TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Roman Dunaytsev

TCP Performance Evaluation over Wired and Wired-cum-Wireless Networks

Julkaisu 873 • Publication 873

Tampere 2010

Tampereen teknillinen yliopisto. Julkaisu 873
Tampere University of Technology. Publication 873

Roman Dunaytsev

TCP Performance Evaluation over Wired and Wired-cum-Wireless Networks

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 26th of March 2010, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2010

ISBN 978-952-15-2327-4 (printed)
ISBN 978-952-15-2391-5 (PDF)
ISSN 1459-2045

ABSTRACT

A fundamental design principle of the Internet is best effort packet delivery. The term “best effort” means that the network will do its best to deliver packets, but without any guarantees. In other words, if a problem occurs, the data are discarded. As long as most Internet applications and services require error-free data delivery and proper sequencing, providing reliable transmission of data from source to destination has become an important issue in the Internet. It is worthwhile to note that error control mechanisms, implemented at the lower layers of the protocol stack, cannot fully replace end-to-end error control, since the end-to-end functionality cannot be achieved in a hop-by-hop manner. The Transmission Control Protocol (TCP), a transport layer protocol, provides transparent transfer of data between application layer entities and releases them from any concern with the detailed way in which reliable delivery of data is achieved. To provide reliability, TCP detects errors or lost data and triggers retransmission until the data are correctly and completely received. TCP is also responsible for ensuring that the sending rate is appropriate for the capabilities of the receiving host (flow control), as well as avoiding introducing too much data into the network, which could cause buffers in some bottleneck routers to overflow and start dropping packets (congestion control). This is done by regulating the rate at which the sending host transmits data. It should be emphasized that the current stability of the Internet mainly depends on TCP congestion control mechanisms, while various queue management algorithms, implemented in routers, are helpful but play a less crucial role in practice. Due to such benefits as end-to-end error control and rate adaptation, TCP is heavily used throughout the Internet: about 90% of today’s Internet traffic is carried by TCP.

Since TCP controls the vast majority of bytes and packets transmitted over the Internet, predicting TCP behavior and optimizing its performance is extremely important for satisfying user needs and providing quality of service in the Internet. In protocol analysis and design, analytical modeling has proven to be a powerful and cost-effective tool for studying the behavior of a communication protocol across the entire parameter space of different operating conditions and protocol parameter settings. Therefore, the research presented in this thesis is concentrated on improving the existing analytical models and developing new ones. The first part of the thesis is focused on analytical modeling of TCP performance in wired networks, capturing various aspects of TCP behavior in different scenarios. The second part is dedicated to TCP performance modeling in wired-cum-wireless networks. The developed models provide a general framework for TCP performance evaluation in a wide range of environments and conditions.

PREFACE

The research work presented in this thesis was carried out during the period from October 2005 to October 2009 in the Department of Communications Engineering, Tampere University of Technology, Finland. It was financially supported by the Centre for International Mobility (CIMO), Tampere Graduate School in Information Science and Engineering (TISE), Nokia Foundation, and Ulla Tuomisen säätiö (PricewaterhouseCoopers Oy), all of which are gratefully acknowledged.

The completion of this work would have been impossible without the assistance and efforts of many wonderful people. First of all, I would like to express my deep gratitude to my thesis supervisor, Prof. Jarmo Harju, for his guidance, constant support, and infinite patience. I would like to thank the reviewers of my thesis, Dr. Pasi Lassila and Dr. Pasi Sarolahti for their valuable comments and suggestions. The members of the Networks and Protocols Group, where the work for this thesis was performed, are sincerely acknowledged. Special thanks go to my friends and co-authors Prof. Yevgeni Koucheryavy, Dr. Dmitri Moltchanov, and Dr. Andrey Krendzel for fruitful cooperation and a pleasant working environment. Moreover, I would like to express my gratitude to Dr. Konstantin Avrachenkov at the MAESTRO team, Sophia Antipolis, INRIA, France, Dr. Evgeny Osipov at the Department of Computer Science and Electrical Engineering, Luleå University of Technology, Sweden, and all their colleagues for sharing professional knowledge and experience with me. I also wish to thank Prof. Markku Renfors, the TISE director, and Dr. Pertti Koivisto, the TISE coordinator, for their organizational and administrative efforts. Finally, my warmest thanks go to (in alphabetical order) Elina Orava, Heikki Vatiainen, Jani Tuomisto, Karri Huhtanen, Kirsi Viitanen, Sari Kinnari, Tarja Erälaukko, Tuure Vartiainen, and Ulla Siltaloppi for their professional help and support.

Roman Dunaytsev

Tampere, January 30, 2010

TABLE OF CONTENTS

LIST OF PUBLICATIONS

[P1] R. Dunaytsev, Y. Koucheryavy, J. Harju, The impact of RTT and delayed ACK timeout ratio on the initial slow start phase, in: Proceedings of IPS-MoMe 2005, Warsaw, Poland, March 2005, pp. 171-176.

[P2] R. Dunaytsev, Y. Koucheryavy, J. Harju, The PFTK-model revised, Computer Communications 29 (13-14) (2006) 2671-2679.

[P3] R. Dunaytsev, Y. Koucheryavy, J. Harju, TCP NewReno throughput in the presence of correlated losses: the Slow-but-Steady variant, in: Proceedings of IEEE INFOCOM Global Internet Workshop 2006, Barcelona, Spain, April 2006, pp. 115-120.

[P4] R. Dunaytsev, K. Avrachenkov, Y. Koucheryavy, J. Harju, An analytical comparison of the Slow-but-Steady and Impatient variants of TCP NewReno, in: Proceedings of WWIC 2007, Coimbra, Portugal, May 2007, pp. 30-42.

[P5] R. Dunaytsev, D. Moltchanov, Y. Koucheryavy, J. Harju, Modeling TCP SACK performance over wireless channels with completely reliable ARQ/FEC, *Submitted for publication in International Journal of Communication Systems*.

[P6] D. Moltchanov, R. Dunaytsev, Modeling TCP SACK performance over wireless channels with semi-reliable ARQ/FEC, *Accepted for publication in Wireless Networks* (DOI: 10.1007/s11276-009-0231-9).

LIST OF ABBREVIATIONS

ACK	ACKnowledgement
AIMD	Additive Increase/Multiplicative Decrease
ARPANET	Advanced Research Project Agency NETwork
ARQ	Automatic Repeat reQuest
BCH	Bose-Chaudhuri-Hocquengham
BER	Bit Error Rate
CA	Congestion Avoidance
CE	Congestion Experienced
cwnd	congestion window
CWR	Congestion Window Reduced
DoD	Department of Defense
DS	Delay Spike
D-SACK	Duplicate-SACK
DupACK	Duplicate ACK
ECE	ECN-Echo
ECN	Explicit Congestion Notification
ECT	ECN-Capable Transport
EDGE	Enhanced Data rates for GSM Evolution
FEC	Forward Error Correction
FIFO	First-In, First-Out
FR	Fast Retransmit/Fast Recovery
FTP	File Transfer Protocol
GBN	Go-Back-N
HARQ	Hybrid ARQ
IETF	Internet Engineering Task Force

IMP	IMPatient
IP	Internet Protocol
IW	Initial Window
LAN	Local Area Network
LFN	Long Fat Networks
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NACF	Normalized AutoCorrelation Function
OS	Operating System
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PAWS	Protection Against Wrapped Sequences
PDU	Protocol Data Unit
PFTK	Padhye-Firoiu-Towsley-Kurose
QoS	Quality of Service
RFC	Request For Comments
RS	Reed-Solomon
RTO	Retransmission TimeOut
RTT	Round-Trip Time
rwnd	receive window
SACK	Selective ACK
SBS	Slow-But-Steady
SR	Selective Repeat
SS	Slow Start
ssthresh	slow start threshold
SW	Stop-and-Wait
SWS	Silly Window Syndrome

TCP	Transmission Control Protocol
TD	Triple-Duplicate
TO	TimeOut
UDP	User Datagram Protocol
WAN	Wide Area Network
WWW	World Wide Web

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This chapter provides an introduction to the subject matter of the dissertation. The motivation of the research, the objective and outline of the thesis are also discussed in this chapter.

1.1 Background and Motivation

The Transmission Control Protocol (TCP) – one of the core protocols of the Internet Protocol Suite – has a long history. TCP first started to take shape in 1974, when Vinton Cerf and Robert Kahn published the basic principles of an internetwork protocol [1], which was called the Transmission Control Program. The challenge of constructing such a protocol was to accommodate different networks that might vary in terms of addressing, maximum packet size, delay, throughput, and reliability. The proposed protocol supported global addressing, fragmentation and reassembly, flow control, end-to-end error control and loss recovery. One of the primary goals was to provide reliable data transmission between remote hosts communicating over heterogeneous networks. It should be noted that communication networks do not inherently guarantee error-free data delivery. Therefore, there is a risk that some packets will be corrupted, dropped, delayed, misrouted, duplicated, or delivered out of order. In order to resolve all these problems, the new protocol was responsible for sequencing, issuing acknowledgements (ACKs), retransmitting unacknowledged packets, and detecting duplicates. A consequence of this error detection/retransmission scheme was that in order delivery could be maintained. Moreover, the protocol was also responsible for regulating the flow of packets to and from the processes it serves as a way of preventing the communicating hosts from becoming overloaded with traffic. To perform these functions, the Transmission Control Program opened and closed logical connections between processes. The first demonstration of the new protocol came in July 1977 [2].

Over the next four years, the protocol went through several modifications [3] [4] [5] [6]. The most significant change was that the monolithic Transmission Control Program has been split into two separate protocols: the Transmission Control Protocol (TCP) and the Internet Protocol (IP), which became the foundation for the Internet Protocol Suite (also known as the TCP/IP suite) we use today. According to [6], TCP maps to the host-to-host layer of the DoD (Department of Defense) model just above IP [7], which provides connectionless and best-effort delivery of TCP segments encapsulated in IP packets. The IP packet header mainly contains addressing and control information to enable routers to forward packets to their proper destinations. IP is also in charge of fragmentation and reassembly of packets that exceed the

Maximum Transmission Unit (MTU) size. Above IP, TCP provides end-to-end flow and error control for applications that need reliable and sequential data delivery. Thus, TCP corresponds to the transport layer (layer 4) of the OSI (Open Systems Interconnection) reference model [8]. In 1982, the US DoD adopted TCP as its primary protocol for reliable data delivery over the ARPANET (Advanced Research Project Agency Network) [2]. Eventually, TCP has become a standard protocol for the Internet and spread throughout the world. Acceptance of TCP was also catalyzed by its implementation in the 4.2BSD Unix operating system (OS) in 1983 [2].

During the next few years, computer networks experienced an explosive growth, and with that growth came serious performance problems: the Internet suffered a series of congestion collapses [9]. In fact, these performance problems were mainly caused by the very TCP mechanisms that provide reliability. That is, congestion collapse begins with a steady increase in the load on the network. As hosts send more packets, more packets are queued in the buffers of the routers. Increased delays and lost data induced by congestion lead to timeouts, which trigger retransmissions, but a large number of retransmissions overload the network even further to a point where the network throughput drops to a small fraction of its normal capacity. The rationale of congestion collapse was the absence of end-to-end congestion control and rate adaptation in the early Internet. The only end-to-end rate control that existed at that time was flow control in TCP [6]. But TCP flow control was intended to prevent the sending process from overwhelming the destination process, not preventing congestion somewhere in the network. At that point, it became clear that there is a need for a mechanism to control congestion in the network and, thereby, to control the amount of data injected into the network.

To avoid congestion collapse from occurring, congestion control algorithms were added to TCP [10] [11]. These algorithms are based on the “conservation of packets” principle. The idea is that a new packet is not put into the network until the previous packet leaves. In other words, the TCP sender uses the reception of an ACK as an indication that one of the packets sent earlier has left the network and initiates the next packet transmission without adding to the level of congestion (for this reason TCP is said to be ACK-clocked). As a rule, packets get lost for two reasons: they are damaged in transit and, therefore, must be rejected, or the network is congested and they were dropped due to buffer overflow somewhere on the path. TCP congestion control is based on the assumption that the packet loss rate due to data corruption is very small (much less than 1%), so the loss of a packet is a strong indication of congestion somewhere in the network between source and destination [11]. Upon detection of a packet loss at the TCP sender, the congestion control algorithms are triggered, which reduces the sending rate multiplicatively and/or increases the TCP retransmission timer exponentially. It is primarily these TCP congestion control algorithms that prevent congestion collapse in today's Internet and establish some degree of fairness with concurrent traffic, providing the possibility to share the available bandwidth approximately evenly among all data flows.

Nowadays, TCP is the de facto standard transport protocol for providing reliable data delivery over best-effort IP networks. Most widely used applications and services in the Internet are TCP-based [12]. As a result, about 90% of today's Internet traffic is carried by TCP [13] [14]. As long as TCP controls the vast majority of bytes and packets transmitted over the Internet, predicting TCP behavior in various environments and optimizing its performance is extremely important for satisfying user needs and providing quality of service (QoS) in modern networks.

Since TCP rate control has a significant impact on user-perceived QoS and the efficiency of the overall Internet, it is no wonder that different TCP-related issues have been extensively studied over the last decade. The massive deployment of wireless networks has attracted a lot of attention of researchers and practitioners to TCP performance in wireless and wired-cum-wireless environments (e.g., see [15] [16] [17] and references therein), both because the widespread use of TCP on the Internet and because wireless networks used to transport TCP traffic present very different characteristics from those TCP was tuned to. While wireless channels exhibit a higher bit error rate (BER) than typical wired channels, TCP assumes that all packet losses are caused by network congestion and decreases the sending rate in an attempt to alleviate the congestion. Thus, packets dropped due to data corruption force TCP to slow down the transmission of new data, even though these packet drops do not signal congestion in the network. Hence, TCP performs poorly over lossy wireless channels.

TCP performance evaluation studies can be split into two broad classes: empirical studies and analytical modeling. Although simulations and measurements are extremely useful tools in performance evaluation, it is a difficult task to simulate and explore TCP behavior across the range of all possible operating conditions and different protocol settings. In this case, analytical modeling is extremely beneficial because it allows to study the performance of TCP over the entire parameter space and very easily apply a "what if" test to different scenarios. Moreover, once a model is obtained, it can be used for a number of purposes. Firstly, it can help researchers and engineers to evaluate the existing TCP implementations and to make design decisions about novel TCP algorithms. Secondly, the obtained results can be applied to dimension network resources [18] and/or to develop new queuing policies and scheduling algorithms [19]. Thirdly, an expression of TCP throughput can serve as the basis for a TCP-friendly rate control protocol (also known as equation-based congestion control) [20] [21], a cross-layer performance control system [22] [23], or a technique to speed up large scale network simulations [24]. Indeed, this list is not exhaustive and can be updated as new ideas and proposals are developed.

1.2 Objective and Outline of the Thesis

Based on the above considerations, we can state that studying TCP behavior in different environments is an important task for improving the service provided to users and the efficiency of network resource utilization, while analytical modeling is indispensable for this purpose because the generality of the conclusions is not possible with experimental investigations alone.

There are a number of ways to classify TCP analytical models. For instance, they are often distinguished by:

- the length of the TCP transfer: short-lived, long-lived, and TCP transfers of arbitrary length (see [25] and references therein);
- the approach used to model TCP performance: renewal theory models, fluid models, processor sharing models, control theoretic models, and fixed-point models (see [26] and references therein);
- the number of TCP flows under consideration: single source models and models of multiple TCP sources;
- the type of the underlying network: pure wired, pure wireless (single-hop and multihop), and wired-cum-wireless, where the term "wired-cum-wireless" refers to the scenario in which mobile/wireless users access information stored somewhere on the Internet;
- whether flow-level dynamics is accounted or not: flow-level models take into account the dynamics related to the arrival and departure of TCP flows, while packet-level models assume a fixed number of long-lived TCP flows (usually just one) and consider the underlying network from the perspective of packet-level parameters such as the packet loss probability and the average delay (see [27] and references therein).

The research presented in this thesis is concentrated on improving the existing analytical models and developing new ones. Specifically, the objective of the study is to develop models for:

- short-lived and long-lived TCP flows;
- TCP performance in wired and wired-cum-wireless networks.

All of the models are single source packet-level models and, except the one for short-lived TCP flows, are based on the renewal theory approach. The developed models provide a general framework for TCP performance evaluation and can be used for different purposes as outlined in the previous section.

This dissertation consists of six chapters, one appendix, and six publications referred in the text as [P1], [P2], …, [P6]. In Chapter 1, the motivation of the research and the objective of the thesis are introduced. In Chapter 2, an overview of the TCP functions and the TCP congestion control algorithms is given. The developed models for TCP performance evaluation in wired networks are presented in Chapter 3. A cross-layer model for a TCP connection running over a wired-cum-wireless network is proposed and discussed in Chapter 4. Chapter 5 provides a summary of the publications and the author's contribution to them. Finally, conclusions are drawn in Chapter 6. Since the model for short-lived TCP flows [P1] involves solving a transcendental equation based on a new approach developed by M.A. Eremin and available in Russian only, Appendix A gives a brief introduction to this method.

2. TCP OPERATION

This chapter outlines the details of TCP operation relevant for the research described in the dissertation. In the first place, a short overview of the TCP functions is presented. Then, a thorough description of the TCP congestion control algorithms is given.

2.1 TCP Functions

TCP is a complex and feature-rich protocol. The specification for TCP is defined in a number of RFCs (Request For Comments) published by the Internet Engineering Task Force (IETF). While the core specification consists of just a few RFCs [6] [28] [29], the total number of TCP-related RFCs is really huge: entering "TCP" in the RFC index search engine [30] gives more than 100 results. In [31], a good summary of the TCP specification documents is provided. However, it dates back to 2006 and does not include the latest updates (e.g., [32] [33]).

TCP fulfills the following functions:

- ordered data transfer and data segmentation;
- multiplexing/demultiplexing;
- flow control;
- error control;
- congestion control.

The basic operation of TCP in each of these areas is described in the next sections.

2.1.1 Data Transfer

TCP is a byte-oriented protocol meaning that it transmits application layer data as an unstructured, but ordered, stream of bytes. TCP transfers a series of bytes, known as a segment, from one host to another. In order to do so, the TCP sender passes segments to the IP layer for placement in IP packets, which will be later routed to the destination host. In turn, the TCP receiver accepts incoming segments from the IP layer and delivers the data bytes to the appropriate application layer process in the same order in which the data were sent.

Both TCP sender and receiver need to agree on the maximum segment size (MSS) they can handle on that connection in each direction (note that TCP connections are full-duplex, so traffic can go in both directions). The MSS option is used to indicate the maximum size of segments that the host can accept. This option is only used at the time a TCP connection is established (in segments where the SYN flag is set on). In accordance with [34], the MSS value advertised in the MSS option field is equal to the sender's MTU minus the IP header size (without options) and the TCP header size (without options). Thus, the MSS is 40 bytes less than the MTU. However, some OSs can send only segments with lengths that are multiples of 512 bytes. For instance, even if the MTU of 1500 bytes is supported, the segments sent will be only 1024 bytes long (instead of 1460 bytes, as allowed by the MSS).

2.1.2 Multiplexing/Demultiplexing

TCP employs a multiplexing/demultiplexing mechanism to allow multiple application layer processes within a host to simultaneously access the network via a single interface. This mechanism assigns 16-bit identifiers, called port numbers, to every instance of every application that is using TCP. The combination of source and destination IP addresses, plus source and destination port numbers uniquely identifies a TCP connection. Using port numbers, the sending TCP can multiplex segments from different processes onto a single link, while the receiving TCP can demultiplex incoming segments to the correct destination processes.

