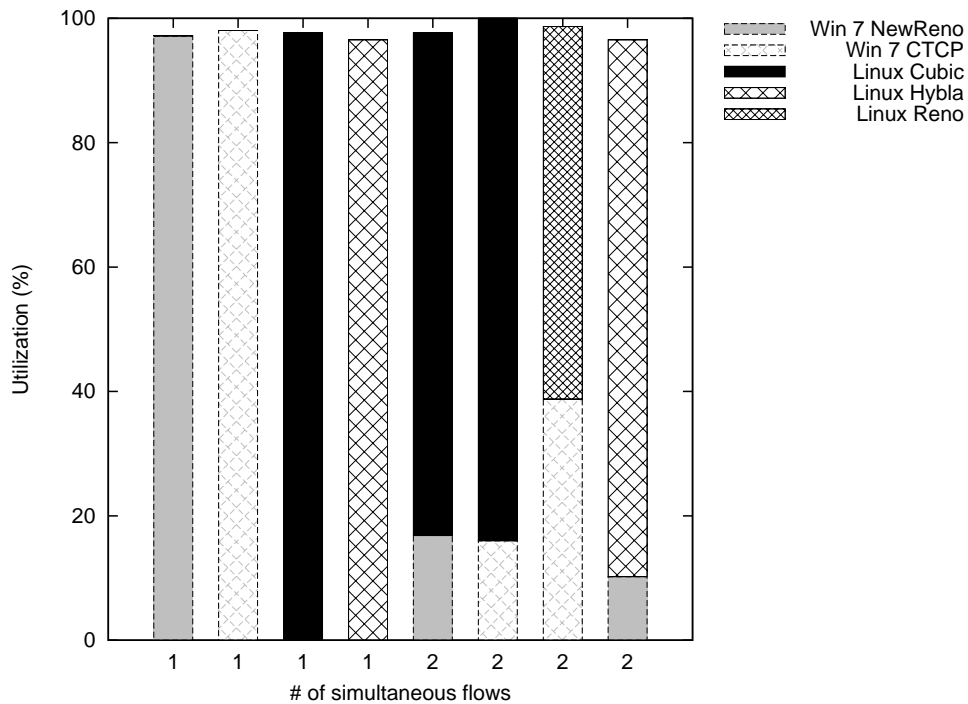


Figure 6.4: Link utilization of single TCP flows (marked "1"), and friendliness between two competing (marked "2") TCP flows of different flavor.

Figure 6.4 shows the friendliness of all aforementioned TCP flavors with buffer at 100 % of BDP. First four columns show the performance of each flavor without any competing flow. Not surprisingly, they are able utilize close to 100 % of the link. The four last columns show friendliness between two competing flows. It is evident that NewReno and CTCP are heavily penalized by the more aggressive counterparts, CUBIC and Hybla. The behavior seen is due to Reno and CTCPs liberal back-off at packet loss, and conservative increase of *cwnd* which

depends on the RTT. As suspected, CTCP shows a performance almost identical to NewReno as discussed in Section 6.1.1. The unfriendliness of CUBIC and Hybla is seen to be devastating for other flows. Implicitly this gives the competing flows priority over Windows 7 flows across the network. Competitors are able to utilize less than 20 % of the available bandwidth. The last column shows the Hybla flow using around 90 % of the link, at the cost of NewReno.

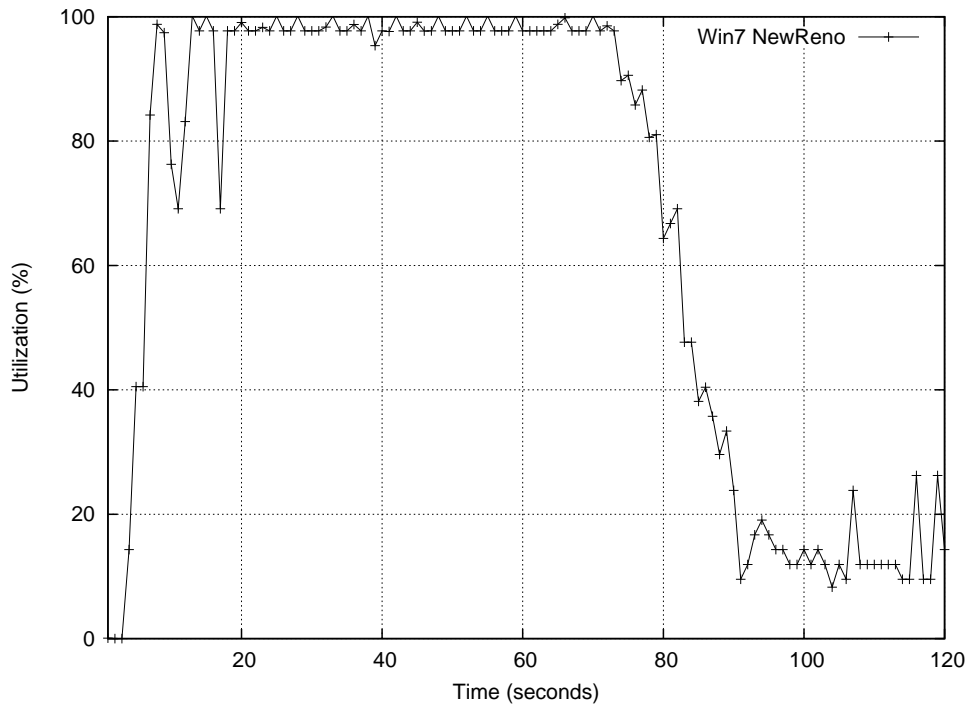


Figure 6.5: Impact of competing CUBIC flow (introduced at approx. 70sec.) on Windows 7 NewReno flow.

A measurement of the throughput as a function of time can be seen in Figure 6.5. This shows the throughput of a NewReno flow throughout the first 120 seconds of the test - note that measurements are not initiated until 180 seconds. Most notably the figure shows the transient phase as the competing CUBIC flow is introduced at approximately 70 seconds. The impact of CUBIC initiates a fairly rapid decline in throughput. Within 10 seconds, or 20 times the RTT, the utilization is reduced to around half of its initial value. And after 20 seconds, NewReno accounts for less than 20 % of the link utilization.

Figure 6.6 shows multiple Reno flows, competing against one CUBIC flow. As more and more Reno flows are added, the share of bandwidth which the Reno flows receive should follow the "Fair share" dotted line. However, it is evident that due to the poor friendliness of CUBIC, there is a gap between actual and fair share. The gap decreases as the number of Reno flow increases, making CUBIC more friendly the more "competition" it meets. In a real traffic scenario with

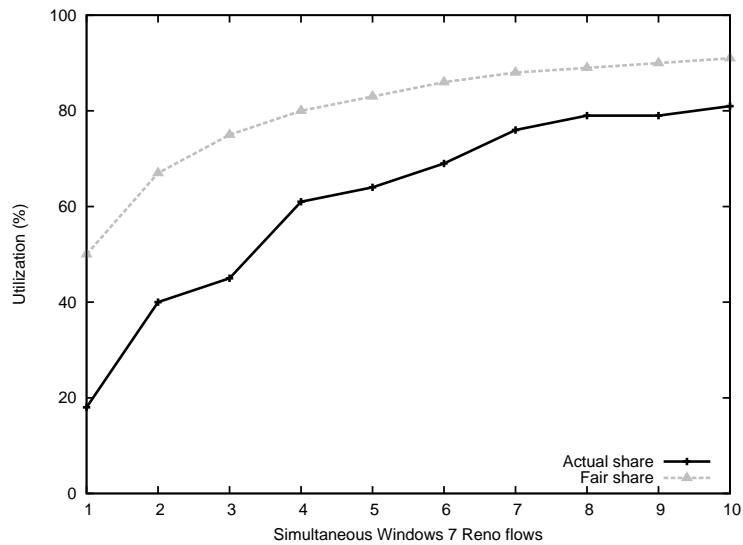


Figure 6.6: Multiple Windows 7 Reno flows vs. one Linux CUBIC flow. All in steady state.

potentially hundreds of flows, CUBIC may prove not be as unfriendly as seen in Figure 6.4 and 6.6

As the buffer size increases, the friendliness-problem becomes emphasized. This is depicted in Figure 6.7. The buffer allows each flow to potentially increase its *cwnd* and throughput above the 500 kbps bandwidth limit. Since CUBIC opens its *cwnd* more aggressively following a loss, and back-off less than NewReno, it will exploit this at a higher pace than NewReno. Consequently the buffer will emphasize the difference between the two flows, and worsen the unfriendliness of CUBIC. For very large buffers, the figure shows an almost devastating performance of NewReno, utilizing about 5 % of the available bandwidth.

Finally, a PEP has been placed in front of the satellite link, following the design in Figure 5.3. The measurements themselves are conducted in a similar fashion as before - for large buffers, the measurement period is increased to six minutes. The flows are terminated at the PEPsal PEP, where new Hybla sessions are initiated towards the intended receiver of the original flow. Figure 6.8 shows the friendliness of the flows with PEPsal enabled. Compared to 6.4, an improvement of around 20 % can be seen in the utilization of NewReno for 100 % buffer. Comparing the second column with Figure 6.7 also shows an increased utilization of about 20 %. Consequently it may seem that the use of a PEP may correct the unfair division of bandwidth.

However, the throughput measured at each receiver fluctuated with long periods during measurements. I.e. the throughput could be measured at near full utilization for a period, before falling to almost zero afterwards, and then repeated. The variance of the results in Figure 6.8 is therefore very large - questioning their validity. The cause of these characteristics are hard to

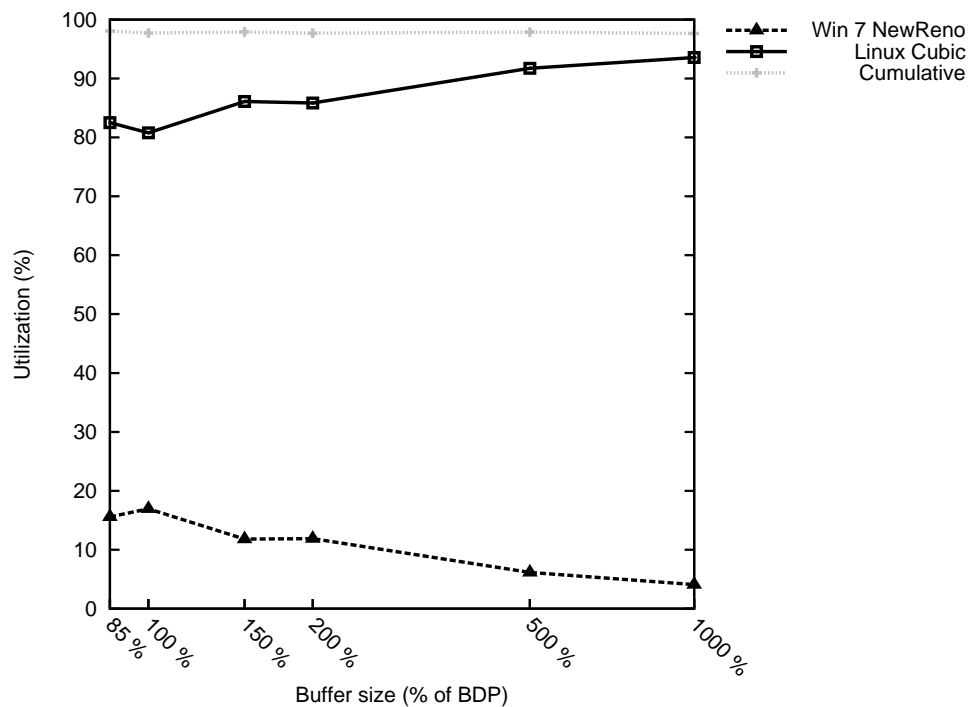


Figure 6.7: Friendliness between CUBIC and Reno for different buffer sizes.

pinpoint, but may be caused by bugs in the PEPsal implementation.

### 6.2.2.1 Implications

The TCP flavors available to Windows 7 (NewReno and CTCP) are harshly penalized when used simultaneously with a flow originating from a Linux (CUBIC) sender. As the flows must compete for the scarce bandwidth at the satellite bottleneck, the Linux sender will aggressively undermine the performance of a Windows 7 flow. Consequently, users would experience an unfair division of throughput in an environment with a mix of both operating systems at the sender side. Senders at a third-party portion of the network (e.g. Internet) can obviously not be dictated. However, in own networks, a mixed environment should be avoided, or other mechanisms e.g. QoS-schemes should be deployed to be address the unfairness. PEPs have shown to be a likely contributor to even out the share, although the performance of PEPsal is to varying to draw any conclusions.