2.1.3 Flow Control

In order to prevent inefficient use of the network bandwidth and other resources, TCP is responsible for flow and congestion control: it ensures that data are transmitted at the rate consistent with the capacities of both TCP receiver and intermediate links in the end-to-end path. TCP flow control is a host-oriented feature trying to prevent a fast sender overloading a slow receiver. To control the amount of data that can be sent at a time, data transfer between TCP peers is performed using the so called sliding window algorithm. The window is the amount of data in the stream of bytes to be transmitted that the TCP receiver allows the TCP sender to send. The TCP sender can transmit only those bytes of the stream that lie inside this window. New data can be sent only with the receiver's permission. In other words, the TCP sender can only transmit a window of segments before receiving any feedback from the TCP receiver. The window moves ("slides") along the byte stream as the TCP receiver acknowledges new data.

TCP flow control can be divided into two types: receiver-side and sender-side flow control. The primary objective of receiver-side flow control is to provide information about the available space in the receive buffer, while the main goal of sender-side flow control is to limit the data flow in response to this feedback information. The window field in the TCP header is used to represent the available space in the receive buffer (also referred to as the receiver advertized window or the receive window, rwnd, for short) and provides a flow control mechanism for the TCP connection. The TCP receiver uses the window field to define a window of sequence numbers beyond the last acknowledged sequence number that the TCP sender is allowed to transmit. Because the amount of free space in the receive buffer may change over time, the window size may vary dynamically during lifetime of the TCP connection.

The silly window syndrome (SWS) refers to the situation when much smaller segments are exchanged across the TCP connection than allowed by the MSS [35]. It can be caused by either of the two involved parties. Firstly, this phenomenon arises when the TCP receiver advances the right window edge whenever it has any new buffer space available to receive data, which in turn causes the TCP sender to transmit small segments. Secondly, the TCP sender can transmit small segments when the data to be sent increase in small increments (instead of waiting for additional data to send a larger segment). As a result, transferring small segments consumes extra network bandwidth due to a large protocol header overhead and introduces unnecessary computational overhead at each node along the end-to-end network path. To increase the efficiency of data transmission, TCP avoids sending and receiving small segments by using the Nagle algorithm (named after its inventor, John Nagle) and the SWS avoidance algorithm. To prevent the sending of small segments, the Nagle algorithm [9] [28] allows for small amounts of data to accumulate in the send buffer and not be sent until an ACK is received for the data previously sent. The SWS avoidance algorithm [28] prevents small window advertisements in the case where a receiving process reads data from the receive buffer slowly. Then, the TCP receiver waits until the available space reaches either 50% of the total buffer size or one MSS, whichever is smaller.

The window field in the TCP header consists of 16 bits, so the maximum rwnd size is $2^{16}-1=65,535$ bytes. Consequently, the TCP sender can have only 65,535 bytes of data in transit at a time, which places a limit on the maximum achievable throughput as $\mathrm{rwnd}/\mathrm{RTT}$, where the RTT (round-trip time) is the time needed for a segment to travel from source to destination and back. To increase performance over long-distance and high-speed networks, often referred to as long fat networks (LFNs), the TCP window scale option has been defined in [36], allowing the TCP receiver to advertise a larger window size than 65,535 bytes. The window scale factor is simply a two’s power multiplier to be applied to the 16-bit advertised window and is between 0 (no scaling performed) and 14. Hence, the biggest possible window is now equal to $65{,}535\times2^{14}{=}1,073,725,440$ bytes. Like the MSS option, the window scale option should only appear in the SYN and SYN/ACK segments during the connection setup. Thus, the scale factor is fixed in each direction after the TCP connection is established.

2.1.4 Error Control

IP provides best-effort packet transmission in which packets may be corrupted, dropped, delayed, misrouted, duplicated, or delivered out of order. The end-to-end data integrity depends on TCP error control, which is based on the TCP checksum calculation and the use of sequence numbers and ACKs. The 16-bit TCP checksum field provides a means for detecting errors in the received data and covers the whole TCP segment plus some parts of the IP header (referred to as a pseudo header). The TCP pseudo header contains the source and destination IP addresses, the code of the transport layer protocol (6 in this case), and the length (in bytes) of the segment.

Each TCP segment is identified by a 32-bit sequence number, which specifies the position of the first data byte of this segment in the sender's byte stream. Upon receiving a segment successfully, the TCP receiver sends an ACK back that contains the value of the next sequence number the TCP receiver is expecting to obtain. If the ACK is not received within a specific time interval, called the TCP retransmission timeout (RTO), the TCP sender assumes that the segment has been lost and will retransmit it.

Since the amount of sequence numbers is finite and wraps around when the limit is reached, extremely high-speed networks present a real risk of multiple different segments bearing the same sequence number. The protection against wrapped sequences (PAWS) is an algorithm that makes use of the TCP timestamps option [36]. These timestamps create a clear distinction between new segments and old duplicates with wrapped sequence numbers without having to increase the sequence number field length.

TCP is a connection-oriented protocol. Hence, connections must be established when needed and terminated when their purpose is completed. The initial sequence number that each host selects for the transmission is communicated to the other end in the SYN and SYN/ACK segments during the connection establishment phase. Moreover, during the connection setup both TCP peers agree on the MSS, window scale, timestamps, and other TCP options.

2.1.5 Congestion Control

Standard TCP congestion control is reactive congestion control in the sense that it uses either packet losses or excessively delayed packets to trigger congestion alleviation actions. In steady state, TCP congestion control is based on the following strategy. The TCP sender increases the window size linearly until a packet loss occurs. Once a packet loss is detected, the TCP sender halves the window size and, consequently, its sending rate. Therefore, this rate control strategy is commonly called additive increase/multiplicative decrease (AIMD). The TCP congestion control algorithms are described in detail in the next sections. It should be emphasized that TCP flow control and congestion control are used in combination, so TCP must limit the sending rate to the minimum of what the receive buffer can accept and what the network can effectively carry.

Table 2.1 summarizes the TCP functions.

Table 2. 1 Summary of the TCP functions

TCP function	**Implementation**	**Basic standards track RFCs**
Ordered data transfer and data segmentation	Connection establishment and termination MSS option Path MTU discovery	793, 1122 879 1191 [37]
Multiplexing/demultiplexing	Port numbers	793, 1122, 1700 [38]
Flow control	Receive window Silly window syndrome avoidance Nagle algorithm Window scale option	793, 1122 813, 1122 896, 1122 1323
Error control	Checksum computation Sequence numbers Protection against wrapped sequences Cumulative and selective ACKs Retransmission timer and retransmissions	793, 1071 [39] 793 1323 1122, 2018, 2883 1122, 2988
Congestion control	Karn’s algorithm Initial window Slow start Congestion avoidance Fast retransmit and fast recovery ECN-support	2988 3390 2581 2581 2581, 3782, 3517 3168

2.2 TCP Congestion Control Mechanisms

The rate at which segments are injected into the network is governed jointly by TCP flow control and congestion control mechanisms. However, while over the last two decades TCP congestion control continues to be a hot topic in academic research as well as in engineering practice, TCP flow control has not gained so much attention in terms of performance evaluation, analysis, and optimization. This can be explained as follows:

- In TCP flow control, by advertising the rwnd size with each ACK, the TCP receiver tells the TCP sender how much data can be sent and successfully stored, thus providing explicit information about the available space in the receive buffer. Armed with this knowledge, the TCP sender can fine-tune its sending rate to avoid overflow and, if possible, underflow of this buffer. As a result, assuming no bandwidth limitations and restrictions imposed by the TCP congestion control algorithms, the sending rate will be mainly limited by application and hardware constraints, rather than TCP performance. Moreover, using window scaling, the TCP sender can transmit up to 1 GB of data per RTT, which is much larger than most networks can now handle. And even if the TCP window scale option is not enabled by default, there are a lot of utilities for adjusting and tweaking TCP settings (e.g., [40] [41]). Hence, TCP flow control, in its current form, is not a performance bottleneck by itself.
- On the other hand, since no explicit information about the available bandwidth on the end-to-end path is provided, the TCP sender infers the state of the network by detecting various signs of congestion such as lack of ACKs at the expected time (implicit indication) or receipt of packets with a special congestion indicator (explicit notification). Without having a clear picture of what is going on in the underlying network, the TCP sender is forced to rely on the AIMD strategy and some other heuristics to effectively mitigate network congestion and efficiently utilize network resources. However, this saw-tooth behavior of the window size leads to a large (and sometimes unnecessary) variation in the sending rate and, therefore, sub-optimal performance. Since TCP congestion control, being responsible for rate adaptation, plays an important role in maintaining QoS for end users, it deserves special attention and careful investigation.

2.2.1 Acknowledgements

TCP uses a feedback mechanism in the form of positive ACKs of the transmitted segments (there are no negative ACKs in TCP) to achieve rate control and provide reliable delivery. Thus, TCP congestion control is based on the flow of ACKs transmitted by the TCP receiver to inform the TCP sender what data have been successfully delivered, so the TCP sender deduces information about network congestion and lost data by examining these ACKs.

When TCP receives a segment from the other end of the TCP connection, it sends an ACK back to the source. However, most TCP implementations use the delayed ACK algorithm as specified in [28]. This causes the receiving TCP to delay ACKs under certain circumstances, which allows to transmit an ACK and data (if any) in a single segment. According to [28], the receiving TCP should send an ACK if one of the following conditions is met:

- a new segment arrives and no ACK was sent for the previously received segment;
- a segment is received, but no other segment arrives within 500 ms (typically, 200 ms);
- an incoming segment fills in all or part of a gap in the sequence space of the receiver's byte stream.

Moreover, the receiving TCP should generate an immediate ACK when an out-of-order segment is received (also known as a "duplicate" ACK). The purpose of this duplicate ACK is to inform the source that a segment was received out of order and which sequence number is expected.

TCP ACKs are cumulative, so they acknowledge that the TCP receiver has correctly received all bytes up through the acknowledged sequence number minus one. Early TCP implementations relied only on this cumulative ACK scheme in which received segments that are not at the left edge of the receive window are not acknowledged. Later it was found that TCP performs poorly when multiple segments are lost from a window of data, since bursty losses generally cause TCP to lose its ACK-based clock, reducing the overall throughput. In order to improve TCP performance during loss recovery in the face of multiple dropped segments, the selective acknowledgement (SACK) option, the use of the SACK option for acknowledging duplicate segments (the D-SACK extension), and the SACK-based loss recovery algorithm have been specified in [42] [43] [44], respectively. Using selective ACKs, the TCP receiver can inform the TCP sender about all segments that have arrived successfully (including non-contiguous and duplicate blocks of data), so the TCP sender needs to retransmit only the segments that have actually been lost.

2.2.2 Retransmission Timer

TCP uses a retransmission timer to ensure data delivery in the absence of any feedback from the other end of the TCP connection. When TCP sends a segment containing data, it retains a copy of the data in the retransmission queue and starts the retransmission timer. If the data are not acknowledged before the retransmission timer expires (i.e., a timeout event occurs), the data are retransmitted. When the segment is acknowledged, TCP removes the data from the retransmission queue. Thus, the TCP retransmission timer indicates when a segment should be retransmitted if no ACK is received. Because of the changing network conditions, the retransmission timeout value must be adjusted dynamically. The algorithm used to compute and manage the retransmission timer is described in [45]. It is based on taking RTT samples (at least one RTT measurement per RTT). Until the first RTT measurement has been made, the initial retransmission timeout value is set to 3 seconds.

One problem that arises with the dynamic estimation of the RTT value is what to do when a timeout event occurs and a segment is retransmitted. When the ACK arrives, it is unclear whether this ACK refers to the original transmission or the latter one. The solution to this retransmission ambiguity problem is known as Karn's algorithm [46] [45] (named after its inventor, Phil Karn). This algorithm dictates that RTT measurements should be ignored for any segment that has been retransmitted. Instead, the retransmission timeout value is multiplied by a factor of two every time a retransmission is repeated, up to some maximum timeout value. Thus, when the sender's retransmission timer reaches zero and the first unacknowledged segment in the send window is resent, the retransmission timer is set to twice the initial value.

2.2.3 Initial Window

With old TCP implementations, the TCP sender started a data transfer by sending multiple segments, up to the window size advertised by the TCP receiver. This is highly efficient when both hosts (source and destination) reside on the same local area network (LAN), but when they are separated by a number of intermediate links with unknown conditions (bandwidth, traffic load, etc.), this can lead to immediate congestion. To prevent an inappropriate amount of traffic being injected into the network, a TCP data flow begins with a small initial window (IW) and then increments it. In the early standards [11] [29], the IW was limited to one or two full-sized segments (i.e., containing the maximum number of data bytes permitted by the MSS). The current standard [47] specifies the upper bound for the IW as

$$\mathrm{IW} = \min\left(4\mathrm{MSS}, \max\left(2\mathrm{MSS}, 4380 \text{ bytes}\right)\right). \tag{2.1}$$

Note that (2.1) allows TCP to transmit up to three segments initially in the common case when using 1500-byte packets. As was demonstrated in [48], the majority of Web servers use an IW of one or two segments, and the rest use an IW of three or four segments.

2.2.4 Slow Start

In addition to the rwnd (i.e., the available buffer space advertised in ACKs), TCP maintains one more window called the congestion window (cwnd), which is similar in concept to the rwnd in that it can be increased or reduced, although these actions are taken according to the ability of the underlying network to handle the amount of data being sent, rather than the capacity of the TCP receiver. Since these two windows represent the amount of data (in bytes) that the TCP sender is allowed to have in transit, the maximum amount of data that can be sent unacknowledged at a time is given by $\min(\mathrm{rwnd}, \mathrm{cwnd})$.

To probe the network path and to determine how much bandwidth is available, TCP uses an algorithm called slow start [29]. When a new TCP connection is established, the cwnd is set to the IW size and the TCP sender starts transmitting data. For every ACK received that acknowledges new data, the cwnd is incremented by the number of bytes in the sender's MSS (i.e., by one full-sized segment). This results in an exponential growth in the number of segments that can be sent per RTT.

During slow start, the cwnd increases exponentially over time until a packet loss occurs or the cwnd reaches the slow start threshold (ssthresh). When the cwnd exceeds the value of the ssthresh, TCP enters the congestion avoidance mode. As a rule, the initial value of the ssthresh is set to 65,535 bytes [11]. In case of a packet loss, TCP interprets it as the evidence that the network is experiencing congestion and reduces the size of the cwnd, which in turn slows down the sending rate and helps to alleviate the congestion problem.

2.2.5 Congestion Avoidance

After the ssthresh is reached, the TCP connection moves into the congestion avoidance phase [29]. In this phase, the cwnd is incremented by $\mathrm{MSS} \times \mathrm{MSS}/\mathrm{cwnd}$ bytes for every non-duplicate ACK. Therefore, when the TCP receiver acknowledges every received segment, the cwnd effectively increases by one full-sized segment for each successfully transmitted window. By using the delayed ACK algorithm, the TCP receiver refrains from acknowledging every incoming segment and sends one ACK for every second segment. Since the TCP sender increases the size of the cwnd based on the number of arriving ACKs, reducing the number of ACKs slows the cwnd growth rate. However, the increase in the cwnd should be at most one segment each RTT, regardless how many ACKs are received in that RTT [11]. Congestion avoidance continues until congestion is detected.

2.2.6 Fast Retransmit and Fast Recovery

TCP detects a segment loss in two different ways. The first way is by a timeout event. The TCP sender starts the retransmission timer when it sends a window of segments and assumes that the data were lost if no ACK has been received within the specified period. Then the TCP sender sets the ssthresh to be one-half the amount of data in flight (also known as the flight size) or two full-sized segments, whichever is larger:

$$\text{ssthresh} = \max\left(\text{FlightSize}/2,\, 2\text{MSS}\right). \tag{2.2}$$

It also sets the cwnd to the size of one full-sized segment and enters slow start. Thus, after retransmitting the lost segment, the TCP sender uses the slow start algorithm to increase the cwnd from one full-sized segment to the new value of the ssthresh. When the ssthresh is exceeded, the TCP sender enters the congestion avoidance phase.

The other way TCP can detect a segment loss is by the arrival of duplicate ACKs, which is known as fast retransmit [29]. When a duplicate ACK is received, the TCP sender does not know if this is because a data segment was lost or because it was delayed and received out of order at the TCP receiver. Then the TCP sender waits for a small number of duplicate ACKs to be received. In order to provide timely detection of lost data, the fast retransmit algorithm is triggered when the TCP sender receives three duplicate ACKs (i.e., four identical ACKs in a row). Thus, after receiving the third duplicate ACK, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

To speed up the recovery of the sending rate after congestion in the network has been detected and eliminated, a new algorithm, called fast recovery, has been specified in [11]. The fast retransmit and fast recovery algorithms are usually implemented together as follows. When the third duplicate ACK is received, the ssthresh is set to the value given in (2.2). The lost segment is retransmitted and the cwnd is set to the ssthresh plus three full-sized segments. For each additional duplicate ACK received, the cwnd is incremented by one full-sized segment. A new segment is transmitted, if allowed by the updated value of the cwnd and the value of the rwnd advertised in the duplicate ACKs. When the next ACK arrives that acknowledges new data, the cwnd is set to the ssthresh and the TCP sender enters the congestion avoidance phase. The reason for not performing slow start is that the receipt of duplicate ACKs not only indicates that a packet has been lost, but also that other packets have left the network and, therefore, there are still data flowing between the two hosts. The fast retransmit algorithm first appeared in TCP Tahoe and was followed by slow start. The fast recovery algorithm appeared in TCP Reno. Later, a number of new features for improving TCP loss detection and recovery have been proposed. Among the most known and widely adopted are the NewReno modification to the fast recovery algorithm [49] and the SACK-based loss recovery algorithm [44].

2.2.7 Explicit Congestion Notification

Routers drop packets for a variety of reasons, but the most common is loss due to buffer overflow or when the queue size exceeds a certain threshold [50]. TCP detects congestion through the occurrence of losses and reduces the sending rate. However, detecting and recovering lost data can be a lengthy process, especially if multiple packets are lost from one window of data and/or there are not enough duplicate ACKs arriving at the TCP sender to trigger fast retransmit/fast recovery. To avoid performance problems associated with dropped packets due to congested routers, a new mechanism, called Explicit Congestion Notification (ECN), has been developed [51]. It provides a means for intermediate routers to set a congestion signaling flag in packets from ECN-capable TCP connections and, thus, to notify the hosts of impending network congestion explicitly, instead of signaling congestion by dropping packets.

ECN support in TCP uses two flags of the reserved field in the TCP header: the ECN-Echo (ECE) flag and the Congestion Window Reduced (CWR) flag. In turn, ECN support in IP uses two flags in the IP header: the ECN-Capable Transport (ECT) flag and the Congestion Experienced (CE) flag. Let us consider basic ECN operation.

- During the connection setup, both TCP peers exchange information about their willingness to use ECN: the host that performs an active open sets the ECE and CWR flags in the TCP header of the SYN segment, while the host that does a passive open sends the SYN/ACK segment but only with the ECE flag. Then, ECN-capable hosts send TCP segments with the ECT and CE flags in the IP header set to either 10 or 01.

- If ECN is not in use and congestion is detected at an intermediate router, packets are dropped as usual. If congestion is detected at ECN-capable routers, then packets are not dropped unless the congestion is very severe. Instead, a router that has congestion imminent sets the ECT and CE flags in the IP header to 11.

- When the receiving host receives this packet, it sets the ECE flag in the TCP header and continues setting this flag in subsequent ACKs.

- When the sending host receives the ACK with the ECE flag, it acts as if a single packet has been dropped: it triggers the congestion avoidance algorithm and halves the cwnd size. Then the sending host sets the CWR flag in the TCP header of the next segment in order to acknowledge the reception of the congestion notification.

- On receipt of the segment with the CWR flag set on, the receiving host stops setting the ECE flag in subsequent ACKs.

It is worthwhile to note that ECN is a dual-layer mechanism, which involves interaction between routers and hosts, thus the potential improvement in TCP performance can only be achieved when both source and destination hosts support ECN and ECN-capable routers are deployed along the path.

2.3 TCP Implementations

The conventional TCP implementations include TCP Tahoe [10], TCP Reno [29], TCP NewReno [49], and TCP SACK [44]. During the last decade, a large variety of TCP modifications have been proposed, ranging from minor changes to the present implementations to completely new design approaches. As a rule, these modifications include algorithms specifically developed to improve TCP performance in different scenarios and networking environments (such as startup behavior, loss recovery, wireless environments, etc.). However, such modifications are usually non-standard and should be used with care, understanding their consequences, especially those concerned with Internet stability and fairness.

The massive deployment of high-speed networks has attracted a lot of attention to TCP performance in LFNs. This is because the standard congestion control algorithms, developed more than a decade ago, fail to utilize the available bandwidth effectively and provide sub-optimal performance in networks with extremely large bandwidth and long delay. The main problems limiting TCP performance in LFNs are as follows. The first problem is the AIMD strategy itself: in congestion avoidance, the cwnd increases linearly by at most one MSS per RTT, while even a single packet loss cuts the cwnd size and, therefore, the sending rate in half. Since the number of RTTs required to restore the previous value of the sending rate depends on the end-to-end path capacity (which is quite large in LFNs), this leads to long periods of underutilization where the sending rate is less than the available bandwidth. The second problem comes from the fact that standard TCP congestion control uses mainly packet drops to infer the state of the network and to estimate the available bandwidth which provides only a rough estimate and does not allow fine-tuning. To overcome these shortcomings, a number of new TCP implementations have been developed, among the most known are TCP Westwood+ [52], HighSpeed TCP [53] and Scalable TCP [54], BIC [55] and CUBIC [56], TCP Vegas [57] and FAST TCP [58]. However, these promising implementations are beyond the scope of the thesis which only aims at analytical modeling of the conventional TCP implementations.

2.3.1 Comparison of TCP Implementations

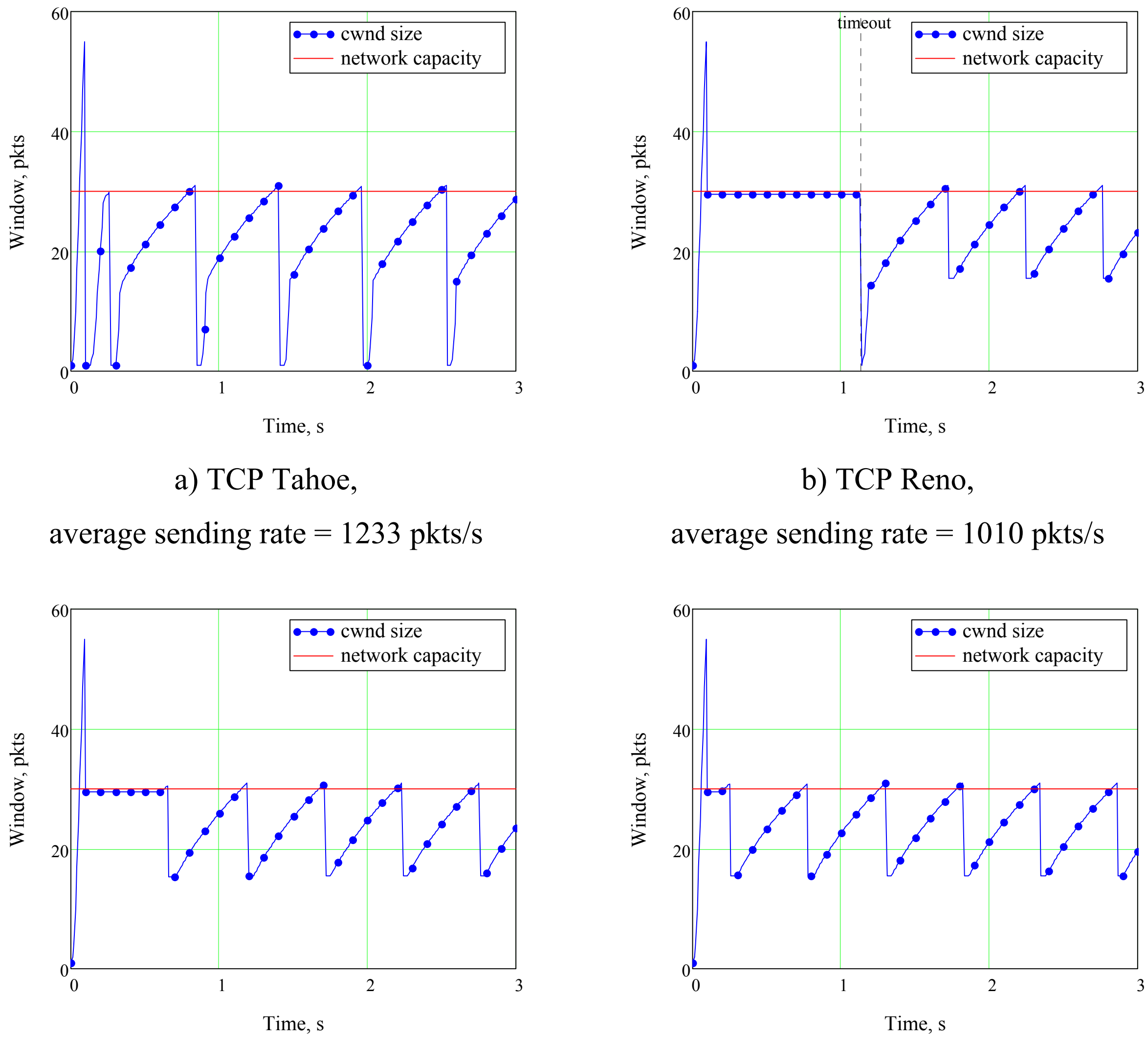
The conventional TCP implementations are summarized in Table 2.2. It is easy to see that they are based on the same congestion control algorithms and differ mainly in fast recovery.

Table 2. 2 Conventional TCP implementations

Congestion control algorithms	TCP Tahoe	TCP Reno	TCP NewReno	TCP SACK
Acknowledgement scheme	Cumulative only	Cumulative only	Cumulative only	Cumulative and selective
Karn's algorithm	Yes	Yes	Yes	Yes
Slow start	Yes	Yes	Yes	Yes
Congestion avoidance	Yes	Yes	Yes	Yes
Fast retransmit	Yes	Yes	Yes	Yes
Fast recovery * NewReno modification ** SACK-based	No	Yes	Yes*	Yes**

Fig. 2.1 illustrates the TCP window evolution obtained using ns-2 [59]. In the ns-2 simulations, the initial value of the ssthresh and the rwnd size were set to be much larger than the maximum number of packets that can be accommodated in the network. During the initial slow start phase, the cwnd grows exponentially from the IW to the ssthresh, so the number of packets injected into the network doubles every RTT (the delayed ACK algorithm is disabled). Eventually, the bottleneck router gets overloaded, resulting in multiple packet drops due to buffer overflow. Every time a segment loss is encountered (regardless of the way it is detected), TCP Tahoe drops the cwnd to one full-sized segment and enters the slow start phase. The fast recovery algorithm, implemented in TCP Reno, NewReno, and SACK, allows the invocation of congestion avoidance instead of slow start after retransmission of a missing segment using the fast retransmit algorithm. This leads to the saw-tooth pattern of AIMD: the cwnd slowly increases (additive increase) and then abruptly is cut in half (multiplicative decrease). As it

follows from Fig. 2.1, TCP Reno, NewReno, and SACK provide similar performance in the presence of uncorrelated packet losses (i.e., when losses are predominantly single packet losses). Everything changes when packet losses occur in bursts (see the initial slow start phase in Fig. 2.1). In particular, TCP Tahoe, NewReno, and SACK significantly outperform TCP Reno under correlated packet losses. This is due to the fact that TCP Reno can recover only one lost segment per invocation of fast retransmit/fast recovery. Three or more losses in the same window usually result in a lengthy timeout [60]. Note that in the ns-2 simulations, the minimum value of the RTO was set to 1 second, since the current standard [45] dictates that whenever the retransmission timeout value is computed, if it is less than 1 second then it should be rounded up to 1 second (which is much larger than the RTT of an average TCP flow).



a) TCP Tahoe,

average sending rate = 1233 pkts/s

b) TCP Reno,

average sending rate = 1010 pkts/s

c) TCP NewReno

average sending rate = 1256 pkts/s

d) TCP SACK

average sending rate = 1259 pkts/s

Fig. 2. 1 TCP window evolution versus the maximum number of packets that can be accommodated in the network, $\mathrm{RTO}_{\min} = 1$ s

2.3.2 Deployment of TCP implementations and TCP options

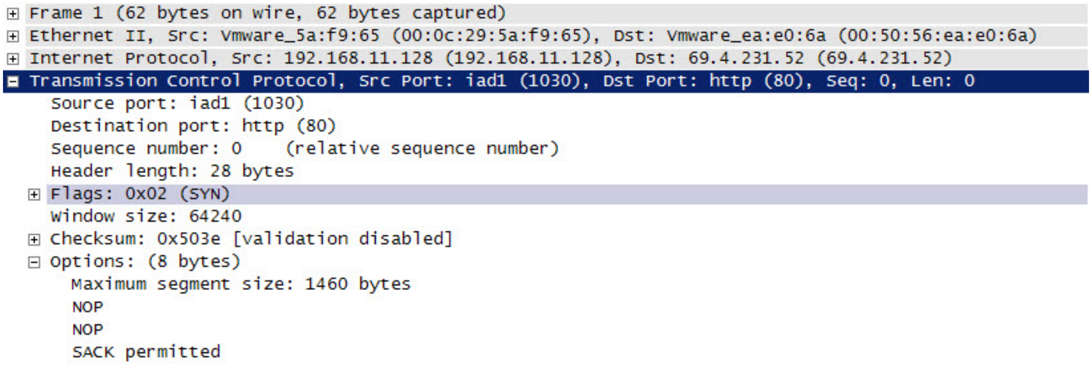
Nowadays, TCP is an integral part of any OS, from desktop to server editions. In [61] [48], the authors explored the prevalence of TCP implementations on Web servers. The measurement results reported in [48] indicate that the fraction of SACK-capable Web servers and the deployment of TCP NewReno have increased significantly in the last few years, while the deployment of TCP Tahoe and TCP Reno has decreased and is limited now to just a few percent.

In this chapter, we investigate the default TCP settings used by modern OSs. According to [62], as for October 2009, the Microsoft Windows family counts for about 90% of all client OSs (out of which Windows XP and Windows Vista are the most popular), whereas the Mac OS X and Linux families constitute approximately 6% and 4%, respectively. The statistics presented in [63], as for October 2009, show that Ubuntu, openSUSE, Fedora, and Mint are the most popular Linux distributions (out of more than 300 distributions listed there). However, since Mint is an Ubuntu-based Linux distribution, we did not consider it separately. Thus, for our study, we used the following OSs:

- Microsoft Windows 2000 Professional SP 4 Build 2195;
- Microsoft Windows XP Professional SP 3 Build 2600;
- Microsoft Windows Server 2003 Standard SP 2 Build 3790;
- Microsoft Windows Vista Home Basic SP 2 Build 6002;
- Microsoft Windows Server 2008 Standard SP 2 Build 6002;
- Microsoft Windows 7 Ultimate Build 7600.16385.090713-1255;
- Ubuntu 9.04 Desktop Edition Kernel 2.6.28-11-generic;
- openSUSE 11.1 Kernel 2.6.27.7-9-pae;
- Fedora 11 Desktop Edition Kernel 2.6.29.4-167.fc11.i686.pae;
- Mac OS X Leopard 10.5.8 Darwin Kernel 9.8.0.

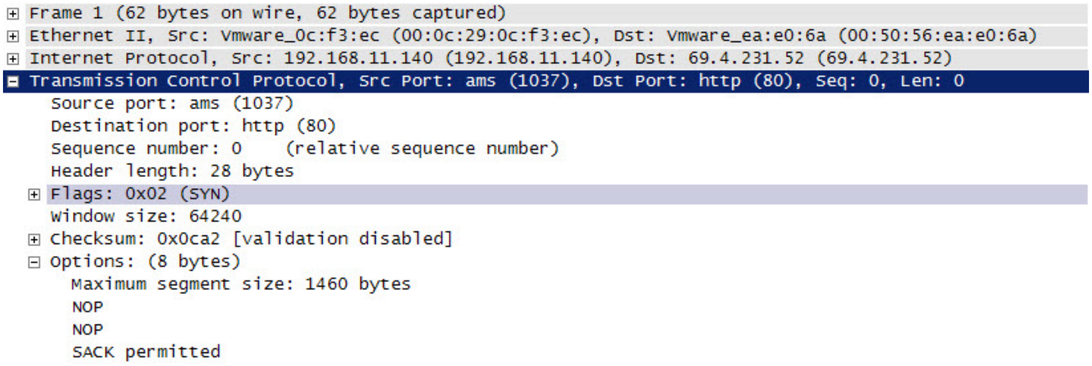
Note that the MSS, window scale, SACK, ECN and other TCP options are sent only in SYN and SYN/ACK segments (i.e., segments with the SYN flag set on). We used Wireshark [64] to capture the packets of interest. The connection establishment segments are shown in Fig. 2.2.

The default TCP settings are summarized in Table 2.3. As it follows from the results obtained, the most popular OSs are SACK-capable. At the same time, while most of them have ECN support, it is not enabled. This is confirmed by recent measurements [48] [13], which show that the fraction of ECN-capable TCP connections is limited to a few percent.



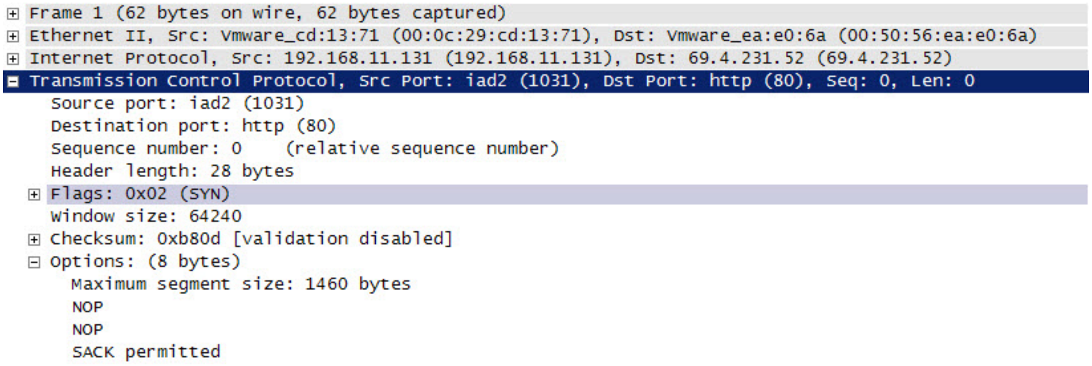
```
⊞ Frame 1 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Vmware_5a:f9:65 (00:0c:29:5a:f9:65), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.128 (192.168.11.128), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: iad1 (1030), Dst Port: http (80), Seq: 0, Len: 0
    Source port: iad1 (1030)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 64240
  ⊞ Checksum: 0x503e [validation disabled]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
```

a) Microsoft Windows 2000 Professional SP 4 Build 2195



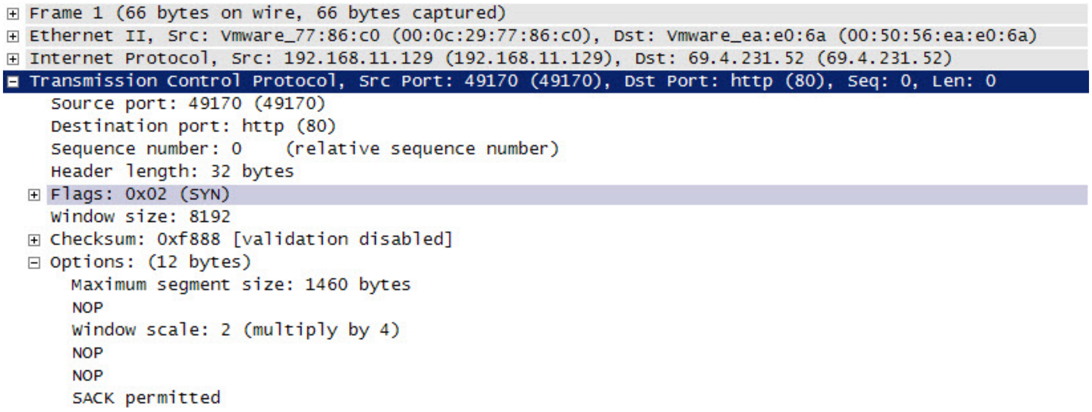
```
⊞ Frame 1 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Vmware_0c:f3:ec (00:0c:29:0c:f3:ec), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.140 (192.168.11.140), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: ams (1037), Dst Port: http (80), Seq: 0, Len: 0
    Source port: ams (1037)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 64240
  ⊞ Checksum: 0x0ca2 [validation disabled]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
```

b) Microsoft Windows XP Professional SP 3 Build 2600



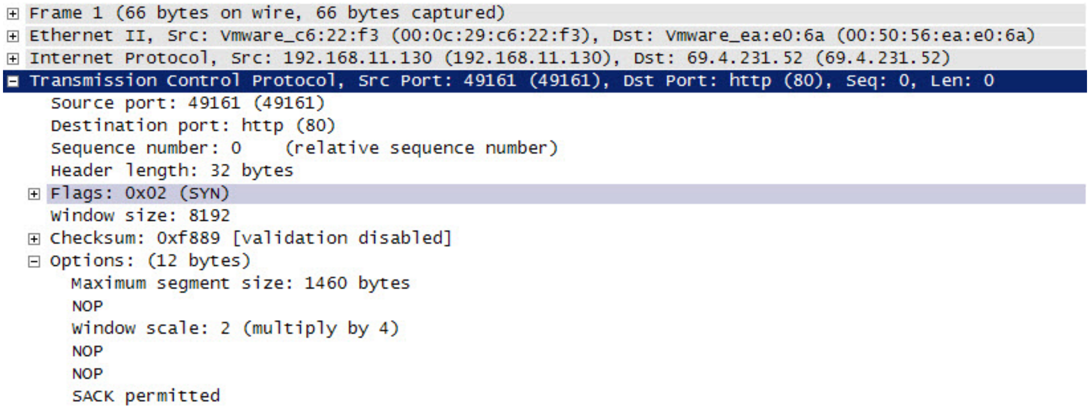
```
⊞ Frame 1 (62 bytes on wire, 62 bytes captured)
⊞ Ethernet II, Src: Vmware_cd:13:71 (00:0c:29:cd:13:71), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.131 (192.168.11.131), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: iad2 (1031), Dst Port: http (80), Seq: 0, Len: 0
    Source port: iad2 (1031)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 28 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 64240
  ⊞ Checksum: 0xb80d [validation disabled]
  ⊟ Options: (8 bytes)
      Maximum segment size: 1460 bytes
      NOP
      NOP
      SACK permitted
```

c) Microsoft Windows Server 2003 Standard SP 2 Build 3790



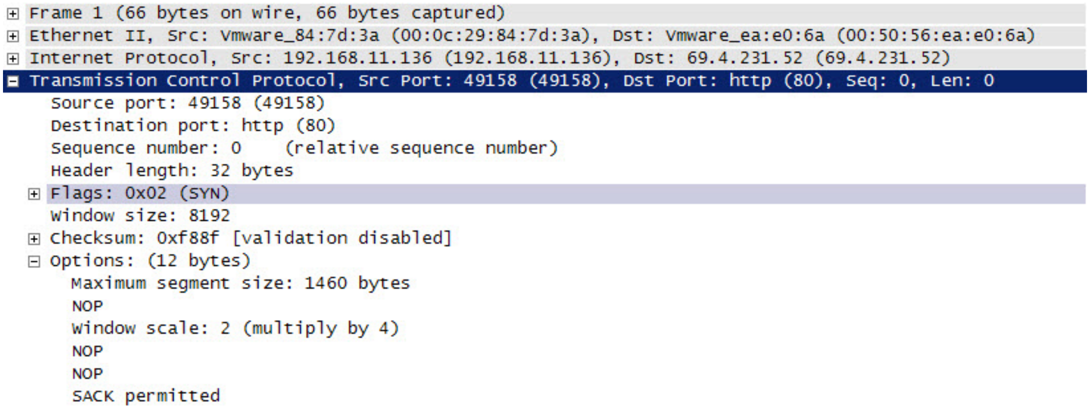
```
⊞ Frame 1 (66 bytes on wire, 66 bytes captured)
⊞ Ethernet II, Src: Vmware_77:86:c0 (00:0c:29:77:86:c0), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.129 (192.168.11.129), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 49170 (49170), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 49170 (49170)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 32 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 8192
  ⊞ Checksum: 0xf888 [validation disabled]
  ⊟ Options: (12 bytes)
      Maximum segment size: 1460 bytes
      NOP
      Window scale: 2 (multiply by 4)
      NOP
      NOP
      SACK permitted
```