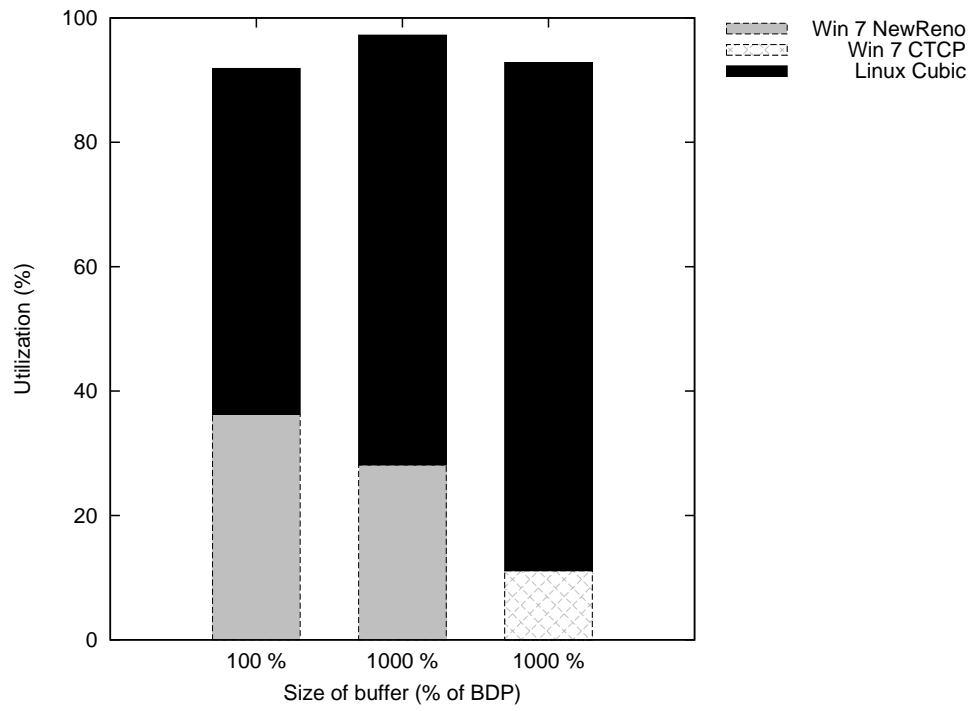


Figure 6.8: Friendliness between different flavors of TCP, all going through PEPsal utilizing Hybla.

### 6.2.3 Bit Error Rate (BER)

As mentioned in Section 2.3.2, satellite links are generally more prone to bit errors and packet losses than wired links. In addition, the emulation incorporates a radio link between the user and the satellite gateway. Since this is a likely scenario, it is interesting to look at TCP performance across lossy links. Measurements are done in a similar fashion as in previous sections; measurement is done over 5 minutes. Figure 6.9 shows each TCP flavors ability to utilize the link for different values of BER. Note that there is no competition; each flow can utilize the entire bandwidth.

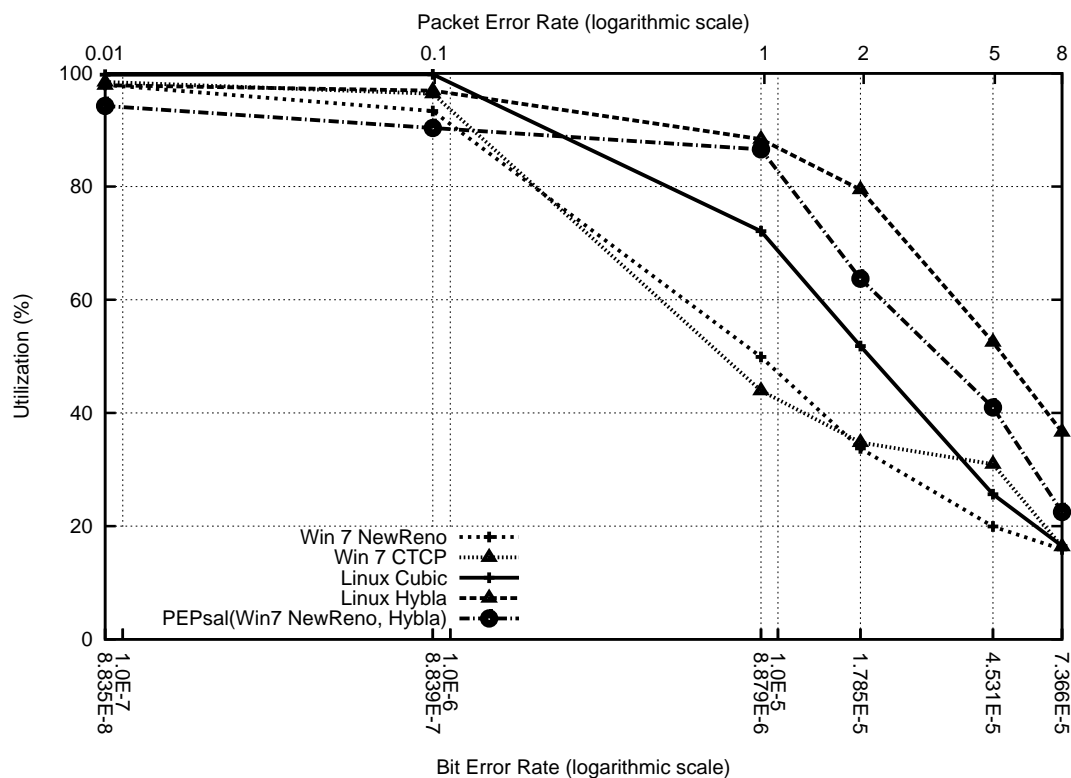


Figure 6.9: Impact of Bit Errors on single TCP flows.

It is evident that Hybla outperform the other flavors as the BER increases. Although CUBIC performs better at low PER values, Hybla achieve around 15 % gain for higher BERs compared to CUBIC. Hybla also outperforms the other flavors when BER reaches 7.3E-5 - more than 15 % higher utilization when compared to all other flows. CUBIC also performs well, but its performance degrades rapidly as the BER becomes high, culminating at around 20 % alongside the other flavors.

Both flavors available in Windows 7 are harshly penalized and performs around 20 % weaker

than CUBIC for moderate BER levels. Comparing the Windows variants with Hybla shows up to 40 % differences in utilization. In addition, NewReno is penalized at lower BERs and achieves close to 90 % utilization for BER at  $8.8E-7$  - slightly more than PEPsal.

PEPsal running Hybla also achieves Hybla-like performances, with around 10-15 % decrease when compared. Note that the PEPsal is fed by a NewReno flow, i.e. PEPsal improves the performance of NewReno with up to 40 %. However, for lower BERs, PEPsal is outperformed by all other flavors, including NewReno. In a scenario where low BERs are common and higher values occurs more seldom, this might prove a great disadvantage.

### 6.2.3.1 Implications

It is evident that the performance in lossy networks varies greatly between TCP flavors. At BER higher than  $1E-6$ , users will experience uneven performance in a mixed-TCP/OS environment. If a Linux transmitter should be able to utilize up to 80 % of the link, a BER between  $1E-6$  and  $1E-5$  is needed. If senders utilize Hybla, the same performance can be reached for BER as high as  $1.8E-5$ . To achieve a similar utilization at above 80 %, a Windows sender may not operate with BERs higher than  $1E-6$ . Such levels of BER are assumed to be easily obtainable across a satellite link, with some exceptions e.g. jamming. However, a hybrid network containing a radio leg would find this BER more challenging to achieve. A PEP may improve the performance of Windows and CUBIC transmitters significantly, at the cost of throughput at lower BERs. To summarize, the BER should not increase above  $1E-6$  if  $\geq 80$  % utilization is needed. This will accommodate the needs for "all" flavors on both operating system.



### 6.2.4 RTT and buffer

Ahead of the satellite link, in each direction, a buffer exists to accommodate overflowing traffic trying to enter the satellite modem/bottleneck. The size of this buffer can effect the performance of TCP. An immediate effect is seen in the Round-Trip Time, since it goes up as packets spend time in the buffer. Figure 6.10 tries to visualize this effect by showing the average RTT for different flavors and buffer sizes. However, since the measurements are done using TCPtrace and Karn's algorithm, large portions of the actual RTT may not be measured and included. Therefore the figure should only be viewed as an indicator, not a precise representation. Note that the buffer size is specified as percentage of Bandwidth-Delay Product (BDP). The BDP of the link is  $512 \text{ kbps} \times 512 \text{ ms}$ , totaling to 33 kB. A buffer size of 200 % would equal 66 kB, and 330 kB for 1000 %.

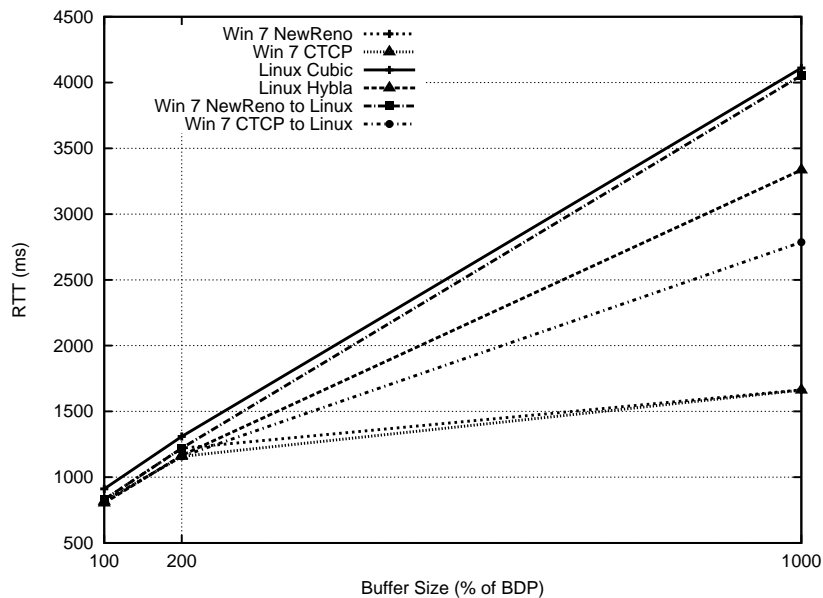


Figure 6.10: Average RTT for different flavors and buffer sizes.

#### 6.2.4.1 Windows 7 Advertised Window problem

Figure 6.10 also shows an interesting discovery made during emulation. Note the short RTT of both NewReno and CTCP, even for a buffer at 1000 % of BDP. An actual view of the situation over time can be seen in Figure 6.11 (NewReno) and 6.12 (CTCP), which shows the actual measured RTT during the transfers. Note that for 1000 % of BDP buffer size, the RTT is rapidly fluctuating between around 1 and 2 seconds. In order to explain this, it is important to note that RTT can also be viewed as a measure of buffer utilization: The longer the RTT, the

more data is in the buffer. As all flavors considered in this paper never keeps its sending rate constant, but tries to increase it, we would expect the RTT, i.e. buffer utilization, to go up to its maximum:

$$RTT_{max} = \frac{Buffer\ size}{Bandwidth} + Link\ delay = \frac{330\ kB}{500\ kbps} + 528\ ms = 5.8\ s \quad (6.5)$$

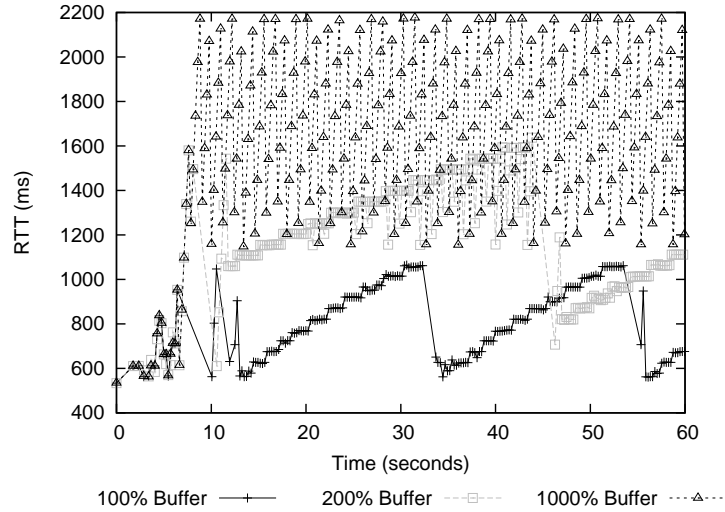


Figure 6.11: RTT of NewReno flow between two Windows 7 hosts.