d) Microsoft Windows Vista Home Basic SP 2 Build 6002



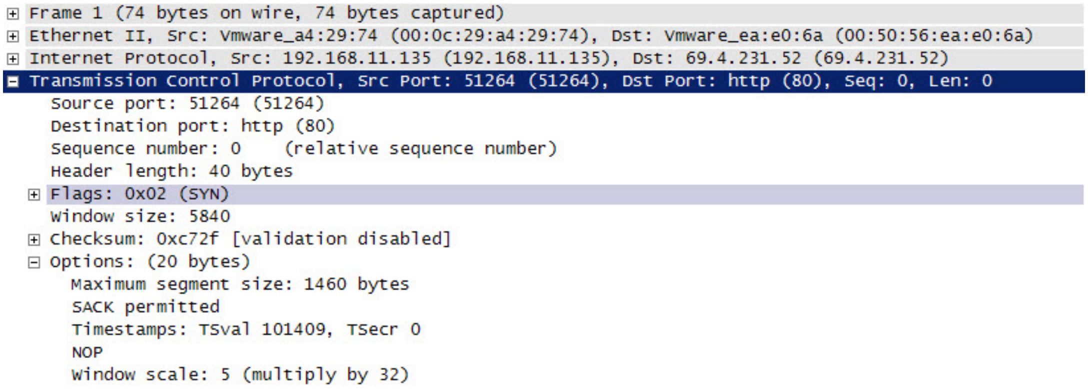
```
⊞ Frame 1 (66 bytes on wire, 66 bytes captured)
⊞ Ethernet II, Src: Vmware_c6:22:f3 (00:0c:29:c6:22:f3), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.130 (192.168.11.130), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 49161 (49161), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 49161 (49161)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 32 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 8192
  ⊞ Checksum: 0xf889 [validation disabled]
  ⊟ Options: (12 bytes)
      Maximum segment size: 1460 bytes
      NOP
      Window scale: 2 (multiply by 4)
      NOP
      NOP
      SACK permitted
```

e) Microsoft Windows Server 2008 Standard SP 2 Build 6002



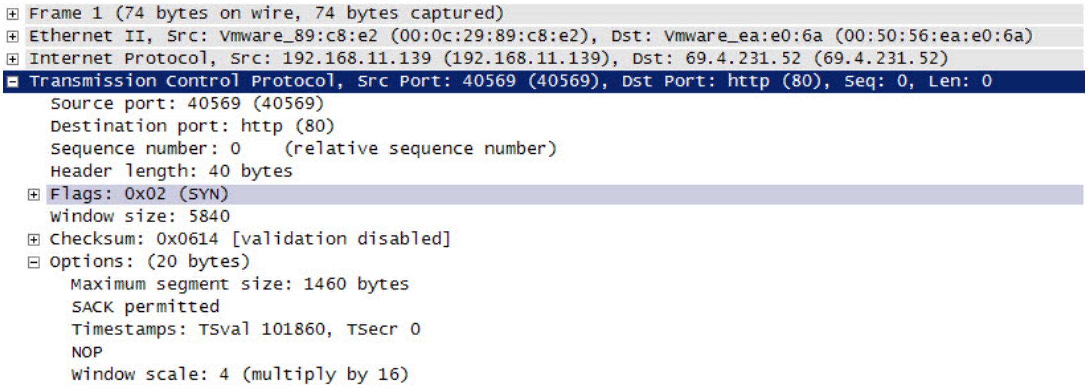
```
⊞ Frame 1 (66 bytes on wire, 66 bytes captured)
⊞ Ethernet II, Src: Vmware_84:7d:3a (00:0c:29:84:7d:3a), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.136 (192.168.11.136), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 49158 (49158), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 49158 (49158)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 32 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 8192
  ⊞ Checksum: 0xf88f [validation disabled]
  ⊟ Options: (12 bytes)
      Maximum segment size: 1460 bytes
      NOP
      Window scale: 2 (multiply by 4)
      NOP
      NOP
      SACK permitted
```

f) Microsoft Windows 7 Ultimate Build 7600.16385.090713-1255



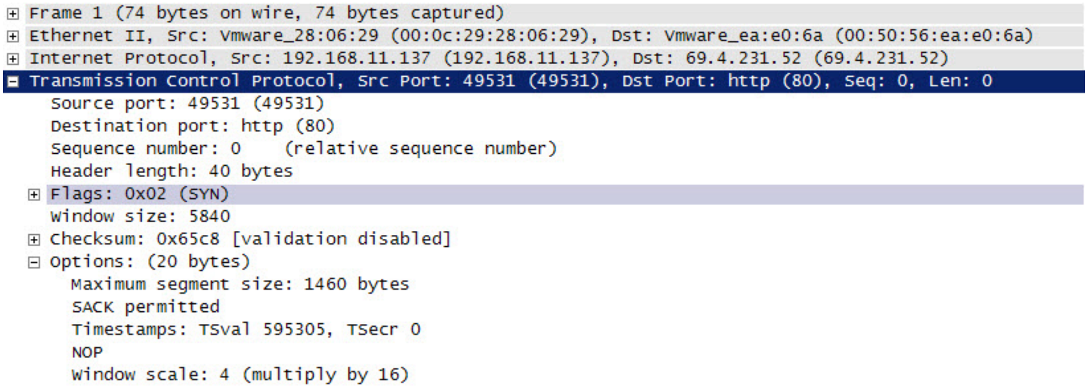
```
⊞ Frame 1 (74 bytes on wire, 74 bytes captured)
⊞ Ethernet II, Src: Vmware_a4:29:74 (00:0c:29:a4:29:74), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.135 (192.168.11.135), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 51264 (51264), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 51264 (51264)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 5840
  ⊞ Checksum: 0xc72f [validation disabled]
  ⊟ Options: (20 bytes)
      Maximum segment size: 1460 bytes
      SACK permitted
      Timestamps: TSval 101409, TSecr 0
      NOP
      Window scale: 5 (multiply by 32)
```

g) Ubuntu 9.04 Desktop Edition Kernel 2.6.28-11-generic



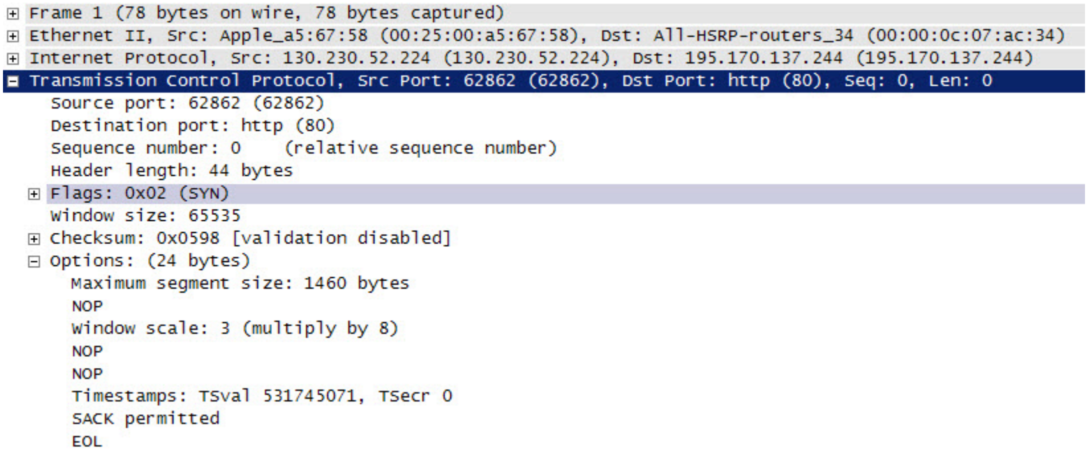
```
⊞ Frame 1 (74 bytes on wire, 74 bytes captured)
⊞ Ethernet II, Src: Vmware_89:c8:e2 (00:0c:29:89:c8:e2), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.139 (192.168.11.139), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 40569 (40569), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 40569 (40569)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 5840
  ⊞ Checksum: 0x0614 [validation disabled]
  ⊟ Options: (20 bytes)
      Maximum segment size: 1460 bytes
      SACK permitted
      Timestamps: TSval 101860, TSecr 0
      NOP
      Window scale: 4 (multiply by 16)
```

h) openSUSE 11.1 Kernel 2.6.27.7-9-pae



```
⊞ Frame 1 (74 bytes on wire, 74 bytes captured)
⊞ Ethernet II, Src: Vmware_28:06:29 (00:0c:29:28:06:29), Dst: Vmware_ea:e0:6a (00:50:56:ea:e0:6a)
⊞ Internet Protocol, Src: 192.168.11.137 (192.168.11.137), Dst: 69.4.231.52 (69.4.231.52)
⊟ Transmission Control Protocol, Src Port: 49531 (49531), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 49531 (49531)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 5840
  ⊞ Checksum: 0x65c8 [validation disabled]
  ⊟ Options: (20 bytes)
      Maximum segment size: 1460 bytes
      SACK permitted
      Timestamps: TSval 595305, TSecr 0
      NOP
      Window scale: 4 (multiply by 16)
```

i) Fedora 11 Desktop Edition Kernel 2.6.29.4-167.fc11.i686.pae



```
⊞ Frame 1 (78 bytes on wire, 78 bytes captured)
⊞ Ethernet II, Src: Apple_a5:67:58 (00:25:00:a5:67:58), Dst: All-HSRP-routers_34 (00:00:0c:07:ac:34)
⊞ Internet Protocol, Src: 130.230.52.224 (130.230.52.224), Dst: 195.170.137.244 (195.170.137.244)
⊟ Transmission Control Protocol, Src Port: 62862 (62862), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 62862 (62862)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 44 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 65535
  ⊞ Checksum: 0x0598 [validation disabled]
  ⊟ Options: (24 bytes)
      Maximum segment size: 1460 bytes
      NOP
      Window scale: 3 (multiply by 8)
      NOP
      NOP
      Timestamps: TSval 531745071, TSecr 0
      SACK permitted
      EOL
```

j) Mac OS X Leopard 10.5.8 Darwin Kernel 9.8.0

Fig. 2. 2 SYN segments

2.4 Conclusions

In this chapter, we considered the TCP congestion control algorithms and explored the default TCP settings used by modern OSs. The results of the study allow us to draw the following conclusions:

- The difference between Reno-based TCP implementations lies mostly in loss detection and recovery and becomes especially clear in the presence of correlated losses. Therefore, special attention should be paid on the fast retransmit and fast recovery algorithms when packet losses occur in bursts.
- According to recent statistics, TCP NewReno and TCP SACK are the most widely used TCP implementations.
- In modern OSs, the SACK and window scale options are enabled by default.
- Modern Windows, Linux, and Mac OSs support ECN but it is disabled by default.

Table 2. 3 Default TCP settings ($\text{MTU} = 1500$ bytes)

Operating system	**IW size (bytes)**	**Delayed ACKs enabled**	**Maximum rwnd (bytes)**	**SACK option enabled**	**Time-stamps enabled**	**ECN support enabled**
Microsoft Windows 2000 Professional SP 4 Build 2195	1460	Yes	$65,535$	Yes	No	No
Microsoft Windows XP Professional SP 3 Build 2600	1460	Yes	$65,535$	Yes	No	No
Microsoft Windows Server 2003 Standard SP 2 Build 3790	4380	Yes	$65,535$	Yes	No	No
Microsoft Windows Vista Home Basic SP 2 Build 6002	2920	Yes	$65,535 \times 2^2 = 262,140$	Yes	No	No
Microsoft Windows Server 2008 Standard SP 2 Build 6002	2920	Yes	$65,535 \times 2^2 = 262,140$	Yes	No	No
Microsoft Windows 7 Ultimate Build 7600.16385.090713-1255	2920	Yes	$65,535 \times 2^2 = 262,140$	Yes	No	No
Ubuntu 9.04 Desktop Edition Kernel 2.6.28-11-generic	1460	Yes	$65,535 \times 2^5 = 2,097,120$	Yes	Yes	No
openSUSE 11.1 Kernel 2.6.27.7-9-pae	1460	Yes	$65,535 \times 2^4 = 1,048,560$	Yes	Yes	No
Fedora 11 Desktop Edition Kernel 2.6.29.4-167.fc11.i686.pae	1460	Yes	$65,535 \times 2^4 = 1,048,560$	Yes	Yes	No
Mac OS X Leopard 10.5.8 Darwin Kernel 9.8.0	1460	Yes	$65,535 \times 2^3 = 524,280$	Yes	Yes	No

3. TCP PERFORMANCE IN WIRED NETWORKS

In this chapter, the developed analytical models for TCP performance evaluation in wired networks are introduced. The chapter also describes the motivation of the research and the main results obtained.

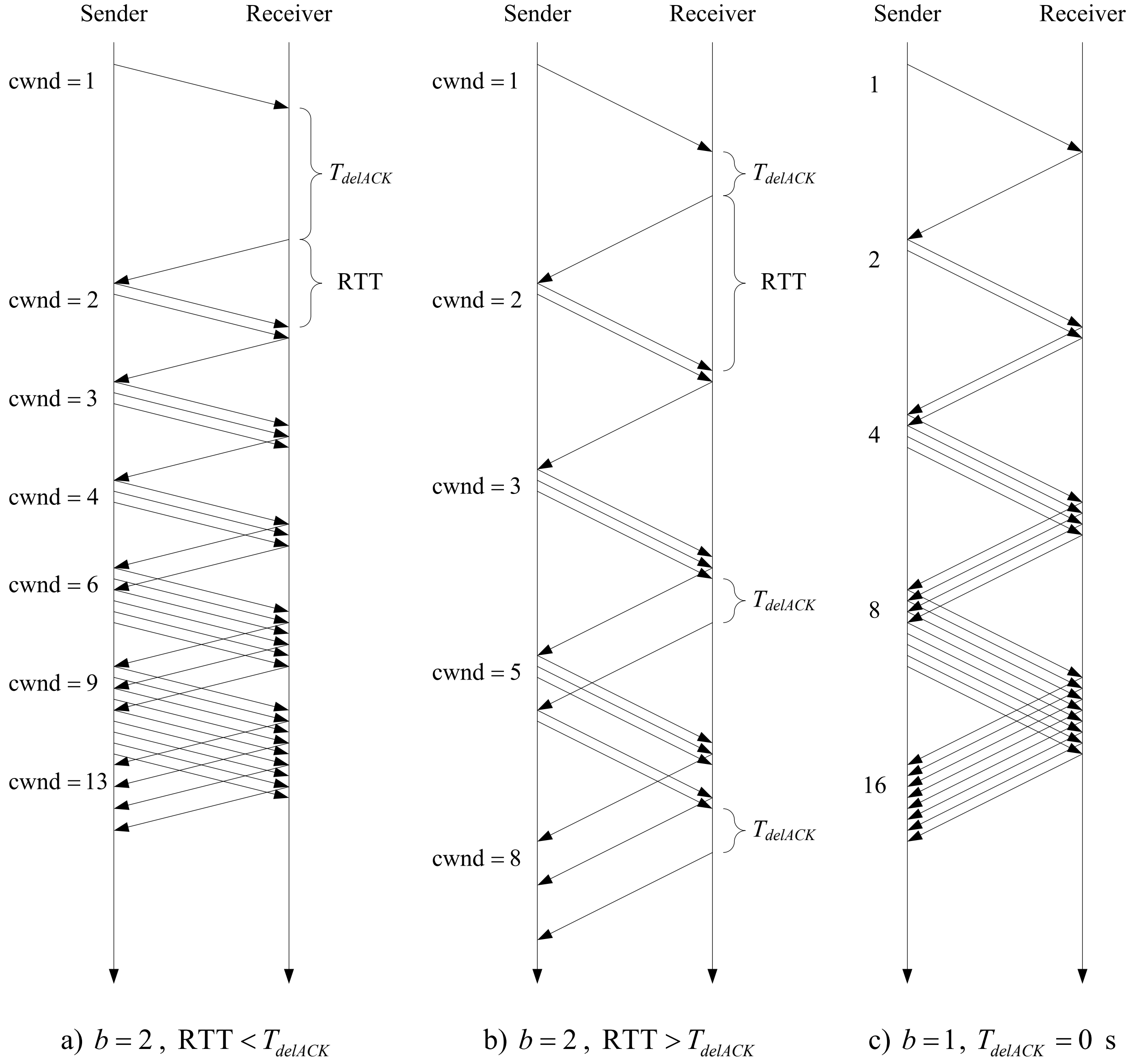
3.1 Short-Lived TCP Flows

Since TCP plays an important role in the Internet, a wide variety of analytical models have been developed for predicting TCP performance under different scenarios. Among the most known and widely referenced are the models presented in [65] [66] [67] [68] [69]. One way to classify TCP analytical models is by the size of the data transfer under consideration [25]. According to numerous measurement studies [70] [71] [72] [13], TCP traffic is dominated by short data transfers. This is also known as the "mice and elephants" phenomenon: much of the traffic on the Internet is carried by a small number of large flows ("elephants"), while most of the flows are short in duration and carry small amounts of data ("mice") [73]. This phenomenon can be considered as invariant, where the term "invariant" refers to an Internet property that has been empirically shown to hold in a very wide range of environments [74]. Since the vast majority of TCP connections are short enough to experience any losses and spend the most part of their lifetime in the initial slow start phase, their performance heavily depends on TCP startup mechanisms: the three-way handshake connection establishment, the IW size, the slow start and delayed ACK algorithms. Consequently, performance modeling and evaluation of short-lived TCP flows has received a lot of attention in the literature (e.g., see [26] and references therein). Moreover, there have been a number of different proposals on improving the startup and/or restart performance of TCP connections [73]. Unfortunately, only one proposal [75], concerning the increase in the permitted upper bound for the IW from one segment to 4380 bytes, has been standardized by the IETF [47] and widely deployed [48].

3.1.1 Motivation and Related Work

The research presented in [P1] was motivated by the fact that different analytical models assume different cwnd increase patterns during the initial slow start phase. Fig. 3.1 presents two examples taken from [76] [77] (Fig. 3.1a and Fig 3.1b, correspondingly). Visual analysis shows that Fig. 3.1a describes the case when the RTT is smaller than the delayed ACK timeout (T_{delACK}), while Fig. 3.1b assumes that the RTT is much bigger than this timeout value, which is usually set to 200 ms [68]. Obviously, the case when the RTT is larger than 200 ms and, hence, RTT $> T_{delACK}$ is more typical for mobile or satellite networks, which are widely deployed today. Therefore, a proper model should capture both cases.

A TCP connection starts with a three-way handshake in which the endpoints exchange their initial sequence numbers. Once the TCP connection has been established, the initial slow start phase begins. In this phase, the TCP sender increases the cwnd by one full-sized segment for every new ACK it receives. Although the TCP receiver can acknowledge every successfully received segment ($b = 1$), as specified in [28], most TCP implementations use the delayed ACK algorithm: the TCP receiver sends one ACK for every two segments that it gets ($b = 2$) or if the delayed ACK timer expires. The slow start phase ends when either the cwnd exceeds the ssthresh or when congestion is observed.



a) $b = 2$, RTT $< T_{delACK}$ b) $b = 2$, RTT $> T_{delACK}$ c) $b = 1$, $T_{delACK} = 0$ s

Fig. 3. 1 Examples of the initial slow start phase, IW $= 1$ full-sized segment

In order to evaluate how the RTT and T_{delACK} ratio affects the cwnd increase pattern, the latency of the initial slow start phase, and the total number of segments sent, we used ns-2 simulations. The rwnd and the ssthresh were set to be high enough not to limit cwnd growth during the initial slow start phase. Our analysis shows that different cwnd increase patterns take place in the following cases:

a) $\text{RTT} \le T_{delACK}$;

b) $\text{RTT} \in \left(T_{delACK}, 2T_{delACK}\right)$;

c) $\text{RTT} = 2T_{delACK}$;

d) $\text{RTT} \in \left(2T_{delACK}, 3T_{delACK}\right]$;

e) $\text{RTT} \in \left(3T_{delACK}, 4T_{delACK}\right]$;

f) $\text{RTT} \in \left(4T_{delACK}, 5T_{delACK}\right]$;

g) $\text{RTT} \in \left(5T_{delACK}, 6T_{delACK}\right]$;

h) etc.

It should be emphasized that the cwnd increase pattern strongly depends on the observed interarrival times and processing delays (i.e., whether the delayed ACK timer expires or not). In this context, case c) can be considered as intermediate. However, we mention it here for the sake of completeness. The obtained results for the first twelve rounds are summarized in Table 3.1 and Table 3.2. It is easy to see that the difference in the cwnd size and in the total number of segments sent rapidly increases with the number of rounds.

As noted in [68], since the TCP receiver sends one ACK for every b-th segment that it receives, the TCP sender will get approximately cwnd/b ACKs every RTT. According to the slow start algorithm, for every new ACK the TCP sender receives, it increases the cwnd by one full-sized segment. Thus, from [68] we have:

$$\text{cwnd}_{i+1} = \text{cwnd}_i + \text{cwnd}_i/b = \gamma\text{cwnd}_i, \quad i = 1, 2, \ldots, \tag{3.1}$$

where γ, $\gamma = 1 + 1/b$, is the rate of exponential growth of the cwnd.

We found that cases a), b), and c) (i.e., when $\text{RTT} \le 2T_{delACK}$) can be closely approximated by (3.1). But if the delayed ACK timer expires before a new segment arrives (as in the cases when $\text{RTT} > 2T_{delACK}$), we get the following cwnd increase pattern:

$$\text{cwnd}_{i+1} = \text{cwnd}_i + \lceil \text{cwnd}_i / b \rceil = \lceil \gamma \text{cwnd}_i \rceil, \quad i = 1, 2, \ldots, \tag{3.2}$$

where $\lceil \ \rceil$ is the ceiling function (i.e., $\lceil x \rceil$ is the smallest integer bigger than or equal to x).

Thus, the examples of the cwnd increase pattern in Fig. 3.1a and Fig. 3.1b can be approximated by (3.1) and (3.2), respectively. Consequently, the comprehensive model can be expressed as

$$\text{cwnd}_{i+1} = \begin{cases} \gamma \text{cwnd}_i, & \text{RTT} \le 2T_{delACK}, \\ \lceil \gamma \text{cwnd}_i \rceil, & \text{RTT} > 2T_{delACK}. \end{cases} \tag{3.3}$$

It should be emphasized that when the TCP receiver does not use the delayed ACK algorithm (see Fig. 3.1c), $\gamma = 2$ and the cwnd increase pattern can be modeled just as (3.1).

Table 3. 1 cwnd increase patterns

Round, i	**Size of the cwnd (in full-sized segments) at the end of round i, cwnd_i**						
	a)	**b)**	**c)**	**d)**	**e)**	**f)**	**g)**
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	5	5	5	5	5	5
5	6	7	7	8	8	8	8
6	9	10	11	12	12	12	12
7	13	15	16	18	18	18	18
8	19	22	24	27	27	27	27
9	28	33	36	40	41	41	41
10	42	49	54	60	61	62	62
11	63	73	81	90	91	93	93
12	94	109	121	135	136	139	140

Table 3. 2 Total number of segments sent

Number of rounds, *n*	**Total number of segments sent during *n* rounds, ssdata_n**						
	a)	**b)**	**c)**	**d)**	**e)**	**f)**	**g)**
1	1	1	1	1	1	1	1
2	3	3	3	3	3	3	3
3	6	6	6	6	6	6	6
4	9	11	11	11	11	11	11
5	15	17	17	19	19	19	19
6	24	26	28	31	31	31	31
7	36	41	43	49	49	49	49
8	54	62	67	76	76	76	76
9	81	95	103	115	117	117	117
10	123	143	157	175	177	179	179
11	186	215	238	265	267	272	272
12	279	323	358	400	402	410	412

A number of models have been developed for the analysis of TCP startup behavior. In [68], Cardwell *et al*. proposed to approximate the cwnd increase pattern as (3.1) and to compute the total number of segments sent during the slow start phase as the sum of a geometric series. Sikdar *et al*. noted that the cwnd increase pattern varies in practice and can make significant differences for short-lived flows. In order to account such complex behavior, the model presented in [69] uses an averaged pattern from (3.3). Finally, Zheng *et al*. proposed to model (3.2) as the Fibonacci sequence [78]. Table 3.3 summarizes the resultant expressions.

Table 3. 3 Analytical models for short-lived TCP flows

Output parameter	Cardwell *et al.* [68]	Sikdar *et al.* [69]	Zheng *et al.* [78]
Size of the cwnd at the end of round n, cwnd_n	$\gamma \mathrm{cwnd}_{n-1}$, $\gamma = 1 + \frac{1}{b}, \quad \mathrm{cwnd}_1 = \mathrm{IW}$	$\left\lfloor 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} \right\rfloor$, $\lfloor \; \rfloor$ is the floor function	$\mathrm{cwnd}_n = C_1 X_1^n + C_2 X_2^n$, $X_{1,2} = \frac{1 \pm \sqrt{5}}{2}$, $C_1 + C_2 = 1$
Number of segments sent by the end of round n, $\mathrm{ssdata}_n = \sum_{k=1}^{n} \mathrm{cwnd}_k$	$\mathrm{cwnd}_1 \frac{\gamma^n - 1}{\gamma - 1}$, $\mathrm{cwnd}_1 = \mathrm{IW}$	$\left\lfloor 2^{\frac{n+1}{2}} + 3\left(2^{\frac{4n-3}{8}}\right) - 2 - \frac{3}{\sqrt{2}} \right\rfloor$	$C_1 X_1^{n+2} + C_2 X_2^{n+2} - 2 \approx$ $\approx C_1 X_1^{n+2} - 2$
Number of rounds to transfer ssdata_n segments	$\log_\gamma \left(\frac{\mathrm{ssdata}_n (\gamma - 1)}{\mathrm{cwnd}_1} + 1 \right)$, $\mathrm{cwnd}_1 = \mathrm{IW}$	$\left\lceil 2 \log_2 \left(\frac{2\mathrm{ssdata}_n + 4 + 3\sqrt{2}}{2\sqrt{2} + 3\left(2^{\frac{5}{8}}\right)} \right) \right\rceil$	$\log_{X_1} \left(\frac{\mathrm{ssdata}_n + 2}{C_1} \right) - 2$

3.1.2 Model Building

The model developed in [P1] uses exactly the same assumptions about the hosts and the underlying network as the models presented in [68] [69] [78]. Namely, we model TCP behavior in terms of "rounds", where a round begins when the TCP sender starts transmitting a window of segments and ends when it receives the first ACK for one or more of these segments. It is assumed that the TCP receiver uses the delayed ACK algorithm as specified in [28]. Similarly to [68] [69] [78], we do not take into account the effects of the Nagle and SWS avoidance algorithms. Instead, we assume that the TCP sender transmits full-sized segments whenever the cwnd size allows, while the rwnd size is assumed to be large enough not to limit the sending rate (i.e., at any moment in time, $\mathrm{cwnd} \le \mathrm{rwnd}$). We also assume that the time needed to send a window of segments is smaller than the RTT, which is supposed to be independent of the transmission window size.

Since the model proposed in [68] approximates well the initial slow start phase when $\mathrm{RTT} \le 2T_{delACK}$, the idea behind this study is to extend it for the case when $\mathrm{RTT} > 2T_{delACK}$. Note that the ceiling function can be represented as

$$\lceil x \rceil = x + r, \quad 0 \le r < 1. \tag{3.4}$$

Thus, we can define the cwnd increase pattern from (3.2) as

$$\begin{aligned} &\text{cwnd}_1 = \text{IW}, \\ &\text{cwnd}_2 = \text{IW} + \lceil \text{IW}/b \rceil = \lceil \text{IW}\gamma \rceil = \text{IW}\gamma + r_1, \\ &\text{cwnd}_3 = \lceil \gamma(\text{IW}\gamma + r_1) \rceil = \text{IW}\gamma^2 + r_1\gamma + r_2, \\ &\ldots \\ &\text{cwnd}_n = \text{IW}\gamma^{n-1} + r_1\gamma^{n-2} + \ldots + r_{n-2}\gamma + r_{n-1}. \end{aligned} \tag{3.5}$$

Since $b = 2$ and $\gamma = 1.5$, we have that the variable r_j, $j = 1,2,\ldots,n-1$, is a discrete variable, which takes on a value of 0 or 0.5. As it follows from Table 3.4, r_j is almost uniformly distributed between these values. Then we use the following approximations:

$$E[r] = \frac{0+0.5}{2} = \frac{1}{4} \tag{3.6}$$

and

$$\text{cwnd}_{i+1} = \lceil \gamma \text{cwnd}_i \rceil \approx \gamma \text{cwnd}_i + E[r]. \tag{3.7}$$

Table 3. 4 cwnd increase pattern from (3.2), $\text{IW} = 1$ full-sized segment

	Round, *i*											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
cwnd_i	1	2	3	5	8	12	18	27	41	62	93	140
r_j	No	0.5	0	0.5	0.5	0	0	0	0.5	0.5	0	0.5

Hence, the number of segments sent by the end of round n can be found as

$$\begin{aligned} &\text{ssdata}_n = \text{IW} + (\text{IW}\gamma + E[r]) + (\text{IW}\gamma^2 + E[r]\gamma + E[r]) + \ldots + \\ &+(\text{IW}\gamma^{n-1} + E[r]\gamma^{n-2} + \ldots + E[r]\gamma + E[r]) = \text{IW}\sum_{k=1}^{n}\gamma^{k-1} + E[r]\sum_{k=1}^{n}(n-k)\gamma^{k-1}. \end{aligned} \tag{3.8}$$

To find the number of rounds required to transfer ssdata_n segments (i.e., the value of n), we transform expression (3.8) to the canonical form:

$$n - \left(\frac{\text{IW}(\gamma-1) + E[r]}{E[r](\gamma-1)}\right)\gamma^n + \frac{\text{ssdata}_n(\gamma-1)^2 + \text{IW}(\gamma-1) + E[r]}{E[r](\gamma-1)} = 0. \tag{3.9}$$

The obtained equation is a transcendental equation, which usually cannot be solved by an exact method, only by a numerical approach. In [P1], we used a new method of finding solutions to transcendental equations proposed in [79]. Appendix A provides a brief introduction to this method. Particularly, let (3.10) be a transcendental equation:

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \ldots + a_{n-d} x^d + a_{n-q} x^q + f(x) + a_n = 0, \tag{3.10}$$

where $f(x)$ is a transcendental function.

According to [79], the equation determinant is given by

$$p = \frac{m^d}{m^q + \dfrac{a_n + f(m)}{a_{n-q}}}, \quad p = \frac{-a_{n-q}}{m^{n-d} + a_1 m^{n-(d+1)} + a_2 m^{n-(d+2)} + \ldots + a_{n-d}}. \tag{3.11}$$

The range of values of m and, consequently, the approximate real roots of the equation can be found from the set of inequalities for $p > 0$ and for $p < 0$. Taking into account that n is positive by definition and, hence, $m > 0$, we arrive at:

$$p < 0 \qquad \begin{cases} m + \dfrac{\mathrm{ssdata}_m (\gamma-1)^2 + \mathrm{IW}(\gamma-1) + E[r]}{E[r](\gamma-1)} > 0, \\ 1 - \dfrac{\left(\mathrm{IW}(\gamma-1) + E[r]\right)\gamma^m}{\mathrm{ssdata}_m (\gamma-1)^2 + \mathrm{IW}(\gamma-1) + E[r]} < 0, \\ m > 0. \end{cases} \tag{3.12}$$

Thus, we have:

$$m > \log_\gamma \left(\frac{\mathrm{ssdata}_m (\gamma-1)^2}{\mathrm{IW}(\gamma-1) + E[r]} + 1 \right). \tag{3.13}$$

As it follows from Table 3.5, expression (3.14) is a fairly good approximation of the number of rounds required to transfer ssdata_n segments:

$$n \approx \log_\gamma \left(\frac{\mathrm{ssdata}_n (\gamma-1)^2}{\mathrm{IW}(\gamma-1) + E[r]} + 1 \right). \tag{3.14}$$

Finally, from (3.5) and (3.6) we obtain the cwnd size at the end of round n:

$$\mathrm{cwnd}_n = \mathrm{IW}\gamma^{n-1} + E[r]\sum_{k=1}^{n-1} \gamma^{k-1} = \mathrm{IW}\gamma^{n-1} + E[r]\frac{\gamma^{n-1} - 1}{\gamma - 1}. \tag{3.15}$$

Table 3. 5 Approximate analytical solution for the number slow start rounds

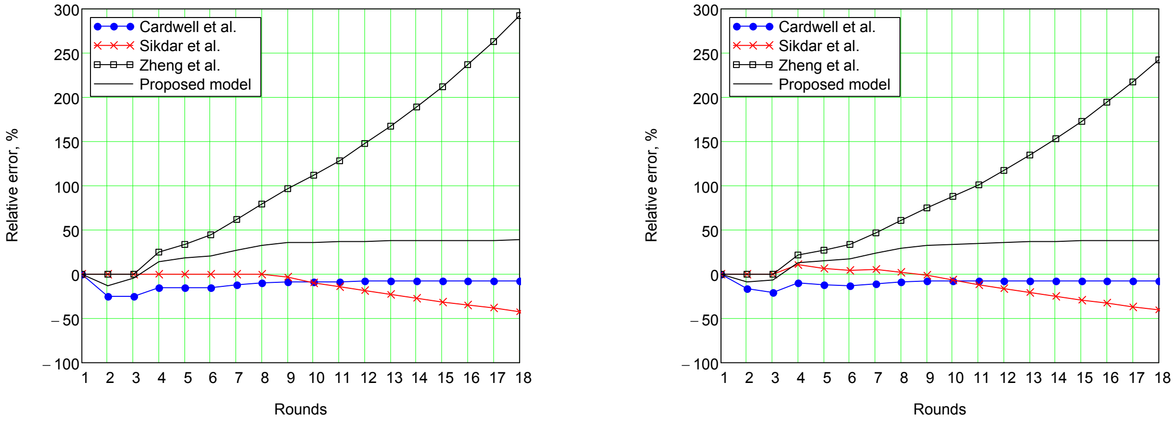
Round, *i*	**IW = 1**			**IW = 2**			**IW = 3**		
	$\text{cwnd}_i = \lceil \gamma \text{cwnd}_{i-1} \rceil$	$\text{ssdata}_i = \sum_{k=1}^{i} \text{cwnd}_k$	n as in (3.14)	$\text{cwnd}_i = \lceil \gamma \text{cwnd}_{i-1} \rceil$	$\text{ssdata}_i = \sum_{k=1}^{i} \text{cwnd}_k$	n as in (3.14)	$\text{cwnd}_i = \lceil \gamma \text{cwnd}_{i-1} \rceil$	$\text{ssdata}_i = \sum_{k=1}^{i} \text{cwnd}_k$	n as in (3.14)
1	1	1	**0.71**	2	2	**0.83**	3	3	**0.88**
2	2	3	**1.71**	3	5	**1.71**	5	8	**1.88**
3	3	6	**2.71**	5	10	**2.71**	8	16	**2.93**
4	5	11	**3.8**	8	18	**3.76**	12	28	**3.97**
5	8	19	**4.91**	12	30	**4.80**	18	46	**4.99**
6	12	31	**5.99**	18	48	**5.82**	27	73	**6.01**
7	18	49	**7.04**	27	75	**6.84**	41	114	**7.03**
8	27	76	**8.07**	41	116	**7.86**	62	176	**8.05**
9	41	117	**9.1**	62	178	**8.88**	93	269	**9.06**
10	62	179	**10.13**	93	271	**9.89**	140	409	**10.7**
11	93	272	**11.14**	140	411	**10.9**	210	619	**11.08**
12	140	412	**12.16**	210	621	**11.91**	315	934	**12.09**
13	210	622	**13.17**	315	936	**12.92**	473	1407	**13.09**
14	315	937	**14.17**	473	1409	**13.92**	710	2117	**14.10**
15	473	1410	**15.18**	710	2119	**14.93**	1065	3182	**15.10**
16	710	2120	**16.18**	1065	3184	**15.93**	1598	4780	**16.10**
17	1065	3185	**17.19**	1598	4782	**16.93**	2397	7177	**17.10**
18	1598	4783	**18.19**	2397	7179	**17.93**	3596	10773	**18.10**

3.1.3 Model Validation and Conclusions

In order to validate the proposed model and compare it with the models presented in [68] [69] [78], we compare the results obtained from the analytical models against the simulation results obtained using ns-2. The ns-2 trials consisted of the cross-product of $\mathrm{IW} = \{1, 2, 3\}$ full-sized segments, $T_{delACK} = \{100, 150, 200\}$ ms, and $\mathrm{RTT} \le NT_{delACK}$, $N = 1, 2, \ldots, 6$. To estimate the accuracy in prediction of the cwnd increase pattern and the number of segments sent during the initial slow start phase, we computed the relative error at round i using the following expression:

$$\varepsilon(i) = \frac{\left(X(i)_{\text{predicted}} - X(i)_{\text{observed}}\right)}{X(i)_{\text{observed}}} 100\%, \quad i = 1, 2, \ldots. \tag{3.16}$$

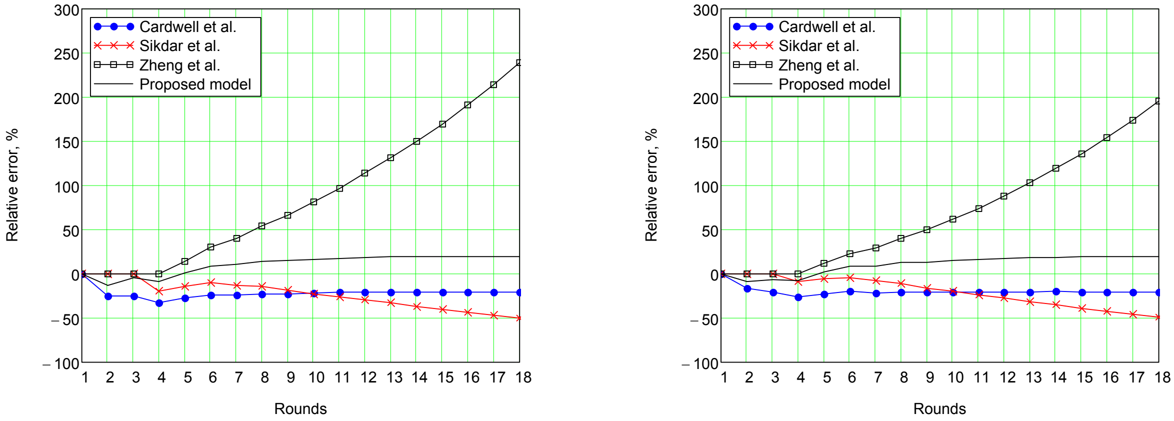
Hence, a model overestimates an actual value when $\varepsilon > 0$ and underestimates it when $\varepsilon < 0$.