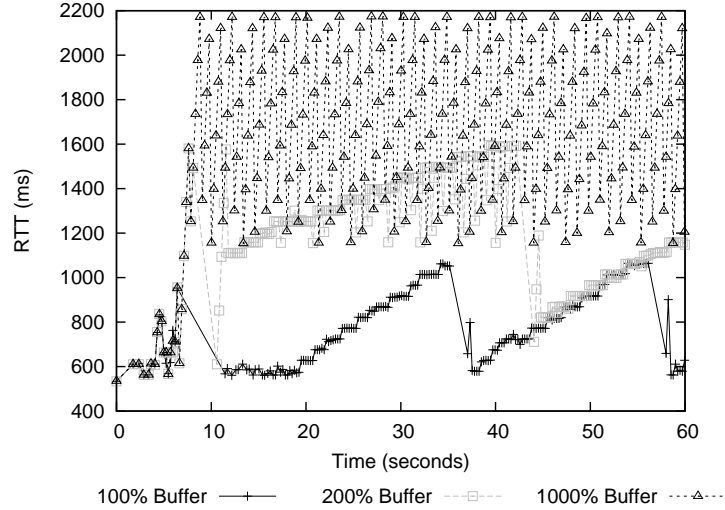


Figure 6.12: RTT of CTCP flow between two Windows 7 hosts.

An expected behavior would be for the transmitter to increase its *cwnd* until 100% of the buffer is utilized. At this point, a packet loss would occur and the transmitter would back off. During this period, the RTT (which can be seen as a measure of buffer utilization) should grow to around 6

seconds before dropping significantly as the sender backs off and the buffer drains. The behavior should repeat until the transfer is terminated. This expected behavior can be seen Figure 6.13 for CUBIC and Hybla. Note the peak at around 5.8 seconds, indicating a full buffer, and a packet loss - as expected.

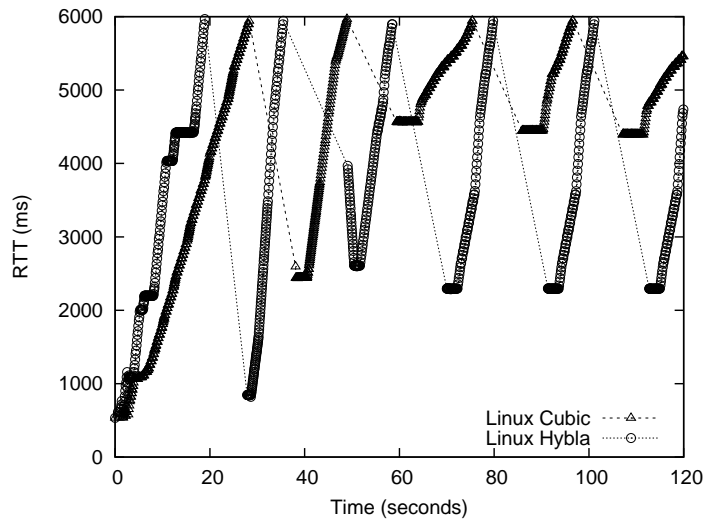


Figure 6.13: RTT of CUBIC and Hybla flows with 1000% BDP buffer.

The explanation to the strange RTT in both NewReno and CTCP can be seen when comparing Figures 6.14 and 6.15. The figures show the advertised receiver window (*rwnd*) signaled by the receiver towards the sender during the transfer.

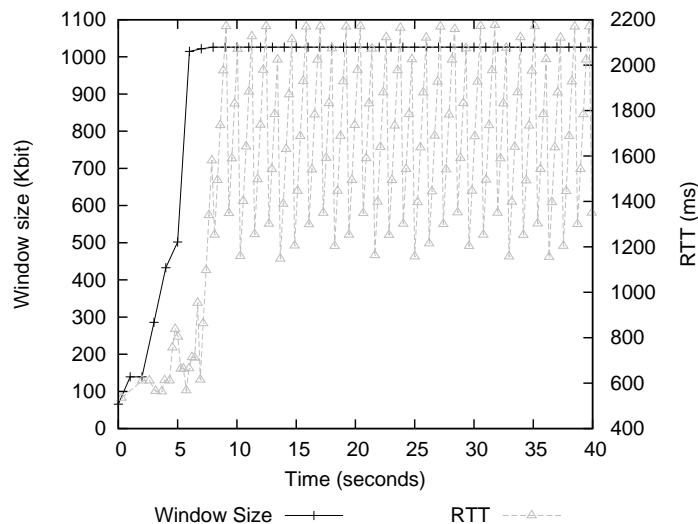


Figure 6.14: RTT and Receiver Window Size for NewReno flow between two Windows 7 hosts.

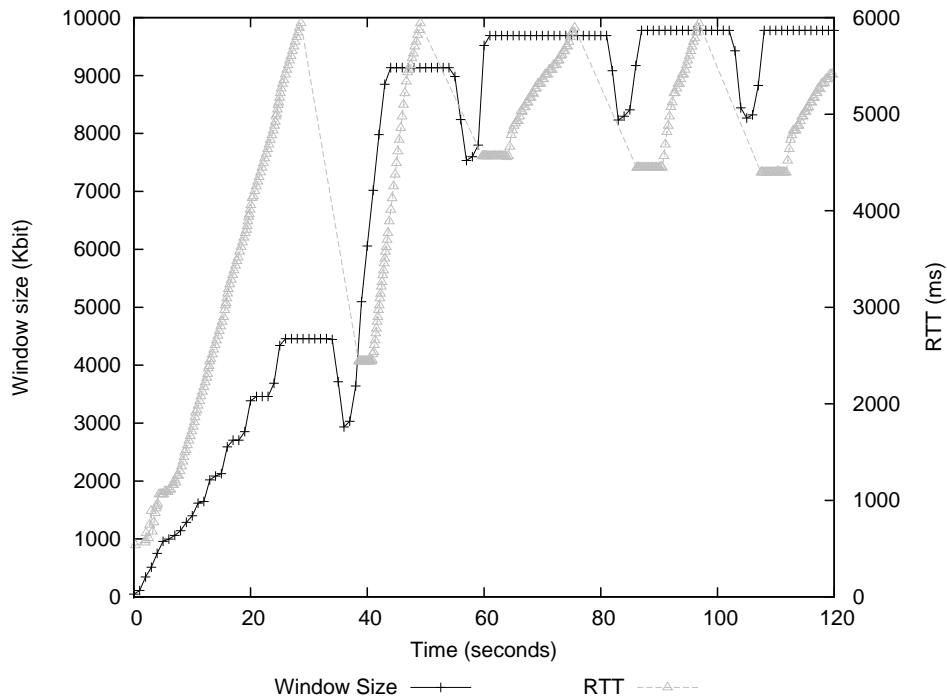


Figure 6.15: RTT and Receiver Window Size for CUBIC flow between two Linux hosts.

As described in Section 2.2.4, the *rwnd* should depict the buffer at the receiver, i.e. how much data he is willing to accept without acknowledgment. The actual transmission window, or allowed flight size is set to the minimum of the congestion window (*cwnd*) and *rwnd*. Typically, the *rwnd* is not the limiting factor since buffer at the receiver usually is not a scarce resource. Figure 6.14 shows the Windows 7 receiver signaling a maximum *rwnd* of around 1000 kbit - compared to Figure 6.15 where the Linux receiver signals a ten times larger window at about 10000 kbit.

Figure 6.16 compare *rwnd* for all buffer sizes. "Minimum to fill link" is calculated as the BDP + buffer size, i.e. all data which possibly can be in flight. "Windows 7 Experimental" seen in the figure is a mode of the Windows 7 Auto-tuning level - the mechanism which control the *rwnd* advertised by Windows 7. It can be adjusted using

```
netsh interface tcp global set autotuninglevel=experimental
```

at the command line. The help describes experimental mode as "Allow the receive window to grow to accomodate extreme scenarios.". This was tested to see if it is possible to impact the *rwnd*. As seen in Figure 6.16, it made no difference.

As Windows 7 does not signal a large enough window, the sender is not able to "fill the link", and thus have to wait for more ACKs as the window fill up, before he can transmit again. This causes the fluctuation in RTT and buffer utilization. Seemingly, Windows 7 is not able

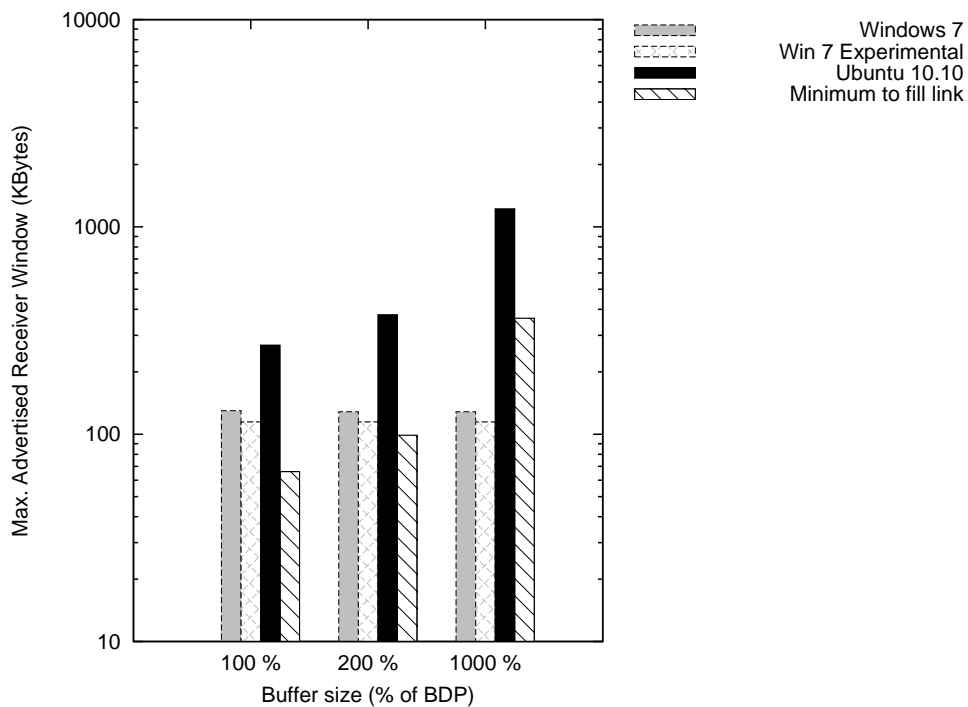


Figure 6.16: Maximum Advertised Receiver Window for different OS and buffer sizes.

to adjust its *rwnd* to accommodate large changes in delay after a transmission has begun. It is not clear how Windows 7 sets its *rwnd* but tests have shown that if the delay is present at the start of the transfer (i.e. the buffer is not empty), the *rwnd* will be increased sufficiently throughout the transfer. This effect can be seen in Figure 6.17, column marked '\*' compared to first column. The same behavior is experienced if delay is added during a transfer using the emulation/*netem*.

Figure 6.17 shows two competing flows, with different kind of receivers. The first column shows a CUBIC flow vs a NewReno flow, where the CUBIC flow has a Windows 7 receiver. This puts a constraint on CUBICs aggressiveness, since it will never be able to fill the buffer above the *rwnd*, around 125 kB (about one third of the buffer). Comparing the share to Figure 6.7 shows around 20 % drop in favor of NewReno. Second column shows the friendliness between two competing CUBIC flows; one is transmitting towards Windows 7, the other towards a Linux receiver. Normally, one would assume an almost equal share of the link. However, the constraints by the *rwnd* results in about 20 % penalty for the flow towards Windows 7. The unfair division in the third column is not caused by the *rwnd* (*cwnd* never reaches *rwnd*), but is rather the same unfriendliness of CUBIC as seen in Figure 6.7. The measurement in the final column is done with the Linux-receiver flow in steady-state, i.e. there is data in the buffer when the flow with the Windows 7 receiver starts. As discussed, this causes Windows 7 to actually increase it

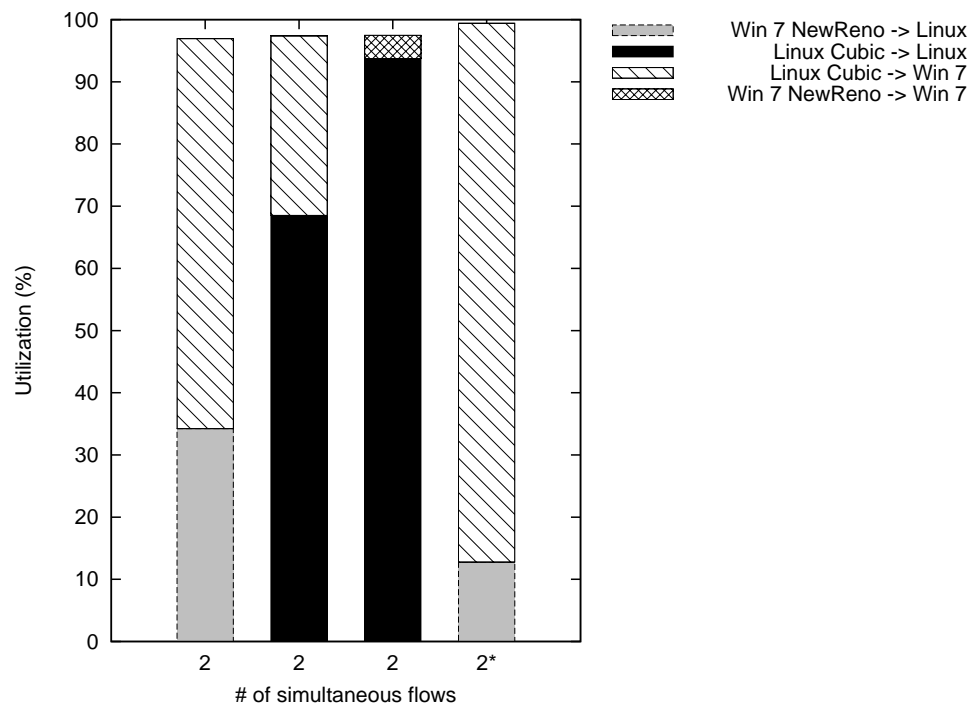


Figure 6.17: Friendliness between competing flows, with 1000% of BDP buffer. Flow with Windows 7 receiver in steady-state, except \*.

*rwnd* as anticipated. CUBIC will consequently not be penalized, and the division of bandwidth is identical to 6.7.

Figure 6.18 supports the explanation by depicting the RTT as it would have been without a restrictive *rwnd*. These new average values for RTT are included in Figure 6.10.

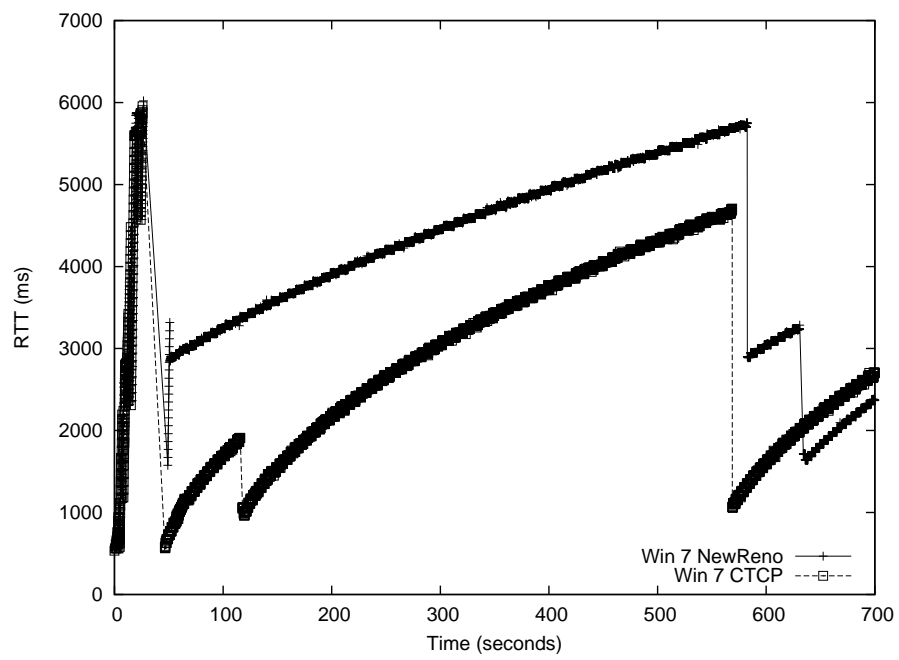


Figure 6.18: RTT of NewReno and CTCP flow from Windows 7 to Linux with 1000% BDP buffer.

### 6.2.4.2 Implications

A large buffer will in most situations result in a long RTT. The high RTT will not effect the immediate goodput, as seen in Figures 6.20 and 6.21. On the contrary, a sufficient large buffer can improve the goodput. As a buffer is completely filled up, one or several packets will inevitably be dropped. This causes the sender to back off, before again starting to increase the *cwnd*. If the buffer contains enough packets, the sender will increase its throughput up to the maximum bandwidth before the buffer is empty. Consequently, the buffer has "absorbed" the variation in throughput at the sender, and 100 % utilization is achieved. Figure 6.19 shows a situation where the buffer size is 100 % of BDP, and utilization is penalized. Note that these measurements are done with only a single flow traversing the link. As the number of flows increases, it is fair to assume that a smaller buffer is needed to mitigate this effect, since a flow backing off allows another flow to throttle up.

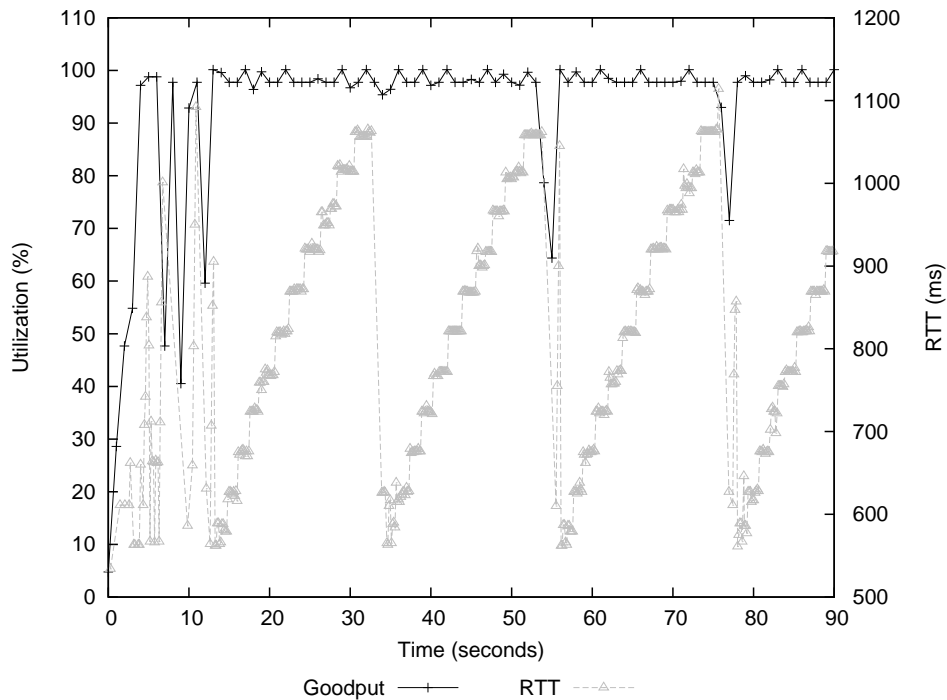


Figure 6.19: RTT and goodput of NewReno flow between two Windows 7 hosts with 100% BDP buffer.

Although there are no penalties to throughput caused by longer RTT, it is fair to assume that different applications or specific uses of TCP depend on the RTT. These may suffer a degrade in performance as the RTT grows.

The small *rwnd* which Windows 7 signals has several implications for TCP transfers towards



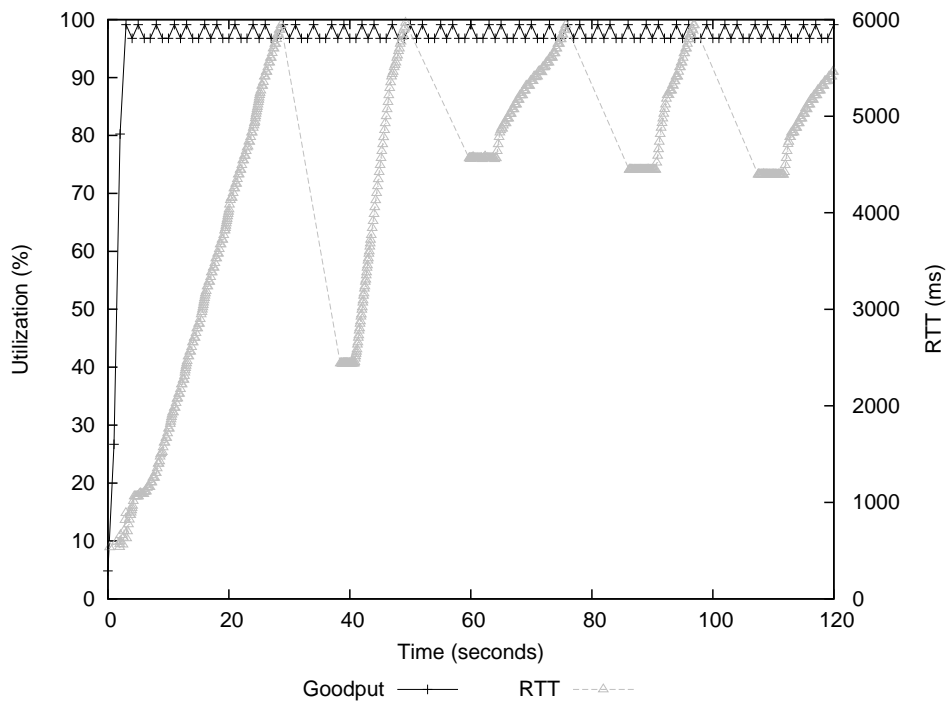


Figure 6.20: RTT and goodput of CUBIC flow between two Linux hosts with 1000% BDP buffer.

a Windows 7 PC. Note that the occurrence of such a situation may be rare since it requires a large elevation of RTT *after* the transmission has started, i.e. the buffer is more or less empty at start of transfer. However, it is not unlikely. As long as the flow is the only utilizer of the link, or a competitor is very conservative, e.g. suffers an identical problem, no implications are experienced. In fact, the receiver will enjoy maximum goodput with a lower RTT than a Linux receiver could expect. However, when another flow is introduced, e.g. NewReno towards Linux, the under-utilization of the buffer will result in an unfair division of bandwidth - at the cost of the original flow. In a mixed-OS/TCP environment with different receivers, it may be necessary to address this problem (no solution has been found). However, a large number of users at the receiving end would most likely mitigate the effect since the buffer would frequently be utilized.

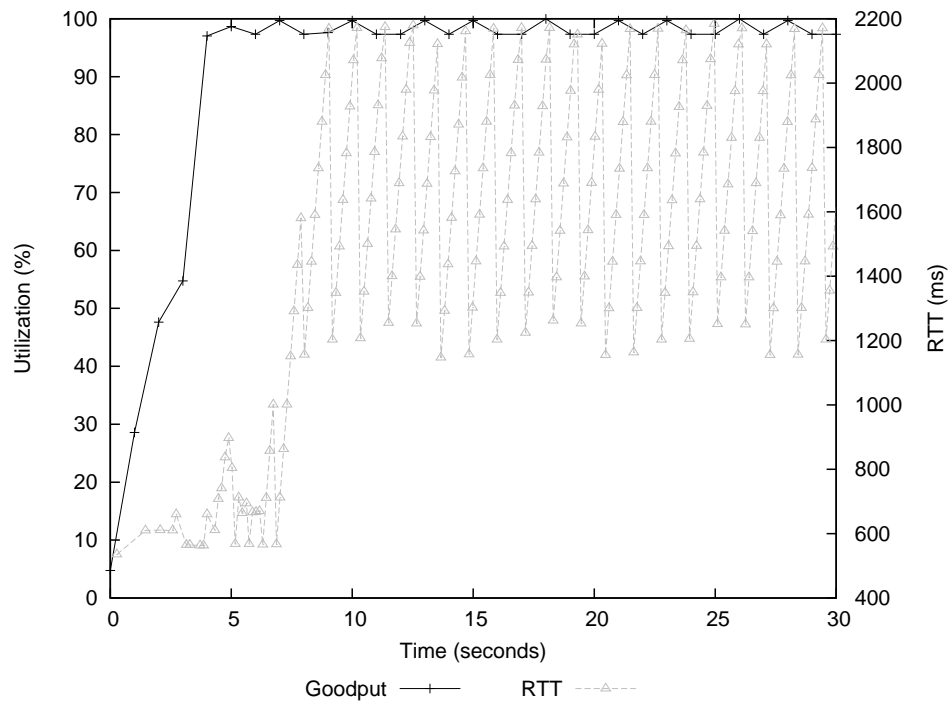


Figure 6.21: RTT and goodput of NewReno flow between two Win 7 hosts, 1000% BDP buffer.

### 6.2.5 Webpages, HTTP, short transfers

The following results try to indicate the impact of a satellite link when downloading various web sites. There are no significant ways a receiver can impact the performance (within reasonable boundaries) on these short downloads, so all measurements are done using Windows 7 and Internet Explorer 8. The web pages were downloaded several times prior to the measurements, allowing the browser to cache whatever content the default settings enable it to. The pages were then downloaded 10 times consequently, all packets captured, and results calculated as an average of the 10 downloads. Duration was measured from the first request or SYN segment until the final FIN (segment with FIN flag set) segment was received, or an obvious end in the transfer occurred. This mixed approach is caused by the fact that not all web servers close the TCP connections when downloading is finished. The type of server and operating system was found using a free online tool at <http://www.netcraft.com>. Note that the results should only be viewed as indications as there are a wide array of parameters and variables which is outside the control of the testbed.