a) The cwnd increase pattern

b) The number of segments sent

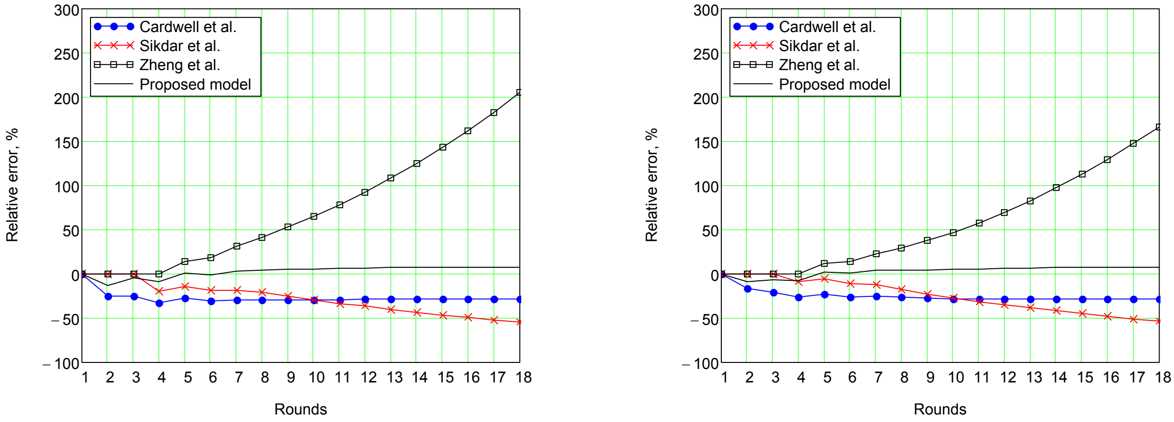
Fig. 3. 2 Initial slow start phase, $\mathrm{RTT} \le T_{delACK}$



a) The cwnd increase pattern

b) The number of segments sent

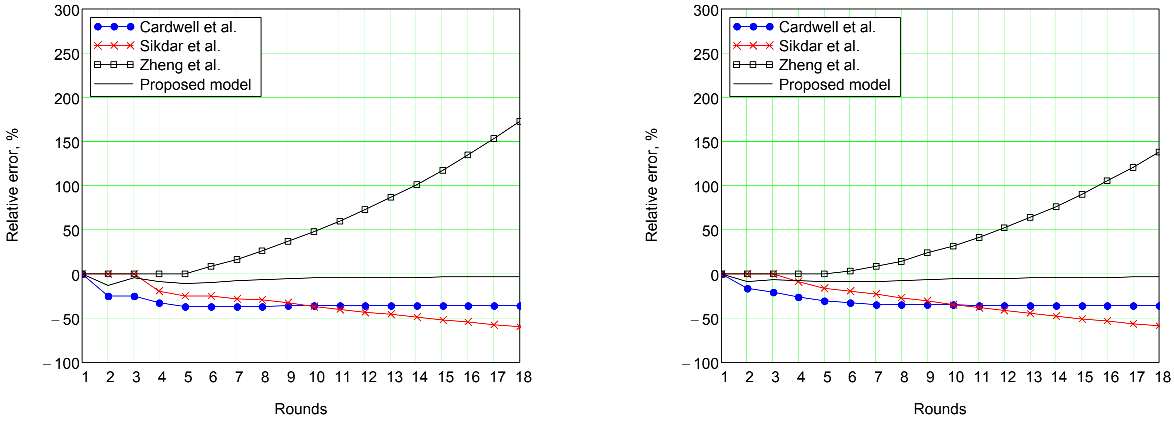
Fig. 3. 3 Initial slow start phase, $\mathrm{RTT} \in (T_{delACK}, 2T_{delACK})$



a) The cwnd increase pattern

b) The number of segments sent

Fig. 3. 4 Initial slow start phase, $\mathrm{RTT} = 2T_{delACK}$



a) The cwnd increase pattern

b) The number of segments sent

Fig. 3. 5 Initial slow start phase, $\mathrm{RTT} \in \left(2T_{delACK}, 3T_{delACK}\right]$

For the sake of briefness, we present the results only for $\mathrm{IW} = 1$ full-sized segment and cases a), b), c), d) (Fig. 3.2, Fig. 3.3, Fig. 3.4, Fig. 3.5, correspondingly). The results for the other two values of the IW and cases e), f), g) are qualitatively similar (see [P1] for details).

The obtained results lead to the following conclusions.

- All the examined models fail to predict the whole range of cases with different RTT and T_{delACK} ratios. For instance, the model proposed by Zheng *et al.* [78] is more suitable for the cases when $\mathrm{RTT} > T_{delACK}$. Within the first rounds, this model captures the cwnd increase pattern and the total number of transmitted segments without any error. Unfortunately, with the increase in the number of rounds, it greatly overestimates these parameters and introduces a significant error. This is due to the fact that the cwnd increase pattern can be represented by the Fibonacci sequence only in the first rounds (i.e., $\mathrm{cwnd} = 1, 2, 3, 5, 8$). After that, the approximation error grows exponentially. Although the model developed by Sikdar *et al.* [69] is intended to capture all possible cases with and without delayed ACK timeout expirations, it gives the best results only for the case when $\mathrm{RTT} \le T_{delACK}$ and $\mathrm{IW} = 1$ (see [P1] for details). Since the proposed model was derived for the case when $\mathrm{RTT} > 2T_{delACK}$, it leads to a significant overestimation for the opposite case, but when $\mathrm{RTT} > 2T_{delACK}$, it outperforms all the examined models.

- As the developed model performs well for the case when $\mathrm{RTT} > 2T_{delACK}$, to obtain a comprehensive model, we only need to select an appropriate model for the case when $\mathrm{RTT} \le 2T_{delACK}$. In addition to acceptable accuracy over a wide range of rounds and different values of the IW size (see [P1] for details), the model proposed by Cardwell *et al.* [68] has a solid theoretical background and can be easily extended to capture the case when the TCP receiver does not use the delayed ACK algorithm. Thus, combining the results from [68] with the developed model, we get the comprehensive analytical model for short-lived TCP flows, which takes into account the impact of different RTT and T_{delACK} ratios on the initial slow start phase.

3.2 TCP Reno Performance and the PFTK-model

There are various metrics used to characterize the performance of TCP implementations: bandwidth utilization, fairness, responsiveness, aggressiveness, etc. Data throughput, as an assessment of the amount of data that can be transmitted per unit of time, is an important metric for quantifying TCP performance. Since TCP throughput is a mean (rather than instantaneous) parameter, this value is averaged over a long time (sometimes considered infinity). Therefore, this performance metric is closely coupled with long-lived TCP flows.

One of the best known analytical models for evaluating the throughput of a TCP connection is the model proposed by Padhye *et al.* in [67], also known as the PFTK-model (named after the authors' surnames: Padhye, Firoiu, Towsley, Kurose). This model is widely referenced: entering the paper title in the Scientific Literature Digital Library search engine gives several hundred citations [80]. The PFTK-model defines the steady-state throughput of a long-lived TCP Reno bulk data transfer as a function of the loss event rate, the mean RTT, the expected duration of a timeout, and the maximum size of the rwnd. It assumes a correlated (bursty) loss model that is better suited for First-In, First-Out (FIFO) Drop-Tail queue management, which is still widely used [81] [82, p.51]. Unfortunately, this model uses an oversimplified representation of fast retransmit/fast recovery dynamics as having (supposedly) negligible effect on TCP Reno throughput. As it will be shown later, this simplification results in overestimation of TCP Reno throughput in the presence of correlated losses. Since new analytical models are often compared with the PFTK-model and use its resultant formula, such inaccuracy in throughput estimation can lead to erroneous results and/or incorrect conclusions. In [P2], the PFTK-model was revised to make it more consistent with actual TCP Reno behavior when segment losses occur in bursts.

3.2.1 Motivation and Model Building

The refined model we develop is based on the same assumptions about the hosts and the underlying network as the PFTK-model. We assume that the sending host uses the TCP Reno implementation [29] and always has data to send. Since we are focusing on TCP performance, we do not consider sender-side or receiver-side delays and limitations due to scheduling or buffering. Therefore, we assume that the TCP sender sends full-sized segments whenever the cwnd allows, while the rwnd is assumed to be always constant. We model TCP behavior in terms of “rounds” as it was done in [67] and in the previous section. Considering the data transfer, we assume that segment losses happen only in the direction from the TCP sender to the TCP receiver. Moreover, we assume that a segment is lost in a round independently of any segments lost in other rounds, while segment losses are correlated within a round (i.e., if a segment is lost, all the remaining segments in that round are also lost). This bursty loss model is a simplified representation of the packet loss process in FIFO Drop-Tail routers. We assume that the time needed to send a window of segments is smaller than the duration of a round. It is also assumed that the probability of a segment loss and the duration of a round are independent of the window size. The latter assumption is justified in case of a high level of statistical multiplexing.

According to [29], a segment loss can be detected in one of the two ways: either by reception of three duplicate ACKs or via retransmission timer expiration. Similarly to [67], let us denote the first event as a TD (Triple-Duplicate) loss indication and the second as a TO (TimeOut) loss indication. As in [67], we develop our model in three steps:

- when the first loss indication in a cycle is exclusively TD and the cwnd is always smaller than the rwnd;
- when the first loss indication in a cycle is either TD or TO and the cwnd is always smaller than the rwnd;
- when the first loss indication in a cycle is either TD or TO and the sending rate is limited by the rwnd.

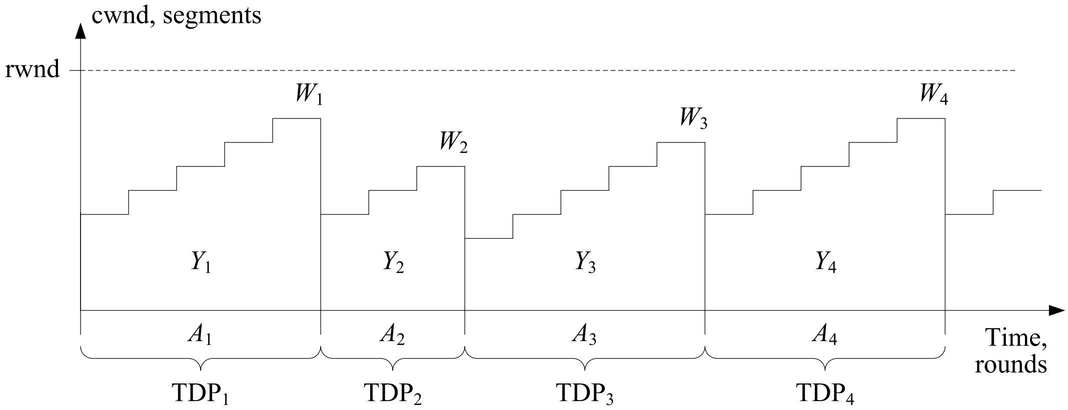


Fig. 3. 6 TCP Reno window evolution in the absence of timeouts (as assumed in [67])

First of all, let us investigate the case when the first loss in a cycle is detected exclusively via a TD loss indication, while the sending rate is not limited by the rwnd (see Fig. 3.6). Considering the cyclic evolution of the cwnd size over time, let Y_i be the number of segments sent during the i-th cycle, $i = 1, 2, \ldots$, A_i be the duration of the cycle in rounds, and W_i be the window size at the end of the cycle. Let $\{W_t\}$, $t > 0$, be a regenerative process with a renewal reward process $\{Y_t\}$, and the moments of time when a segment loss is detected be regenerative points. Then we can define the TCP long-term steady-state throughput as

$$B = \lim_{t\to\infty} B_t = \lim_{t\to\infty} \frac{N_t}{t} = \frac{E[Y]}{E[A]}, \tag{3.17}$$

where B_t is the sending rate in the interval $[0, t]$; N_t is the number of segments sent in this interval; $E[Y]$ is the expected number of segments sent during a cycle; $E[A]$ is the expected duration of a cycle in rounds.

Note that in [67] a cycle is defined as a period of time between two TD loss indications and is denoted as a TD Period (TDP). Fig. 3.7 shows the i-th TDP according to [67]. The TDP starts immediately after the TD loss indication, so the current value of the cwnd (expressed in full-sized segments) is set to $W_{i-1}/2$. The TCP receiver sends one ACK for every b-th segment that it receives (in Fig. 3.7 and Fig. 3.8, $b = 2$), so the cwnd increases linearly with a slope of $1/b$ segments per round until the first segment loss occurs. Let us denote by α_i the first lost segment in the i-th cycle and by X_i the round where this loss occurs (see Fig. 3.7). According to the sliding window algorithm, after the segment α_i, $W_i - 1$ more segments are sent before the loss is detected at the TCP sender and the current TDP ends.

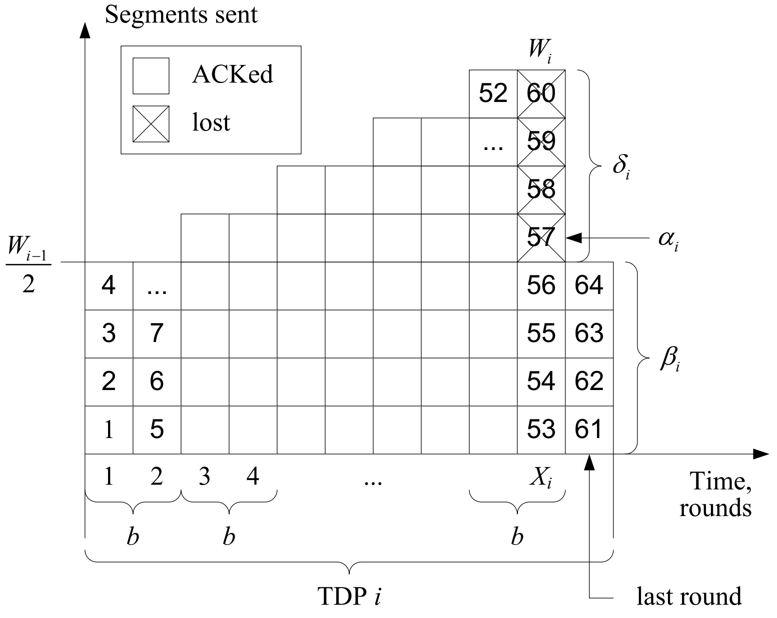


Fig. 3. 7 TDP (as assumed in [67])

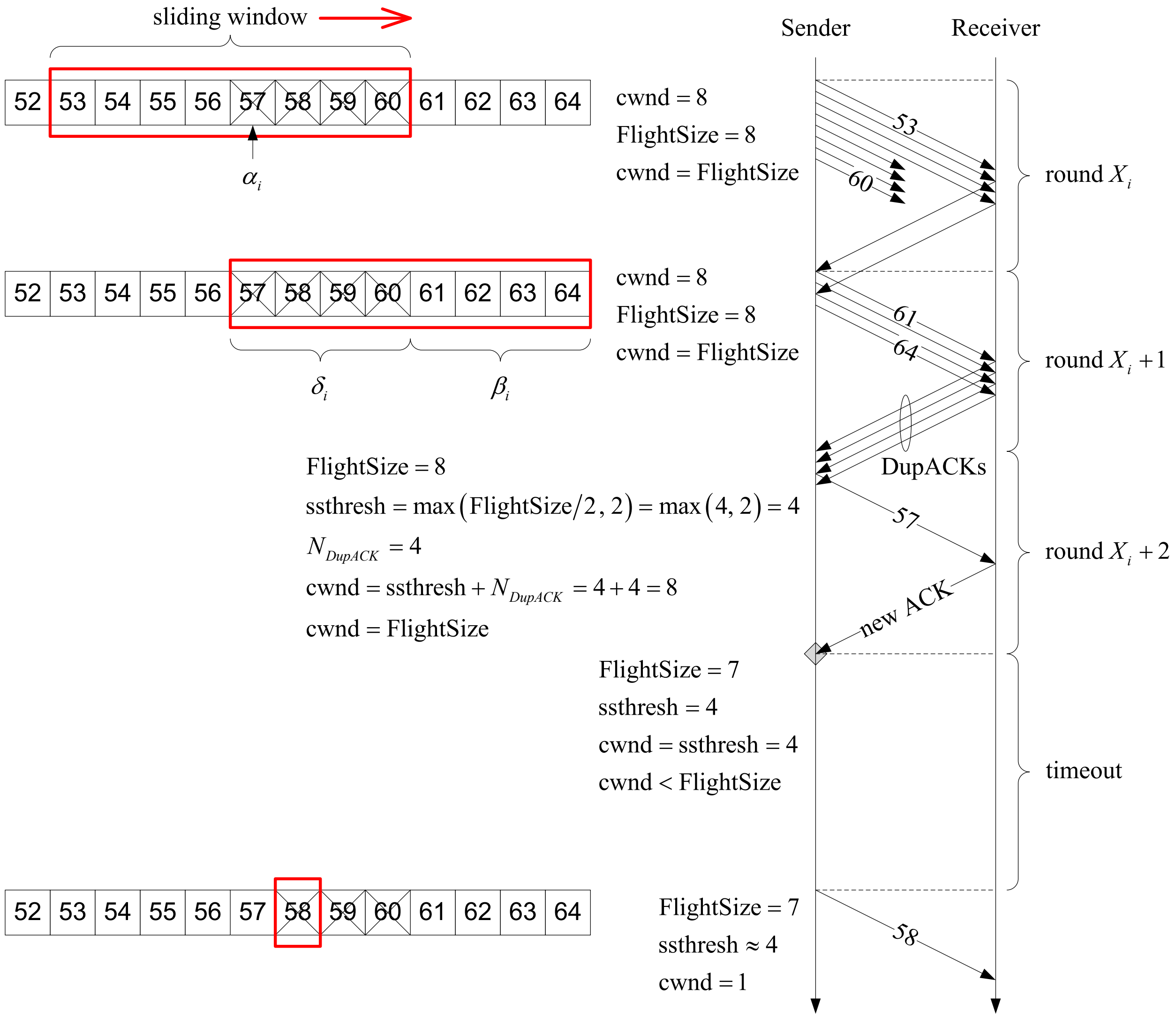


Fig. 3. 8 Transmission of segments in the last rounds of the i-th TDP

Let us consider the evolution of the cwnd size in the i-th cycle after the detection of the first loss (see Fig. 3.8). Taking into account the assumption about correlated losses within a round (i.e., if a segment is lost, so are all the following segments till the end of the round), all segments following α_i in the round X_i are lost as well. Let us define δ_i to be the number of segments lost in the round X_i and β_i to be the number of segments sent in the next round X_i+1 of the i-th cycle. In [67], it is assumed that the random variable β_i is uniformly distributed from zero to W_i-1 segments. Thus, taking into account that $\beta_i = W_i - \delta_i$ (see Fig. 3.8), we can use the following approximation:

$$E[\beta] = \frac{E[W]-1}{2} \approx \frac{E[W]}{2}, \quad E[\delta] = \frac{E[W]+1}{2} \approx \frac{E[W]}{2}. \tag{3.18}$$

After a TD loss indication, the TCP sender enters the fast retransmit/fast recovery phase and performs a retransmission of what appears to be the missing segment. The ssthresh and the current value of the cwnd are updated according to [29] as

$$\text{ssthresh} = \max\left(\text{FlightSize}/2,\, 2\right), \quad W' = \text{ssthresh} + N_{DupACK}, \tag{3.19}$$

where FlightSize is the number of segments that have been sent, but not yet acknowledged; W' is the value of the cwnd during the fast recovery phase; N_{DupACK} is the number of received duplicate ACKs.