There are several companies which try to measure the distribution of operating systems and web servers used at the Internet. By analyzing the responses of a web server one may identify the type of server and operating system on which it runs. Querying a large amount of servers may then produce an indicator on the distribution of systems. W3techs.com reports that 64.1 % of servers are running Unix and 35.9 % Windows, as of April 2011 [61]. Netcraft.com reports 61 % Unix (assumed most Apache and nginx servers run on Linux), and 18.83 % Windows, as of April 2011 [62]. Similarly, Securityspace.com reports 72.09 % Unix (same assumption), and 17.33 % Windows, as of July 2009 [63]. The reported distributions vary some, which may be anticipated considering the method of measurement. It seems fair to assume that Unix-based web servers has a slightly higher utilization than Windows-based on the Internet.

Table 6.2 shows the penalty induced on download duration in the final column, while the other columns depict properties of the downloads/website: "#TCP" is number of TCP connections utilized during the transfer. "Size" is the total bytes downloaded. While "Size/Conn." is the size divided by the number of connections. Note that the TCP flavor used in the commercial vendor F5 BIG-IP servers are unknown.

The download of unik.no are the most penalized, which may be predicted based on the shorter Initial Window of Windows transmitters (2 segments, compared to 3 for Linux). On the other hand, the download of ubuntu.com is the second most penalized. This is somewhat surprising since Linux are utilized, and the average kB per connection is low. As mentioned, these results should only be viewed as indications due to the range of uncontrollable variables in the tests. One explanation might be that a single TCP flow is responsible for the bulk of the ubuntu.com download.

Webpage	OS	#TCP	Size	Size/Conn.	High RTT penalty(%)
cnn.com	Linux	≈35	≈200 kB	5.7	20.4
vg.no	F5 Big-IP	≈50	≈150 kB	3	26.9
facebook.com	F5 Big-IP	≈14	≈180 kB	12.9	74.0
ubuntu.com	Linux	≈8	≈45 kB	5.6	188.9
unik.no	Win 2008	≈5	≈30 kB	6	379.9

Table 6.2: Penalty and properties of web page downloading for high RTT.

Figure 6.22 shows data transferred (received) during the Slow Start phase of both a Linux and Windows 7 sender. Measurements are done at the receiver, giving the correct and actual delays from a SYN is received (Time = 0) until data is transferred. The figure shows the benefits of the increased Initial Window. Although the senders utilize the same Slow Start algorithm, Linux is able to transmit a larger amount of data in a shorter period of time. This is solely due to a Initial Window of 3 segments (2 for Windows 7), which allows the *cwnd* to grow at a faster rate since it is increased by one segment for every ACK. This confirms the theory explained in Section 4.1.1.3.

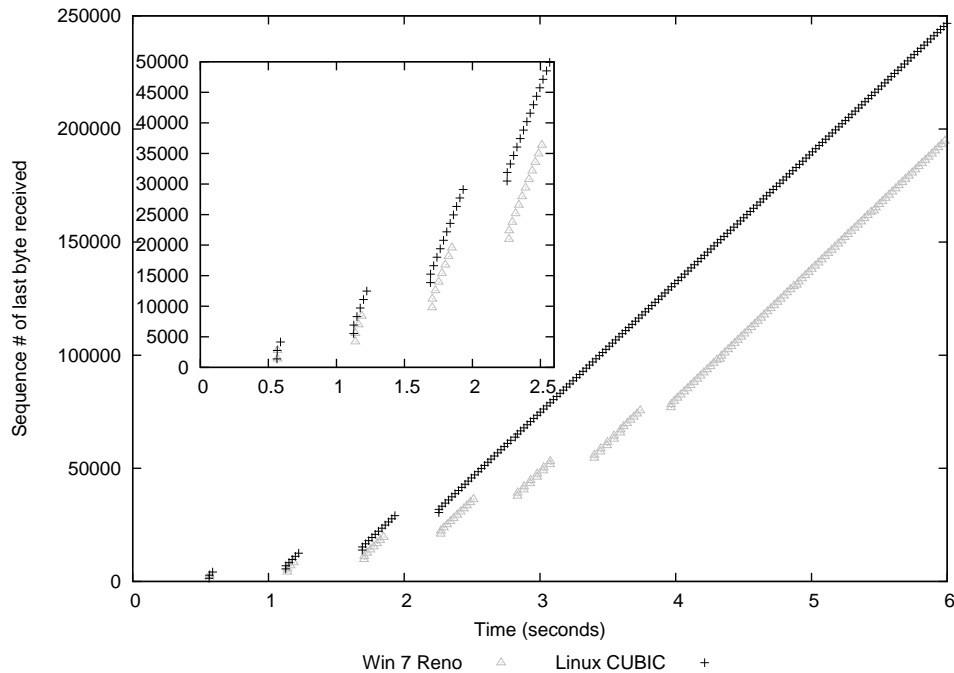


Figure 6.22: Time-sequence graph showing first seconds of CUBIC and NewReno transfers.

Table 6.3 shows the actual average duration of the downloads. Note however, these values are

dependent on a wide range of factors like CPU, web browser, web page content, etc. In addition, a user might be able to interact, read and consider a web page completely downloaded before all data is actually transferred. Consequently, the values should not be considered as representative download durations for these web pages.

	Added RTT	
<b>Webpage</b>	<b>0 ms</b>	<b>528 ms</b>
cnn.com	25.9 s	31.2 s
vg.no	15.4	19.5
facebook.com	5.8 s	16.5
ubuntu.com	5.7	16.5
unik.no	0.9 s	4.2

Table 6.3: Average time to download various web pages for different RTT.

#### 6.2.5.1 Implications

As mentioned, it is very difficult to measure HTTP performance using actual web pages. Some indicators may although be pointed out. It is evident that even short HTTP transfers will be penalized when transmitted across networks with satellite links. The lowest increase in download time was about 20 %, and ranged up to 380 %. There are no mechanisms available for a receiver to greatly impact the performance of these short transfers. The senders initial window, and its expanding of *cwnd* governs the transmission. Consequently, one may expect an added penalty in short transfers of up to about 1 second from a Windows sender when compared to Linux. This can be seen as the gap in between Linux and Windows in Figure 6.22.



## Chapter 7

# Conclusion

This thesis has emulated and measured the performance of TCP NewReno, Compound TCP, CUBIC, TCP Hybla and PEPsal. These are both widespread implementations and some are proposed solutions to the Transport Layer challenges in a long-delay, medium-bandwidth, high-loss environment. A thorough description and analysis is presented, including descriptions of similar technologies like the Space Communications Protocol Specifications - Transport Protocol. The work has been done in the context of the Norwegian Armed Forces acquisition of a communications satellite, and the increasing use of civilian technology, e.g. IP and TCP, in military networks.

There exist a gross unfriendliness from the default Linux implementation CUBIC, towards available Windows 7 TCP flavors (NewReno and CTCP). Utilizing large buffers, a TCP flow originating from a Windows 7 computer may be confined to as little as 5 % of the available bandwidth when competing against a CUBIC flow. Moderate buffers increase the share of Windows 7 flows up to about 20 %. The satellite-tailored TCP Hybla, shows identical or increased unfriendliness. Any solutions should be applied to the sender side of a flow, or at an intermediary proxy. Possible solutions include the use of the Performance Enhancing Proxy PEPsal, tuning of Linux and Windows 7 implementations, or avoiding mixed-OS/TCP environments.

The thesis has also shown a significant poorer performance of NewReno and CTCP in lossy networks when compared to CUBIC. A newer protocol, TCP Hybla, shows promising resilience and good performance, even at large packet losses. The usage of CUBIC or Hybla would greatly benefit a transmitter in such a network. If such a deployment is difficult, the use of a PEPsal at the sender side of a satellite link would significantly improve performance regardless of TCP flavor at sender. PEPsal depends on the ability to read TCP header information, which is encrypted in the black portion of a military network. Several solutions exist to overcome this

problem, including layered encryption schemes, inter-boundary signaling and red enclaves in the black portion. A easily deployed solution would be to position PEPsal at the red-black boundary - however, this may decrease its performance.

Presumably a bug has been identified in the TCP implementation Windows 7. The auto-tuning of the Receiver Window in Windows 7 seem unable to accommodate a large increase in the RTT (e.g. buffer filling up) during a transmission. This imposes constraints on the aggressiveness of the sender, and may penalize the transmission in competition with other flows.

### 7.1 Future work

This thesis has shown the unfriendliness between a Windows 7 flow competing with a Linux flow for bandwidth. This phenomenon should be confirmed in a real-life network with realistic traffic loads and distribution of different TCP flavor flows. Future work may also include a tuning of the Windows 7 TCP implementation (although assumed to be difficult in a Microsoft operating system), to match the aggressiveness of Linux. And similarly, an evaluation of the friendliness of other TCP flavors available to Linux is interesting since deployment is simple. This work should be done at a sender-intensive node in the network, i.e. a server or computer frequently transmitting large amounts of data.

For a general approach, an intermediary proxy may be deployed to decrease unfriendliness for all senders transmitting through it. An evaluation of available implementations is necessary, along with solutions to the dependence of TCP header information at the proxy.

TCP Hybla has shown to be superior in lossy environments. A future work may include the test of Hybla-only transmitters in a network with varying degradations, e.g. Mobile Ad-hoc networks (MANET). These test should also be done in networks without satellite link as this may be a likely scenario for MANETs. In addition to proving Hybla's performance in lossy low-RTT networks, this could indicate its general performance through low-RTT networks - which would be interesting in scenarios where the proxy must operate far away from a satellite link.



# Bibliography

- [1] Norwegian Ministry of Defense. Acquisition of Communications Satellite for Norwegian Armed Forces. Proposition 56 S Prop. 56 S, December 2009.
- [2] M. Molinari and J. Pezeshki. Approaches to using Performance Enhancing Proxies in the Gig Black Core. *MILCOM 2007*, 2007.
- [3] T. Maseng and K. Øvsthus. Telecommunication in tomorrows military. *FFI-FOKUS*, 5, November 2003.
- [4] FFI. NORwegian Modular Network Soldier. *FFI FACTS*, November 2006.
- [5] C. Heiningen. Army develops smartphone framework, applications for the front lines. <http://www.army.mil/-news/2011/04/18/>, April 2011.
- [6] F.D. Kronewitter et.al. HAIPE Compliant TCP Performance Enhancing Proxy for Bandwidth-On-Demand Enviroment. *MILCOM 2008*, pages 1–7, 16–19, 2008.
- [7] H. Yousefi'zadeh, X. Li, and A. Qureshi. A Comparison of TCP Unfriendly-Region Congestion Control Solutions. *2010 Military Communications Conference - Unclassified Program - Networking Protocols and Performance Track*, 2010.
- [8] B. Rossow et. al. Effective use of SatCom in encrypted networks - Final Report. *Thales Norway*, 2011.
- [9] M. Allman and A. Falk. On the Effective Evaluation of TCP. *ACM SIGCOMM Computer Communication Review*, 29, October 1999.
- [10] H.Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4), April 1980.
- [11] ISO/IEC. 7498-1 Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, 1994.
- [12] D. Roddy. *Satellite Communications*. McGraw-Hill, fourth edition edition, 2006.

- [13] M. Hassan and R. Jain. *High Performance TCP/IP Networking*. Alan R. Apt, 2004.
- [14] J. Postel. RFC-793 Transmission Control Protocol, DARPA Internet program protocol specification, September 1981.
- [15] L. S. Brakmo, S.W. O'Malley, and L.L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review*, 24, October 1994.
- [16] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant . *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 42, 2008.
- [17] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22:547–466, 2004.
- [18] K. Ramakrishnan. RFC-3168 The Addition of Explicit Congestion Notification (ECN) to IP, September 2001.
- [19] S. Floyd and K. K. Ramakrishnan. ECN (Explicit Congestion Notification) in TCP/IP. <http://icir.org/floyd/ecn.html>, June 2009.
- [20] M. Allman, D. Glover, and L. Sanchez. RFC-2488 Enhancing TCP Over Satellite Channels using Standard Mechanisms, January 1999.
- [21] M. Allman, V. Paxson, and E. Blanton. RFC-5681 TCP Congestion Control, September 2009.
- [22] Van Jacobson and M. J.Karels. Congestion Avoidance and Control. *ACM SIGCOMM*, November 1988.
- [23] C. Partridge and T.J. Shepard. TCP/IP Performance over Satellite Links. *IEEE Network*, September/October 1997.
- [24] S. Fu, M. Atiquzzaman, and W. Ivancic. Evaluation of SCTP for Space Networks. *IEEE Wireless Communications*, October 2005.
- [25] Y. Zhang, editor. *Internetworking and Computing over Satellite Networks*. Kluwer Academic Publishers, 2003.
- [26] W. Zhenyong, G. Qing, and G. Xuemai. Comprehensive Computational Analysis on TCP in Satellite Links. *Conference on Innovative Computing, Information and Control*, 2006.
- [27] Z. Zhou and J. S. Baras. TCP over GEO satellite hybrid networks. *IEEE/ACM Trans. Netw.*, 14:753–766, 2006.
- [28] S. Dawkins et.al. RFC-2760 Ongoing TCP Research Related to Satellites, February 2000.

- [29] C. Caini and R. Firrincieli. End-to-end TCP enhancements performance on satellite links. *11th IEEE Symposium on Computers and Communications*, 2006.
- [30] C. Caini, R. Firrincieli, and D. Lacamera. Comparative Performance Evaluation of TCP variants on Satellite Enviroments. *IEEE ICC 09*, 2009.
- [31] C. Caini, R. Firrincieli, and D. Lacamera. PEPsal: a Performance Enhancing Proxy designed for TCP Satellite connections. *IEEE*, 2006.
- [32] H. Obata, S. Takeuchi, and K. Ishida. A New TCP Congestion Control Method Considering Adaptability over Satellite Internet. *25th IEEE International Conference on Distributed Computing Systems Workshops*, 2005.
- [33] V. Jacobson, R. Braden, and D. Borman. RFC-1323 TCP Extensions for High Performance, May 1992.
- [34] M. Allman. Improving TCP Performance over Satellite Channels. Master's thesis, Ohio University, 1997.
- [35] M. Marchese. Study and performance evaluation of TCP modification and tuning over satellite links. *International Journal of Satellite Communications*, 19:93–110, 2000.
- [36] M. Allman, S. Floyd, and C. Partridge. RFC-3390 Increasing TCP's Initial Window, October 2002.
- [37] N. Dukkipati et. al. An Argument for Increasing TCPs Initial Congestion Window. *ACM SIGCOMM Computer Communication Review*, 40(3), July 2010.
- [38] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCPs Initial Window. Internet Draft: draft-hkchu-tcpm-initcwnd-01.
- [39] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. RFC-2018 TCP Selective Acknowledgment Options, October 1996.
- [40] M. Allman, V. Paxson, and W. Stevens. RFC-2581 TCP Congestion Control, April 1999. Obsoleted.
- [41] S. Floyd, T. Henderson, and A. Gurtov. RFC-3782 The NewReno Modification to TCP's Fast Recovery Algorithm, April 2004.
- [42] J. Davies. New Networking Features in Windows Server 2008 and Windows Vista. <http://technet.microsoft.com/en-us/library/bb726965.aspx>, February 2006.
- [43] D.J. Leith, R.N. Shorten, and G. McCullagh. Experimental evaluation of Cubic-TCP. *Proceedings of the 6th International Workshop on Protocols for Fast Long Distance Networks*, 2008.

- [44] K. Tan et. al. A Compound TCP Approach for High-speed and Long Distance Networks. *Microsoft Research*, 2005.
- [45] K. Tan et. al. Compound TCP: A Scalable and TCP-Friendly Congestion Control for High-speed Networks . *4th International workshop on Protocols for Fast Long-Distance Networks*, 2006.
- [46] J.C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. *Proceedings of ACM SIGCOMM '96*, pages 270–280, 1996.
- [47] CCSDS. SPACE COMMUNICATIONS PROTOCOL SPECIFICATION (SCPS) - TRANSPORT PROTOCOL (SCPS-TP), October 2006.
- [48] J. Doffoh, R. Mereish, and M. Puckett. Analysis and Comparison of Acceleration Protocol for TCP over Satellite. *MILCOM 2005*, 1:279–285, 2005.
- [49] J. Border et. al. RFC-3135 Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations, June 2001.
- [50] I. Thomson, A. O. Waller, and G. Jones. Performance enhancing proxies and security. *IEE Digest*, 2003.
- [51] Vyatta. *Vyatta System - Basic Routing Reference Guide*. <http://vyatta.org/documentation>, February 2009. Document Revision VC5 v03.
- [52] Vyatta. *Vyatta System - VPN Reference Guide*. <http://vyatta.org/documentation>, February 2009. Document Revision VC5 v03.
- [53] The Linux Foundation. netem. <http://www.linuxfoundation.org/collaborate/workgroups/networking/>, November 2009. Visited in January 2011.
- [54] M. Carbone and L. Rizzo. Dummynet Revisited. *ACM SIGCOMM Computer Communication Review*, 40, 2010.
- [55] C. Caini, R. Firrincieli, and D. Lacamera. A Linux Based Multi TCP Implementation for Experimental Evaluation of TCP Enhancements. *Proc. SCS SPECTS*, pages 875–883, 2005.
- [56] The Linux Foundation. TCP probe. <http://www.linuxfoundation.org>, November 2009. Visited in January 2011.
- [57] The Linux Foundation. TCP testing. <http://www.linuxfoundation.org>, November 2009. Visited in January 2011.
- [58] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM SIGCOMM Computer Communication Review*, 17, 1987.
- [59] S. Kent. RFC-4303 IP Encapsulating Security Payload (ESP), December 2005.

- [60] W.Stallings. *Cryptography and Network Security - Principles and Practice*. Prentice Hall, fifth edition, 2011.
- [61] W3 Techs. Usage of operating systems for websites. [http://w3techs.com/technologies/overview/operating\\_system/all](http://w3techs.com/technologies/overview/operating_system/all), April 2011. Visited in April 2011.
- [62] Netcraft. April 2011 Web Server Survey. <http://news.netcraft.com/archives/category/web-server-survey/>, April 2011. Visited in April 2011.
- [63] Security Space. Web Server Survey. [https://secure1.securityspace.com/s\\_survey/data/200907/index.html](https://secure1.securityspace.com/s_survey/data/200907/index.html), August 2009. Visited in April 2011.



# Appendices





# Appendix A

## Configuration of Vyatta routers

### A.1 Router D102

```
vyatta@D102:~$ show version
```

```
Version : 999.deltattr.07281008
Description: 999.deltattr.07281008
Copyright: 2006-2010 Vyatta, Inc.
Built by : Jon.Andersson@Thales.no
Built on : Wed Jul 28 08:04:40 UTC 2010
Build ID : 1007280804-317b46a
Boot via : disk
Uptime : 20:12:07 up 4:22, 1 user, load average: 0.07, 0.03, 0.01
```

```
vyatta@D102:~$ show interfaces
```

Interface	IP Address	State	Link	Description
eth0	158.38.122.9/29	up	up	
eth1	10.0.0.1/24	up	up	

```
vyatta@D102# show
```

```
interfaces {
    ethernet eth0 {
        address 158.38.122.9/29
        duplex full
        hw-id 00:10:f3:1b:8c:a8
    }
}
```

```
        speed 10
    }
    ethernet eth1 {
        address 10.0.0.1/24
        duplex full
        hw-id 00:10:f3:1b:8c:a9
        speed 10
    }
    ethernet eth2 {
        duplex auto
        hw-id 00:10:f3:1b:8c:aa
    }
    ethernet eth3 {
        duplex auto
        hw-id 00:10:f3:1b:8c:ab
    }
    loopback lo {
    }
}
protocols {
    static {
        route 192.168.0.0/24 {
            next-hop 158.38.122.10 {
            }
        }
    }
}
service {
    ssh {
        port 22
    }
}
system {
    gateway-address 158.38.122.10
    host-name D102
    login {
        user root {
            authentication {
```

```
        encrypted-password $1$06yETpK5$FLkPZtj/C2J.TnP7ywcCG1
        plaintext-password ""
    }
}
user vyatta {
    authentication {
        encrypted-password $1$DlkUAo.g$XqhsUu5vufL9oKXbMo1MIO
    }
}
ntp-server 0.vyatta.pool.ntp.org
package {
    auto-sync 1
    repository community {
        components main
        distribution stable
        url http://packages.vyatta.com/vyatta
    }
}
syslog {
    global {
        facility all {
            level notice
        }
        facility protocols {
            level debug
        }
    }
}
vpn {
    ipsec {
        esp-group ESP-D102 {
            compression disable
            proposal 1 {
                encryption aes256
            }
        }
    }
}
```

```
ike-group IKE-D102 {
    lifetime 28800
    proposal 1 {
        encryption aes256
    }
}
ipsec-interfaces {
    interface eth0
}
site-to-site {
    peer 158.38.122.10 {
        authentication {
            mode pre-shared-secret
            pre-shared-secret asd
        }
        ike-group IKE-D102
        local-ip 158.38.122.9
        tunnel 1 {
            allow-nat-networks disable
            esp-group ESP-D102
            local-subnet 10.0.0.0/24
            remote-subnet 0.0.0.0/0
        }
    }
}
}
```

## A.2 Router D103

```
vyatta@D103:~$ show version
Version : 999.deltattr.07281008
Description: 999.deltattr.07281008
Copyright: 2006-2010 Vyatta, Inc.
Built by : Jon.Andersson@Thales.no
Built on : Wed Jul 28 08:04:40 UTC 2010
Build ID : 1007280804-317b46a
```

## APPENDIX A. CONFIGURATION OF VYATTA ROUTERS

---

Boot via : disk

Uptime : 19:10:53 up 4:23, 1 user, load average: 0.00, 0.00, 0.00

vyatta@D103:~\$ show interfaces

Interface	IP Address	State	Link	Description
eth0	158.38.122.10/29	up	up	
eth1	192.168.0.1/24	up	up	
eth2	-	up	up	
eth3	-	up	down	
lo	127.0.0.1/8	up	up	
lo	::1/128	up	up	

vyatta@D103# show

```
interfaces {
  ethernet eth0 {
    address 158.38.122.10/29
    duplex full
    hw-id 00:10:f3:1b:8c:e4
    speed 10
  }
  ethernet eth1 {
    address 192.168.0.1/24
    duplex full
    hw-id 00:10:f3:1b:8c:e5
    speed 10
  }
  ethernet eth2 {
    address dhcp
    hw-id 00:10:f3:1b:8c:e6
  }
  ethernet eth3 {
    duplex auto
    hw-id 00:10:f3:1b:8c:e7
  }
  loopback lo {
  }
}
protocols {
```

```
static {
    route 10.0.0.0/24 {
        next-hop 158.38.122.9 {
        }
    }
    route 192.168.1.0/24 {
        next-hop 192.168.0.2 {
        }
    }
}
}
service {
    nat {
        rule 1 {
            outbound-interface eth2
            source {
                address 192.168.0.0/24
            }
            type masquerade
        }
        rule 2 {
            outbound-interface eth2
            source {
                address 10.0.0.0/24
            }
            type masquerade
        }
        rule 3 {
            outbound-interface eth2
            source {
                address 192.168.1.0/24
            }
            type masquerade
        }
    }
    ssh {
        port 22
    }
}
```

```
}
system {
  gateway-address 158.37.91.1
  host-name D103
  login {
    user root {
      authentication {
        encrypted-password $1$JQu2U4kr$Y2snGU6aaTKM/FN6/Cpsx.
        plaintext-password ""
      }
    }
    user vyatta {
      authentication {
        encrypted-password $1$TwAtQV8q$yfoC1.4J85q578HUUW11eX/
        plaintext-password ""
      }
    }
  }
  ntp-server 0.vyatta.pool.ntp.org
  package {
    auto-sync 1
    repository community {
      components main
      distribution stable
      url http://packages.vyatta.com/vyatta
    }
  }
  syslog {
    global {
      facility all {
        level notice
      }
      facility protocols {
        level debug
      }
    }
  }
}
```

```
vpn {
  ipsec {
    esp-group ESP-D103 {
      compression disable
      lifetime 1800
      proposal 1 {
        encryption aes256
      }
    }
    ike-group IKE-D103 {
      lifetime 3600
      proposal 1 {
        encryption aes256
      }
    }
    ipsec-interfaces {
      interface eth0
    }
    site-to-site {
      peer 158.38.122.9 {
        authentication {
          mode pre-shared-secret
          pre-shared-secret asd
        }
        ike-group IKE-D103
        local-ip 158.38.122.10
        tunnel 1 {
          allow-nat-networks disable
          esp-group ESP-D103
          local-subnet 0.0.0.0/0
          remote-subnet 10.0.0.0/24
        }
      }
    }
  }
}
```