Since we assume (as was done in [67]) that the number of lost segments per loss event is approximately equal to half of the segments within the transmitted window, then $E\left[N_{DupACK}\right] = E[\beta]$ and $E[\text{FlightSize}] = E[W]$. Hence, we can determine $E[W']$ as follows:

$$E[W'] = E[\text{ssthresh}] + E\left[N_{DupACK}\right] = \frac{E[W]}{2} + \frac{E[W]}{2} = E[W]. \tag{3.20}$$

As $E[W'] = E[\text{FlightSize}]$, it is expected that the TCP sender will not send new segments in the fast recovery phase. After the successful retransmission of the segment α_i, the TCP sender will receive a new ACK, indicating that the TCP receiver is waiting for the segment α_i+1. As a consequence of receiving this new ACK, the fast retransmit/fast recovery phase ends and according to [29] the new value of the cwnd is set as $W = \text{ssthresh}$, where the ssthresh is from (3.19). Since the number of segments that are still unacknowledged is larger than the new value of the cwnd, the TCP sender cannot transmit new segments. Therefore, this ACK will be the last one. As the TCP sender will not be able to invoke the fast retransmit algorithm again, then it will wait for expiration of the retransmission timer, which was restarted after the successful retransmission of the segment α_i in accordance to step 5.3 in [45] (denoted by the grey diamond in Fig. 3.8). After TCP retransmission timer expiration, the values of the cwnd and the ssthresh are set as $\mathrm{cwnd} = 1$ and $\mathrm{ssthresh} = \max\left(\mathrm{FlightSize}/2, 2\right)$, and the slow start phase begins. Consequently, the expected number of segments sent during the fast retransmit/fast recovery (FR) phase and the expected number of rounds in this phase can be defined as

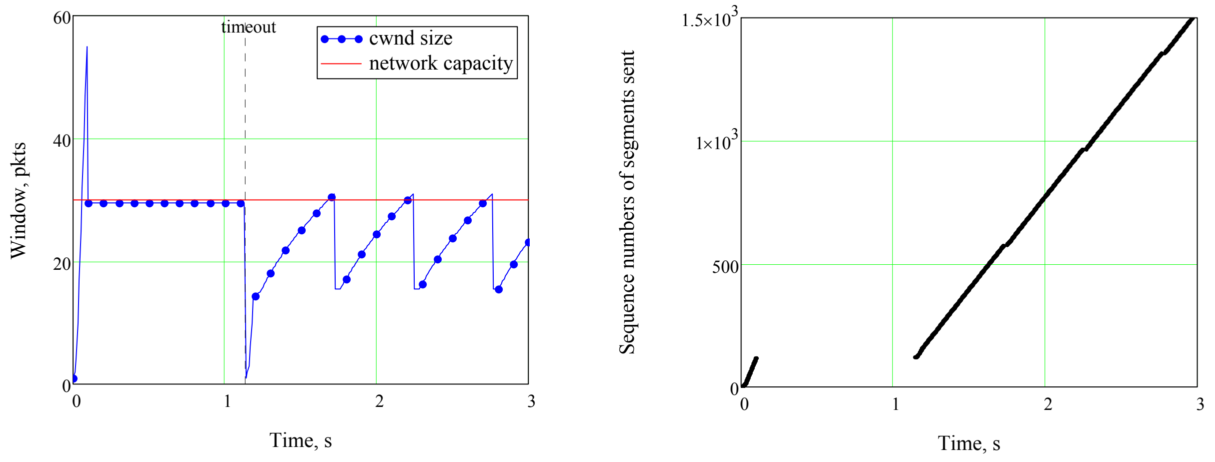
$$E\left[Y^{FR}\right] = 1, \quad E\left[A^{FR}\right] = 1. \tag{3.21}$$

Thus, in the presence of correlated losses and when the first loss in a cycle is detected via a TD loss indication, the following sequence of steps is expected:

- triggering the fast retransmit and fast recovery algorithms, retransmission of the first lost segment;
- waiting for TCP retransmission timer expiration, which was restarted after the successful retransmission of the first lost segment;
- triggering the slow start algorithm.

Our observations well agree with the results from [60] [83], showing that TCP Reno has serious performance problems when multiple segments are dropped from a window of data and that these problems result from the need to wait for TCP retransmission timer expiration before reinitiating data flow. Moreover, the empirical measurements from [67] (Table 2, columns “Loss Indic.” and “T0”) show that the significant part of loss indications (at the average, about 70%) is due to timeouts, rather than TD loss indications.

The inability of TCP Reno to recover from multiple losses without waiting for the retransmission timer to expire and performing slow start is illustrated in Fig. 3.9. The ns-2 scenario is similar to that in Chapter 2. During the unconstrained slow start phase, the number of packets injected into the network doubles every RTT (the delayed ACK algorithm is disabled). Ultimately, the bottleneck router gets overloaded, resulting in multiple packet drops due to buffer overflow. As it was noted in [60] and is shown in Fig. 3.9a, TCP Reno is optimized for the case when a single packet is dropped from a window of data (see the congestion avoidance phase with the saw-tooth oscillations caused by AIMD). But when multiple packets are dropped from the same window (see the initial slow start phase), TCP Reno has performance problems. This is caused by the fact that only one of the lost segments can be recovered by a single invocation of the fast retransmit algorithm, so the rest are usually recovered using slow start after TCP retransmission timer expiration. Obviously, lengthy idle periods result in a much smaller TCP throughput compared to an almost perfect saw-tooth pattern as assumed in [67] (see Fig. 3.6).



a) The window evolution

b) The segments sent

Fig. 3. 9 TCP Reno behavior, $\text{RTO}_{\min} = 1$ s

In order to include the fast retransmit/fast recovery phase and the slow start phase, we define a cycle to be a period between two TO loss indications (except for periods between two consecutive timeouts). Therefore, a cycle consists of the slow start phase, the congestion avoidance phase, the fast retransmit/fast recovery phase, and one timeout. An example of the evolution of the cwnd size during the i-th cycle is shown in Fig. 3.10, where the congestion avoidance phase (TDP in [67]) is supplemented with the slow start phase at the beginning and the fast retransmit/fast recovery phase with one timeout at the end.

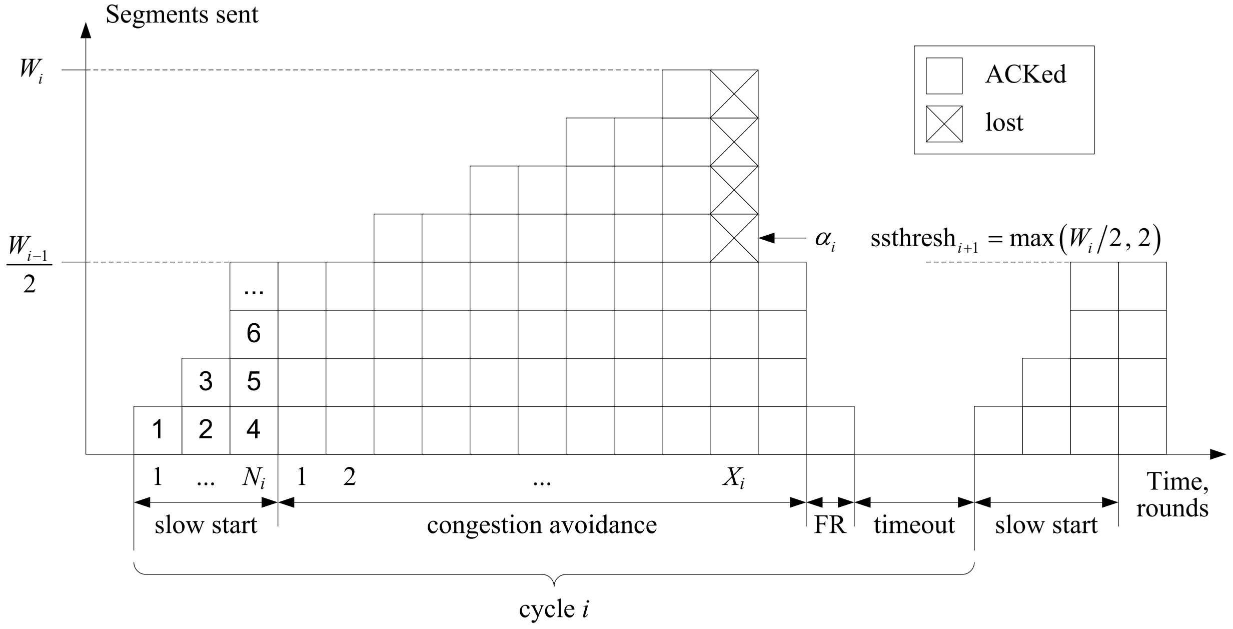


Fig. 3. 10 Example of a cycle (by the new definition)

Further modeling is generally the same as in [67] and is described in detail in [P2]. Finally, the steady-state throughput of a long-lived TCP Reno bulk data transfer can be expressed as

$$B =$$

$$\begin{cases} \dfrac{\dfrac{1}{p} + E[W] + \widehat{Q}\left(E[W]\right)\dfrac{p}{1-p}}{\overline{\mathrm{RTT}}\left(\max\left(\log_2 E[W], 2\right) + b\left(\dfrac{E[W]}{2} + 1\right) + 2 \right) + \overline{\mathrm{RTO}} + \widehat{Q}\left(E[W]\right)\left(\overline{\mathrm{RTO}}\dfrac{f(p)}{1-p} - \overline{\mathrm{RTT}} \right)}, & W_{\max} > E[W], \\ \dfrac{\dfrac{1}{p} + W_{\max} + \widehat{Q}\left(W_{\max}\right)\dfrac{p}{1-p}}{\overline{\mathrm{RTT}}\left(\max\left(\log_2 W_{\max}, 2\right) + \dfrac{bW_{\max}}{8} + \dfrac{4 + bpW_{\max}}{4pW_{\max}} + \dfrac{3}{2} \right) + \overline{\mathrm{RTO}} + \widehat{Q}\left(W_{\max}\right)\left(\overline{\mathrm{RTO}}\dfrac{f(p)}{1-p} - \overline{\mathrm{RTT}} \right)}, & W_{\max} \le E[W], \end{cases} \qquad (3.22)$$

where $W_{\max}$ is the maximum possible value of the rwnd (expressed in full-sized segments); $\widehat{Q}$ is the probability that a loss indication is TO for the expected window size at the end of a cycle, $\widehat{Q}\left(E\left[W\right]\right) \approx \min\left(1, 3/E\left[W\right]\right)$; $f\left(p\right) = 2p + 2p^2 + 4p^3 + 8p^4 + 16p^5 + 32p^6$; $\overline{\mathrm{RTT}}$ is the mean RTT; $\overline{\mathrm{RTO}}$ is the expected duration of a timeout; $E\left[W\right]$ is the expected window size at the end of a cycle and is given as

$$E\left[W\right] = -\left(\frac{2+3b}{3b}\right) + \sqrt{\frac{8}{3bp} + \left(\frac{2+3b}{3b}\right)^2}. \qquad (3.23)$$

As in the previous section, b refers to the delayed ACK algorithm: when the TCP receiver acknowledges every successfully received segment, $b = 1$; when the delayed ACK algorithm is enabled, $b = 2$.

It should be noted that the definition of $f\left(p\right)$ is different from the one used in [67]. This is because the duration of the first timeout from a sequence of consecutive timeouts has been incorporated in the duration of a cycle (see Fig. 3.10).

3.2.2 Model Validation and Conclusions

In order to validate the proposed model and compare it with the original one [67], we compare the results obtained from the analytical models against the simulation results obtained using ns-2. In our experiments, we used a “dumbbell” topology with a single bottleneck (see Fig. 3.11). To model a TCP Reno connection, we used Agent/TCP/Reno as the TCP sender, Agent/TCPSink/DelAck as the TCP receiver (so the delayed ACK algorithm was enabled), and Application/FTP as the greedy application process which always has data to send. We set the MSS to be 1460 bytes and $W_{\max}$ to be 10 full-sized segments.

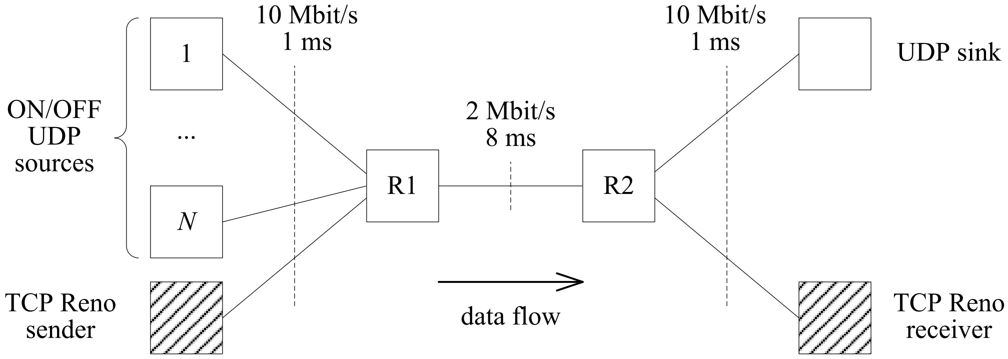


Fig. 3. 11 ns-2 simulation setup

As it was noted in [84], Web traffic tends to be self-similar in nature, while a superposition of many ON/OFF sources, whose ON/OFF times are independently drawn from heavy-tailed distributions such as the Pareto distribution, can produce asymptotically self-similar traffic [85]. Thus, we modeled the effects of competing Web-like traffic, sharing the bottleneck with the considered TCP Reno connection, as a superposition of a large number of ON/OFF UDP sources. The number of ON/OFF UDP sources was varied between 220 and 420 with a step of 10 sources. In our experiments, we used the shape parameter of the Pareto distribution of 1.2, the mean ON time of 1 second, and the mean OFF time of 2 seconds. During ON times the UDP sources sent data at 12 kbit/s.

To quantify the accuracy of the analytical models, we computed the average error using the following expression:

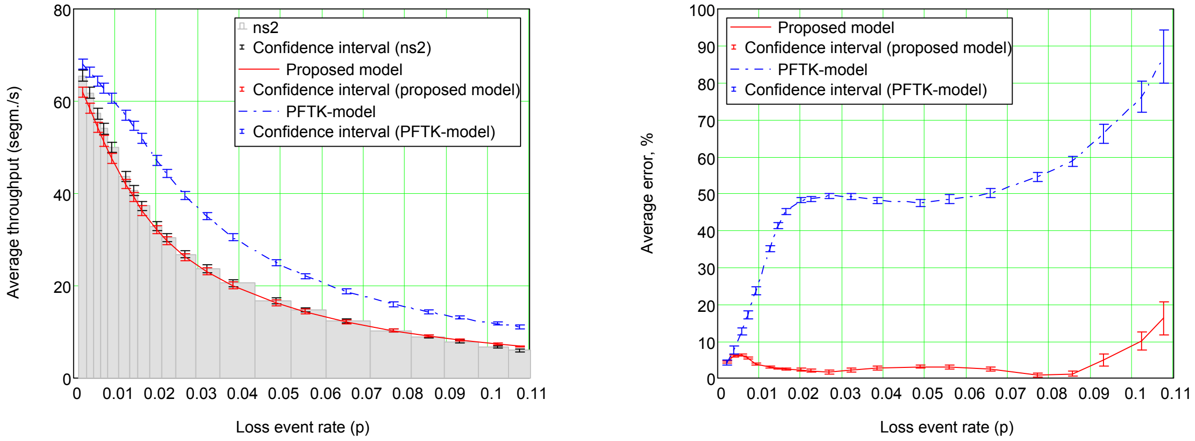
$$\text{Average error} = \frac{\sum_{\text{observations}} \frac{\left| B(p)_{\text{predicted}} - B(p)_{\text{observed}} \right|}{B(p)_{\text{observed}}}}{\text{number of observations}} 100\%. \tag{3.24}$$

Fig. 3.12a shows the average TCP Reno throughput (calculated and measured) as a function of the loss event rate p. The average errors of the proposed model and the PFTK-model are presented in Fig. 3.12b. As it follows from the results depicted in Fig. 3.12b, the refined model has the average error smaller than 5% over a wide range of loss rates with the mean of 3%, while the original PFTK-model performs well only when the loss rate is quite small and significantly overestimates TCP Reno throughput in the middle-to-high loss rate range (up to 50% and above).

Notice that by varying the number of ON/OFF UDP sources and, therefore, the volume of background traffic, we obtain the following two extreme cases (see Fig. 3.13a).

- When the volume of background traffic is very low, almost all packet losses are caused by the considered TCP connection itself. In this case, the transmission window size shows a perfect saw-tooth behavior: when the last segment in a window gets lost, the current value of the cwnd is halved, the TCP sender performs a retransmission of what appears to be the missing segment and returns to the congestion avoidance mode with a linear increase of the cwnd size until the next loss. Thus, segment losses are independent and the loss process tends to be periodic. Since the proposed model assumes that packet losses are correlated within a round, the throughput prediction error of the revised model increases. Meanwhile, since almost all the time the TCP connection operates in the congestion avoidance mode and, hence, loss recovery dynamics does not seriously affect the performance, the throughput prediction error of the PFTK-model decreases (see Fig. 3.12b). Fig. 3.13b presents the average number of lost segments per loss event according to the ns-2 simulations and (3.18). While constructing the model, we assumed (similar to [67]) that in case of congestion along the path about half of the transmitted window of segments will be lost. It is easy to see that in our simulations this assumption holds only when the number of ON/OFF UDP sources is equal to 380. In this case, the average error of the proposed model is less than 1%.

- On the other hand, when the volume of background traffic is quite high, the timeout probability increases, so the TCP connection spends the most part of its lifetime in the slow start phase or waiting for TCP retransmission timer expiration. Under such conditions, both models fail to accurately predict the steady-state throughput (Fig. 3.12b).