# Appendix B

## Gnuplot scripts

### B.1 tcpprint

```
#!/bin/bash
if [ $# -ne 2 ];
then
    echo "Usage: tcpprint data output"
    exit 1
fi

gnuplot <<EOF
set style data linespoints
set style line 1 lc rgb "black" pt 1
set style line 2 lc rgb "grey" pt 8

#set title "$1"
set key right bottom
set xlabel "time (seconds)"
set ylabel "Segments (cwnd, ssthresh)"
#set terminal png
set terminal post eps
set output "$2"
plot "$1" using 1:7 ls 1 title "snd_cwnd", \
    "$1" using 1:(\$8>=2147483647 ? 0 : \$8) ls 2 title "snd_ssthresh"
```

EOF

## B.2 Selected script

```
#!/bin/bash
(cat <<EOF
set style data histograms
set style fill solid 1.0 border -1
set datafile separator ","
set style line 1 lc rgb "black" pt 1
set style line 2 lc rgb "grey" pt 8
set key reverse top outside
set boxwidth 0.6
set terminal post eps
set output "$2"
set size 0.9,0.9
set ylabel "Utilization (%)"
set xlabel "Size of buffer (% of BDP)"
set yrange [0 : 100]

#set style histogram cluster gap 5
set key autotitle columnhead
set style histogram rowstacked

plot "$1" using (\$2/4691.60):xticlabels(1) ls 2, \
"$1" using (\$3/4691.60) ls 2 fs pattern 1, \
"$1" using (\$4/4691.60) ls 1
EOF
) | gnuplot -persist
```

## Appendix C

# PEPsal (iptables) script

```
#!/bin/bash

echo "8192 2100000 8400000" >/proc/sys/net/ipv4/tcp_mem
echo "8192 2100000 8400000" >/proc/sys/net/ipv4/tcp_rmem
echo "8192 2100000 8400000" >/proc/sys/net/ipv4/tcp_wmem

SAT_RECV="192.168.1.0/24"
NQ=9
OUT_IFACE="eth1"
CLIENTS_IFACE="eth0"

iptables -t mangle -F
iptables -t nat -F
iptables -t nat -F TCP_OPTIMIZATION
iptables -t mangle -F TCP_OPTIMIZATION

/sbin/iptables -I PREROUTING -t mangle -p tcp --syn -j TCP_OPTIMIZATION
/sbin/iptables -I PREROUTING -t nat -p tcp --syn -j TCP_OPTIMIZATION

iptables -t mangle -I TCP_OPTIMIZATION -i eth0 -s 192.168.1.0/24
-p tcp -j NFQUEUE --queue-num=9
iptables -t nat -A POSTROUTING -s $$SAT_RECV -o $$OUT_IFACE -j MASQUERADE
iptables -t nat -I TCP_OPTIMIZATION -i eth0 -s 192.168.1.0/24
-p tcp -j REDIRECT --to-port 6009
```

---

## Appendix D

# Linux Traffic Control

```
#!/bin/bash
```

```
tc qdisc add dev eth0 handle 1: root htb
tc class add dev eth0 parent 1: classid 1:1 htb rate 2Mbps
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 2Mbps
tc qdisc add dev eth0 parent 1:11 handle 10: tbf rate $1kbit buffer $2
  peakrate $3kbit mtu $4 limit $5

tc qdisc add dev eth0 parent 10:1 handle 101: netem delay $6ms

tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32
  match ip dst 0.0.0.0/0 flowid 1:11
```

---

# Appendix E

## Additional Results

This appendix includes results generated, but not utilized, or considered abundant, in analysis. They are appended here for the interested reader.

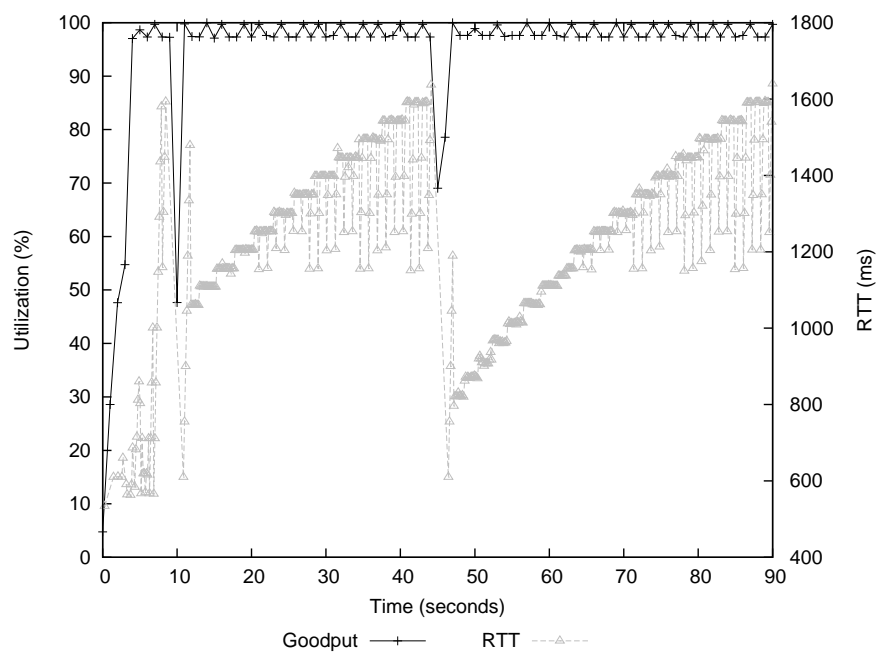


Figure E.1: RTT and goodput of NewReno flow between two Windows 7 hosts with 200% BDP Buffer.

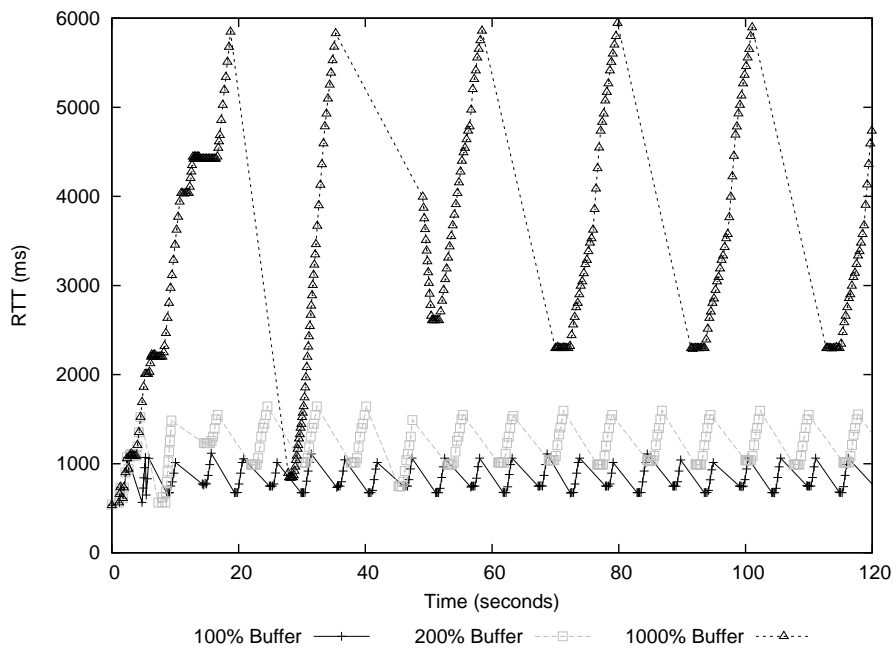


Figure E.2: RTT of Hybla flow between two Linux hosts.

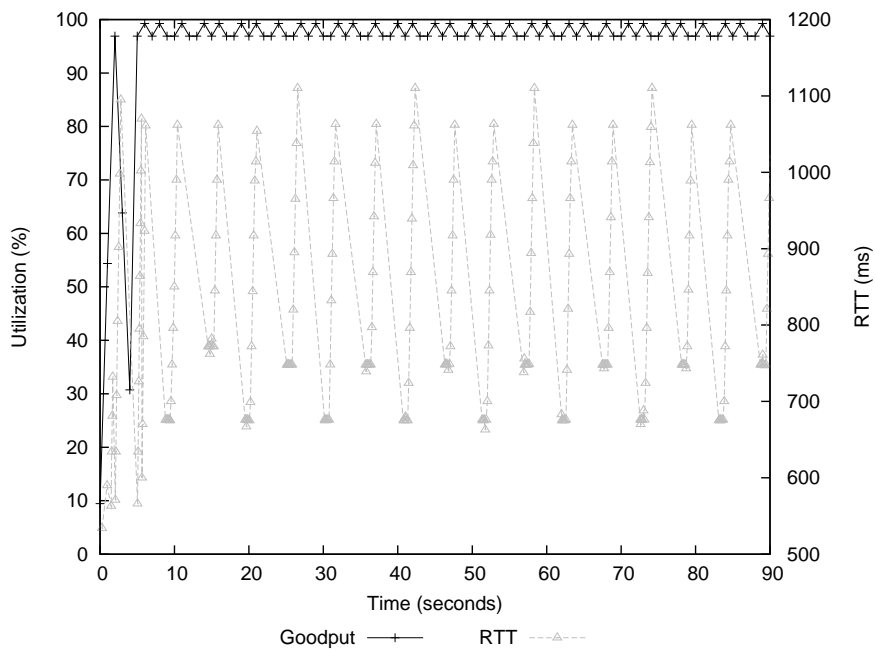


Figure E.3: RTT and goodput of Hybla flow between two Linux hosts with 100% BDP Buffer.



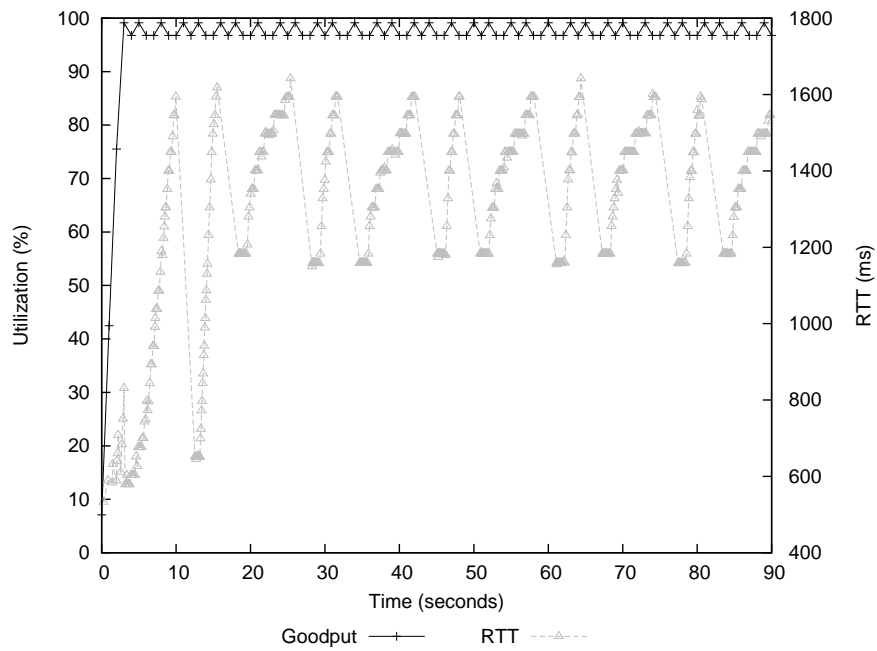


Figure E.4: RTT and goodput of Hybla flow between two Linux hosts with 200% BDP Buffer.

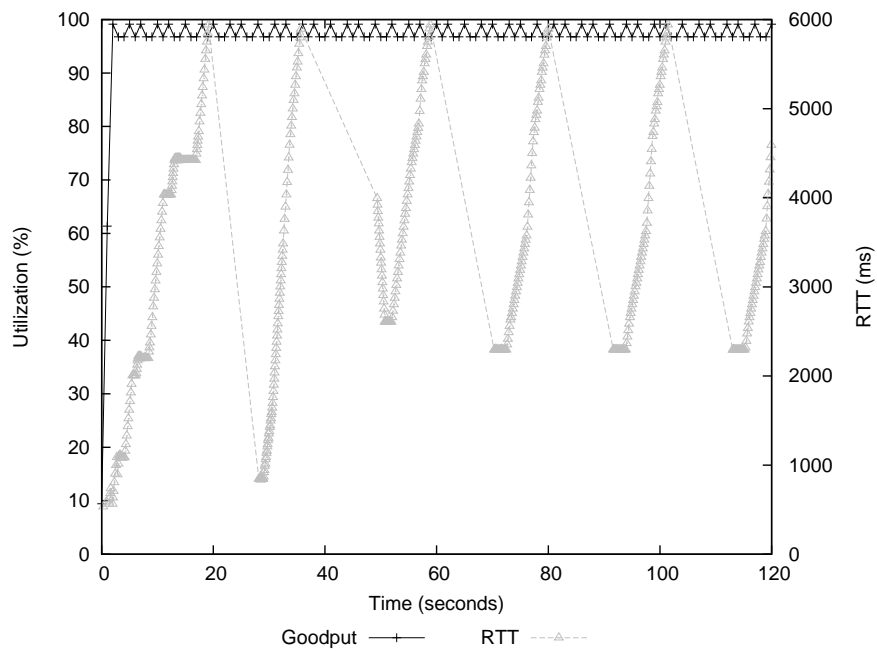


Figure E.5: RTT and goodput of Hybla flow between two Linux hosts with 1000% BDP Buffer.

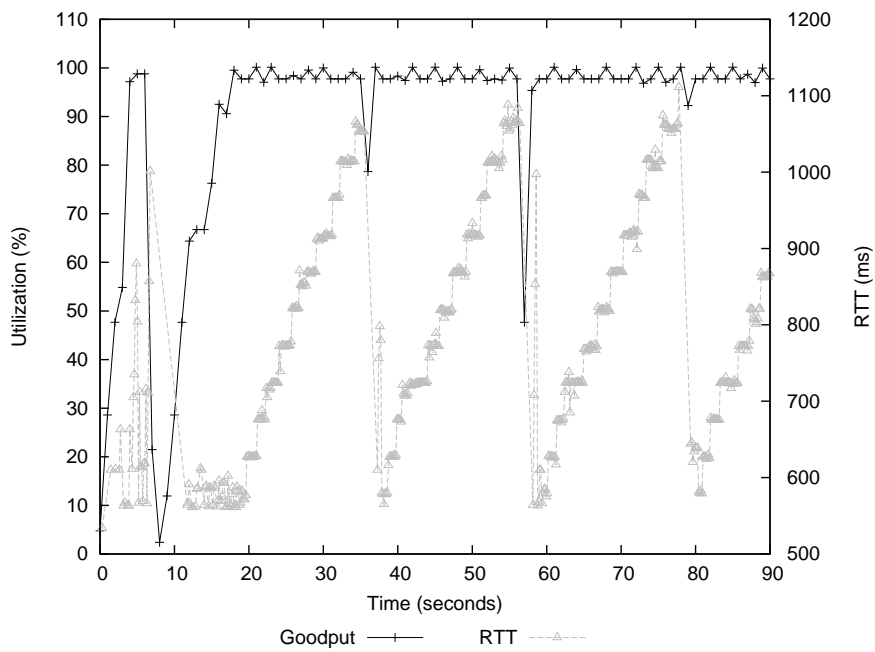


Figure E.6: RTT and goodput of CTCP flow between two Windows 7 hosts with 100% BDP Buffer.

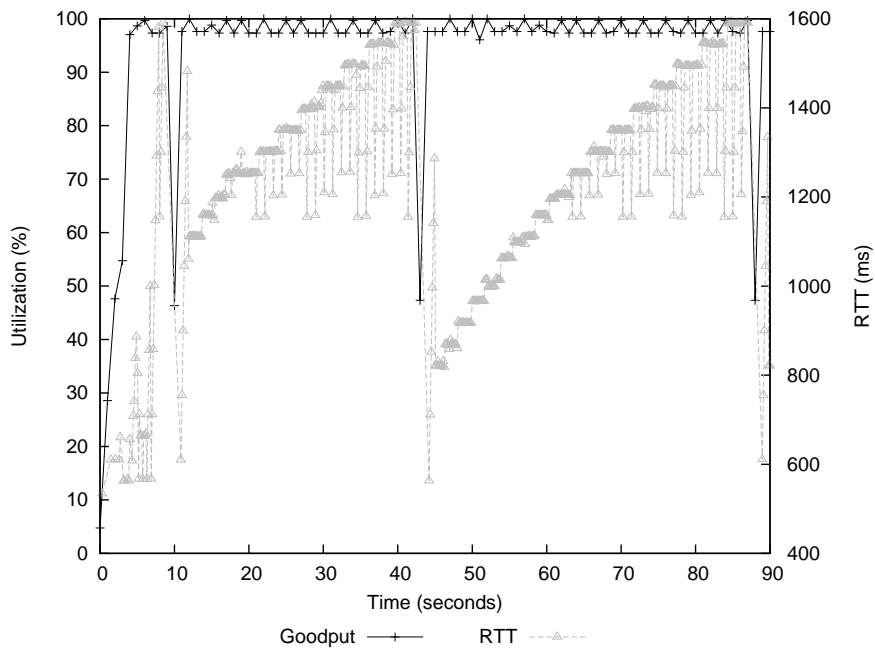


Figure E.7: RTT and goodput of CTCP flow between two Windows 7 hosts with 200% BDP Buffer.

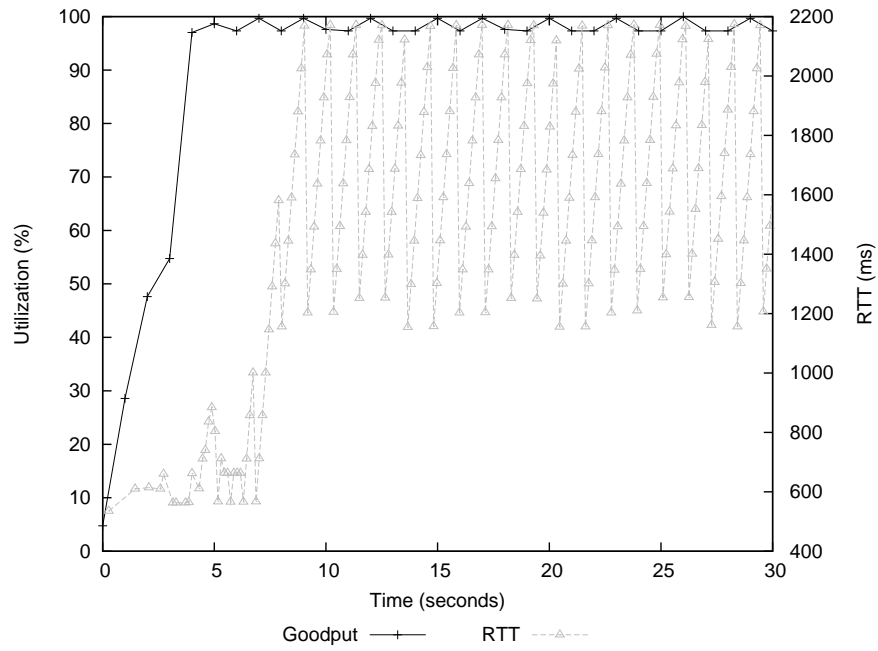


Figure E.8: RTT and goodput of CTCP flow between two Windows 7 hosts with 1000% BDP Buffer.

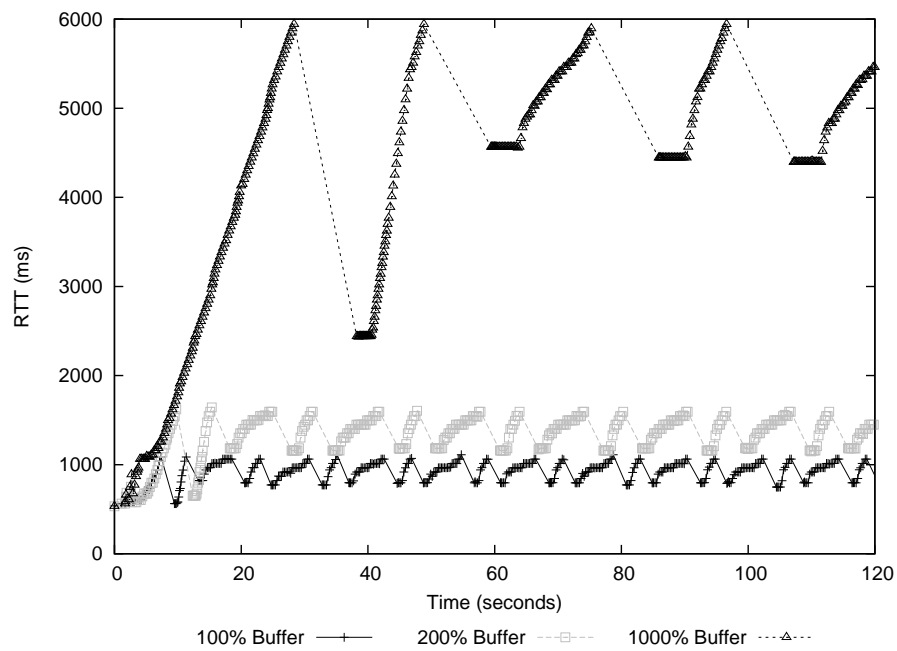


Figure E.9: RTT of CUBIC flow between two Linux hosts.

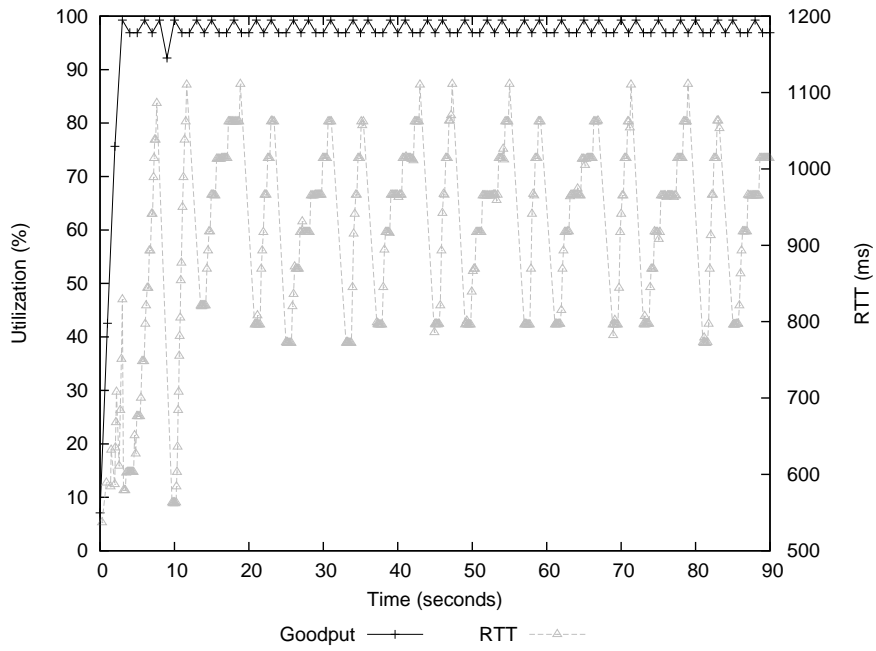


Figure E.10: RTT and goodput of CUBIC flow between two Linux hosts with 100% BDP Buffer.

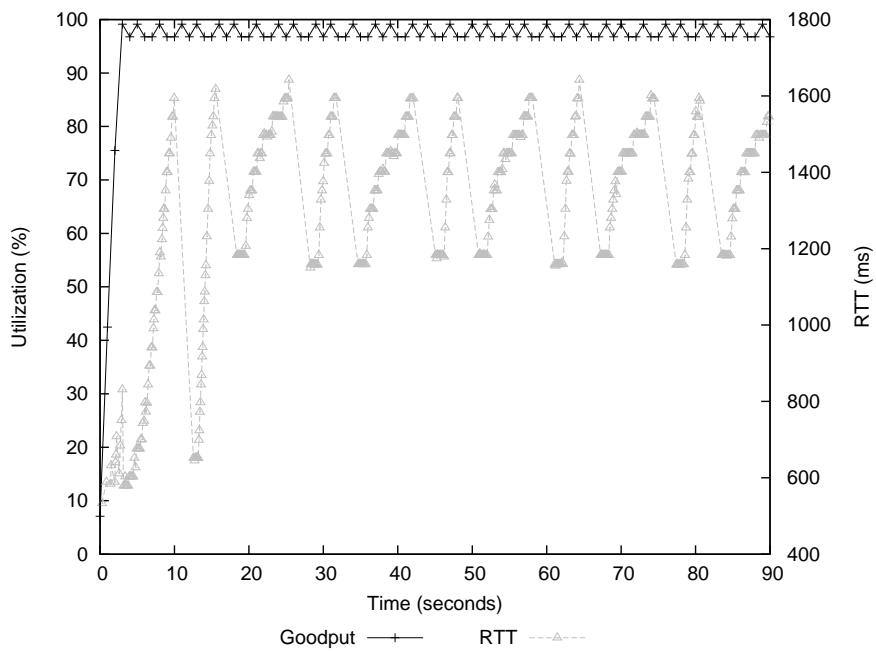


Figure E.11: RTT and goodput of CUBIC flow between two Linux hosts with 200% BDP Buffer.