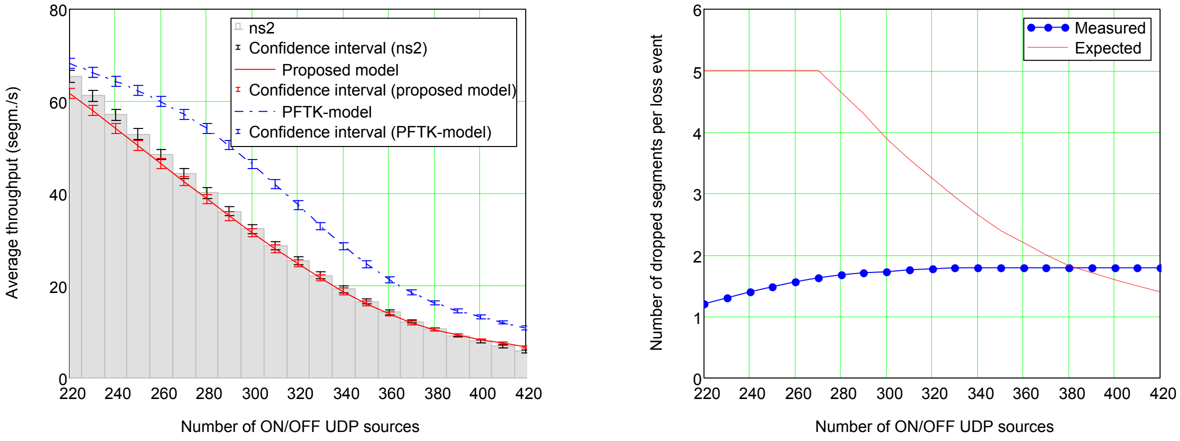


a) The steady-state throughput

b) The average error

Fig. 3. 12 ns-2 simulation results versus predicted values for the TCP Reno steady-state throughput

Finally, it is worthwhile to note that it is a non-trivial task to simulate large bursts of packet loss (as assumed by the bursty loss model in [67] and, thus, in [P2]), since a typical ns-2 loss event consists of just a few packet drops per TCP flow (Fig. 3.13b). A similar observation was also made in [86]. This is caused by statistical multiplexing and interleaving of packets from different flows, which expands the inter-packet distance of each flow and, hence, decreases the number of dropped packets belonging to the same flow in case of congestion. Moreover, the empirical measurements from [82, p.51] show the majority of losses are single packet losses. Nevertheless, we believe that even if the bursty loss model is not very typical in today's Internet, TCP performance in the face of such losses deserves to be addressed and carefully studied.



a) On the accuracy of the models

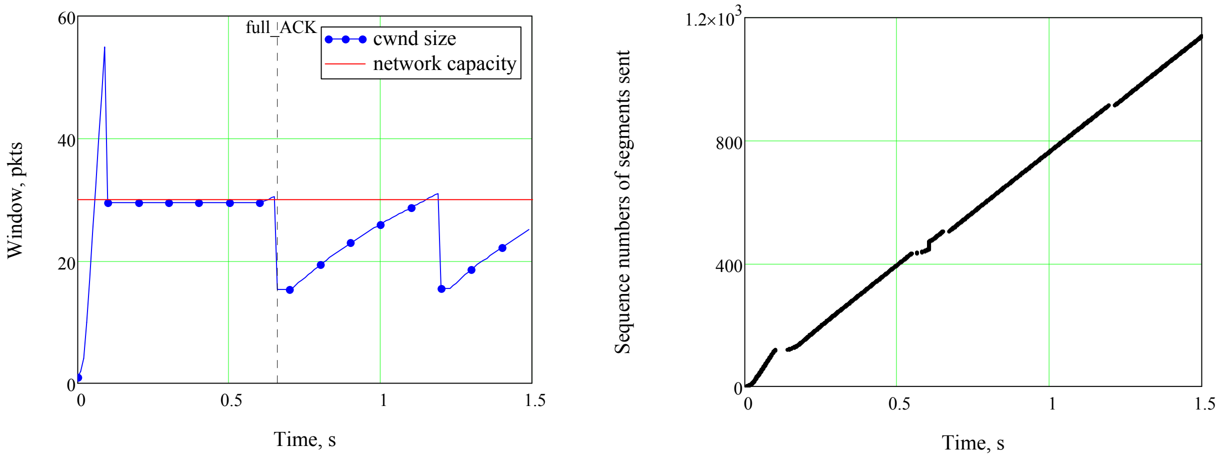
b) On the average loss burst length

Fig. 3. 13 Effect of the volume of background traffic on the models' accuracy

3.3 TCP NewReno Performance

For a long time, the reference TCP implementation has been TCP Reno first deployed in the 4.3BSD-Reno and specified in [29]. This document defines the four intertwined congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. TCP NewReno is a subsequent modification of the basic TCP Reno implementation and incorporates slow start, congestion avoidance, and fast retransmit from [29] with a modified fast recovery algorithm [49]. This modification concerns the sender's behavior during fast recovery when a partial ACK is received that acknowledges some but not all of the segments sent before entering the fast recovery phase. While in TCP Reno the reception of a partial ACK takes the TCP sender out of the fast recovery mode, in TCP NewReno the TCP sender stays in fast recovery until either a full ACK arrives that acknowledges all of the segments outstanding by the time the fast recovery phase was entered (see Fig. 3.14), or the TCP retransmission timer expires (see Fig. 3.15). As a result, TCP NewReno provides more stable performance in the face of multiple dropped segments and, therefore, avoids multiple reductions of the cwnd and reduces the frequency of timeout-based loss recovery, which are typical for TCP Reno [60].

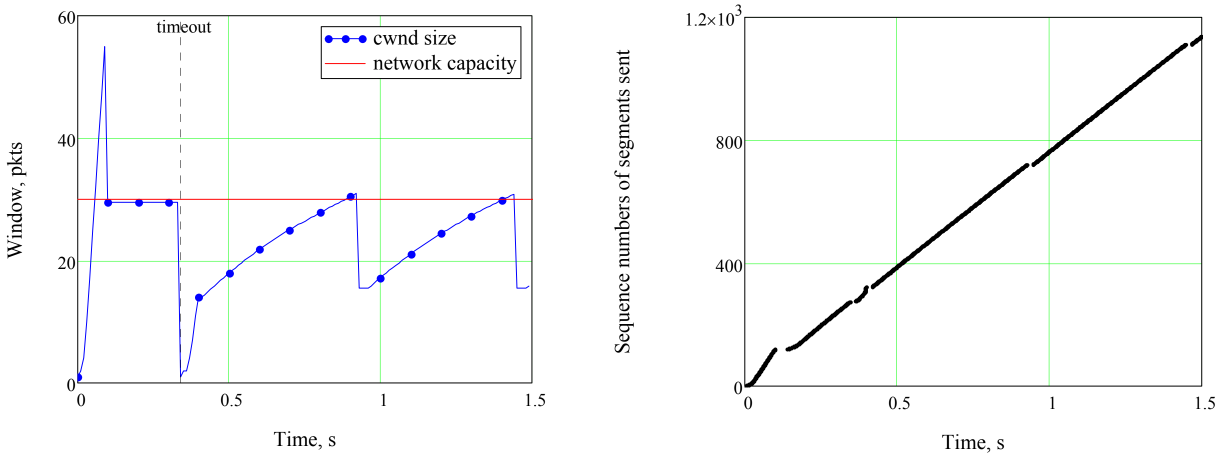
The current standard [49] specifies two variants of TCP NewReno: Slow-but-Steady and Impatient. The only difference between them lies in the TCP retransmission timer resetting scheme in response to partial ACKs. In the Slow-but-Steady variant, the TCP sender resets the retransmission timer after each partial ACK. Consequently, when N segments have been lost from a window of data, the Slow-but-Steady variant can remain in the fast recovery phase for N RTTs, retransmitting by one lost segment every RTT (Fig. 3.14a). In the Impatient variant, the TCP sender performs resetting only after the first partial ACK. Therefore, if a large number of segments were lost from a window of data, the TCP retransmission timer ultimately expires and the TCP sender will enter the slow start phase (Fig. 3.15a). Depending on the given operating conditions (the number of lost segments, delay variation, etc.) either one or the other variant may provide better performance.



a) The window evolution

b) The segments sent

Fig. 3. 14 The Slow-but-Steady variant of TCP NewReno, $\text{RTO}_{\min} = 200$ ms



a) The window evolution

b) The segments sent

Fig. 3. 15 The Impatient variant of TCP NewReno, $\text{RTO}_{\min} = 200$ ms

3.3.1 Motivation and Model Building

While TCP performance modeling has received a lot of attention during the last years, the majority of the proposed models were developed for the TCP Reno implementation (e.g., see [26] and references therein). In the absence of sufficient analytical background, the current standard [49] recommends the Impatient variant of TCP NewReno based only on simulation results. Although network simulation is a powerful tool in protocol analysis and design, it is a difficult task to simulate TCP NewReno behavior across the range of all possible operating conditions and protocol parameter settings. In this case, an analytical model would be extremely useful because it not only allows to apply a "what if" test to different scenarios, but also to explore TCP NewReno performance over the entire parameter space. Thus, the objective of the study in [P4] is two-fold. Firstly, we develop an analytical model of the steady-state throughput of the Impatient variant of TCP NewReno, which together with the previously obtained model of the Slow-but-Steady variant in [P3] gives us a comprehensive model of TCP NewReno throughput. We then evaluate the TCP NewReno variants and define the most preferable one. Since the difference between them only appears when multiple segments are lost from the same window of data, we focus our analysis on the case of bursty losses inherent to a Drop-Tail environment. While constructing our model, we use exactly the same assumptions about the hosts and the network as in section 3.2.1 (of course, with the exception of the TCP implementation in use, which is TCP NewReno in this case).

According to [49], a segment loss (if the TCP sender is not already in the fast recovery phase) can be detected in one of the two ways: either by reception of three duplicate ACKs or via TCP retransmission timer expiration. In the latter case, the TCP sender enters slow start and recovers what appears to be the lost segment(s) using the Go-Back-N strategy, but no fast recovery is performed. Since the fast recovery algorithm is the distinctive feature of TCP NewReno, we focus on the "pure" TCP NewReno behavior, where all loss detections are due to "triple-duplicate" ACKs. However, the model can be easily extended to capture loss detections via timeouts by following the approach proposed in [67].

Similar to [67], let us consider steady-state TCP NewReno behavior as a sequence of renewal cycles, where a cycle is a period between two consecutive loss events detected by reception of three duplicate ACKs. Then we can define the TCP steady-state throughput as the ratio between the expected number of segments sent during a cycle and the expected duration of a cycle. Let δ_{i-1} denotes the number of segments lost in cycle $i-1$. In contrast to the Slow-but-Steady variant, where the TCP sender recovers one lost segment per round and the number of rounds in the fast retransmit/fast recovery phase of the i-th cycle can be defined as $A_i^{FR} = \delta_{i-1}$, in the Impatient variant the TCP sender resets the retransmission timer only after the first partial ACK, so the number of rounds in the fast retransmit/fast recovery phase of the i-th cycle can be expressed as $A_i^{FR} = \min\left(\delta_{i-1}, 1 + \mathrm{RTO}/\mathrm{RTT}\right)$. Let us define $\overline{\delta}$ to be the average number of segments lost in a row per loss event (also known as the average loss burst length) and τ to be the ratio between the expected duration of a timeout ($\overline{\mathrm{RTO}}$) and the mean RTT ($\overline{\mathrm{RTT}}$), $\tau = \overline{\mathrm{RTO}}/\overline{\mathrm{RTT}}$, $\tau > 1$. For simplicity, we suppose that τ is integer. We assume that when $\overline{\delta} < \tau + 1$, the resetting scheme of the TCP retransmission timer has a negligible effect on the fast recovery phase and the steady-state throughput of the Impatient variant is identical to that of the Slow-but-Steady one. On the other hand, if $\overline{\delta} \geq \tau + 1$, it is expected that the TCP sender cannot recover all lost segments during the fast recovery phase and after TCP retransmission timer expiration it will invoke the slow start algorithm. Thus, a Slow-but-Steady cycle consists of the fast retransmit/fast recovery (FR) phase followed by the congestion avoidance (CA) phase, while an Impatient cycle includes, in addition, the slow start (SS) phase after fast recovery. Fig. 3.16 and Fig. 3.17 present examples of transmission of segments during the i-th cycle for the Slow-but-Steady and Impatient variants, respectively.

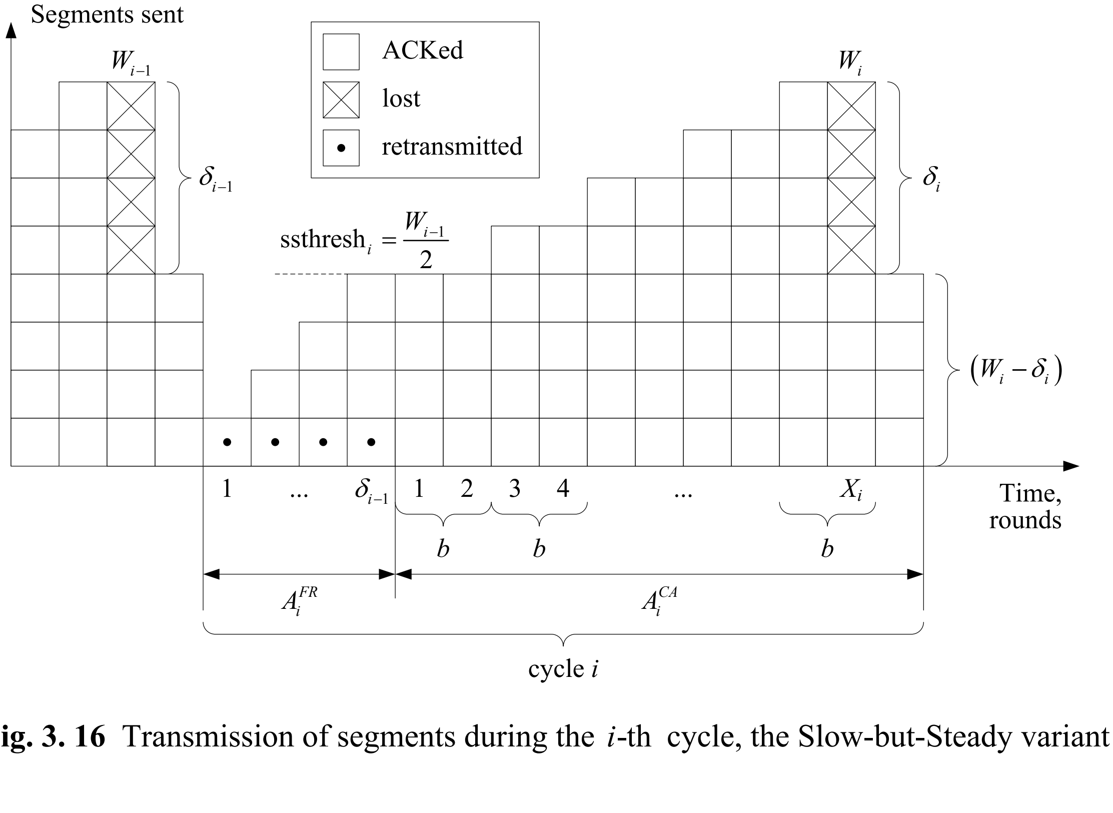


Fig. 3. 16 Transmission of segments during the i-th cycle, the Slow-but-Steady variant

After reception of the third duplicate ACK, the TCP sender enters the fast retransmit/fast recovery phase and sets the ssthresh as (3.19). Note that duplicate ACKs are triggered by out-of-order segment arrivals at the TCP receiver, while the fast retransmit algorithm resends a segment only when three duplicate ACKs arrive in a row at the TCP sender. This implies that to trigger the fast retransmit algorithm, the TCP sender must have at least four outstanding segments, out of which three segments must be successfully delivered. Then $W_{i-1} \geq 4$ and we get:

$$\text{ssthresh}_i = \max\left(W_{i-1}/2, 2\right) = W_{i-1}/2. \tag{3.25}$$

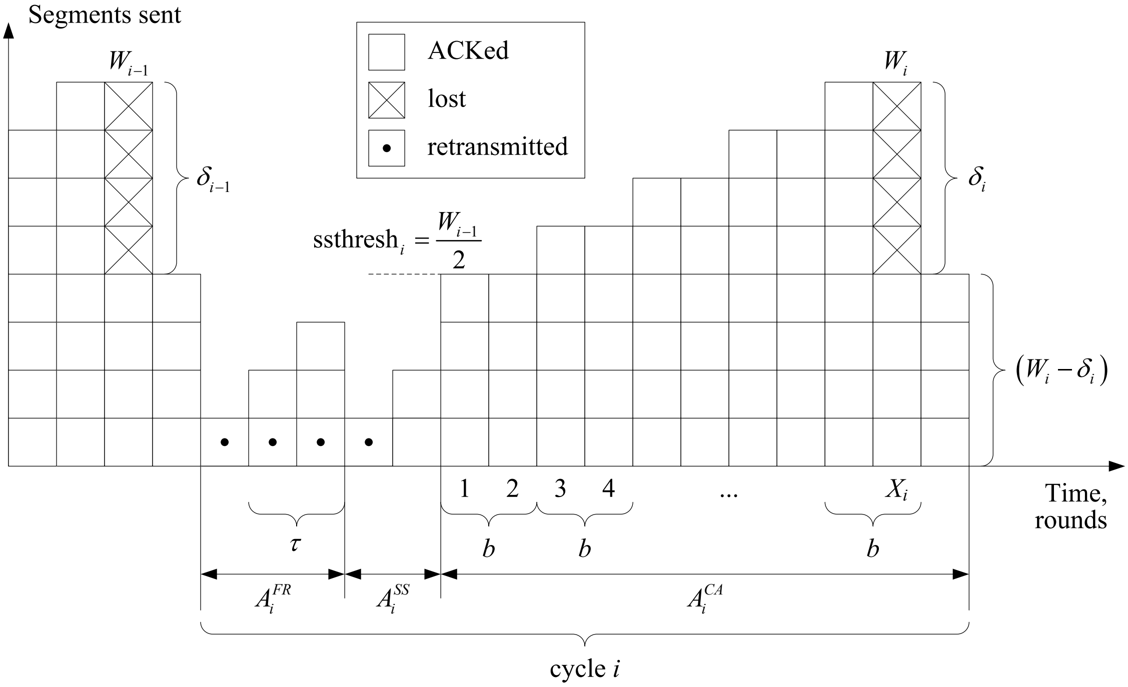


Fig. 3. 17 Transmission of segments during the i-th cycle, the Impatient variant

As shown in [87] [88], the number of new segments sent in the k-th round of the fast retransmit/fast recovery phase of the i-th cycle can be found as

$$P(k) = \max\left(0, \text{ssthresh}_i - \delta_{i-1} + k - 1\right), \quad 1 \leq k \leq \delta_{i-1}, \tag{3.26}$$

where ssthresh_i is given by (3.25).

Using (3.26) and taking into account the number of retransmitted segments, we can determine the total number of segments sent during the fast retransmit/fast recovery phase of the i-th cycle. In case of the Slow-but-Steady variant, we obtain:

$$Y_i^{FR} = \delta_{i-1} + \sum_{k=1}^{\delta_{i-1}} P(k). \tag{3.27}$$

Then the expected number of segments sent during the fast retransmit/fast recovery phase can be defined from (3.27) as

$$E\left[Y^{FR}\right]=\begin{cases}\bar{\delta}+\dfrac{\bar{\delta}}{2}\left(E[W]-\bar{\delta}-1\right), & \bar{\delta}\le\dfrac{E[W]}{2},\\ \bar{\delta}+\dfrac{E[W]}{4}\left(\dfrac{E[W]}{2}-1\right), & \bar{\delta}>\dfrac{E[W]}{2}.\end{cases} \tag{3.28}$$

For the Impatient variant, we get:

$$Y_i^{FR}=\tau+1+\sum_{k=1}^{\tau+1}P(k), \tag{3.29}$$

and

$$E\left[Y^{FR}\right]=\begin{cases}\tau+1+(\tau+1)\left(\dfrac{E[W]+\tau-2\bar{\delta}}{2}\right), & \bar{\delta}\le\dfrac{E[W]}{2},\\ \tau+1+\dfrac{E[W]}{4}\left(\dfrac{E[W]}{2}-1\right)+\left(\tau+1-\bar{\delta}\right)\left(\dfrac{E[W]+\tau-\bar{\delta}}{2}\right), & \bar{\delta}>\dfrac{E[W]}{2}.\end{cases} \tag{3.30}$$

Further modeling is generally the same as in [67] and is described in detail in [P3] [P4]. Finally, the steady-state throughput of a long-lived TCP NewReno bulk data transfer can be expressed as follows. In case of the Slow-but-Steady (SBS) variant, we have:

$$B^{SBS}=\begin{cases}\dfrac{1/p+E\left[W^{SBS}\right]-1}{\overline{\text{RTT}}\left(\bar{\delta}+b\left(\dfrac{E\left[W^{SBS}\right]}{2}+1\right)+1\right)}, & E\left[W^{SBS}\right]<W_{\max},\\ \dfrac{1/p+W_{\max}-1}{\overline{\text{RTT}}\left(\bar{\delta}+\dfrac{bW_{\max}}{2}+E\left[V^{SBS}\right]+1\right)}, & E\left[W^{SBS}\right]\ge W_{\max},\end{cases} \tag{3.31}$$

where p is the loss event rate; $E\left[W^{SBS}\right]$ is the expected window size at the end of a cycle; $E[V]$ is the expected number of rounds when the transmission window size is limited by the TCP receiver, so it remains constant and equal to the maximum size of the rwnd (denoted as $W_{\max}$).

$$E\left[W^{SBS}\right]=\begin{cases}-\left(\dfrac{3b+2\bar{\delta}}{3b}\right)+\sqrt{\dfrac{8}{3bp}+\dfrac{4\left(\bar{\delta}^2+\bar{\delta}-2\right)}{3b}+\left(\dfrac{3b+2\bar{\delta}}{3b}\right)^2}, & \bar{\delta}\le\dfrac{E\left[W^{SBS}\right]}{2},\\ -\left(\dfrac{3b-1}{3b+1}\right)+\sqrt{\dfrac{8(1-p)}{p(3b+1)}+\left(\dfrac{3b-1}{3b+1}\right)^2}, & \bar{\delta}>\dfrac{E\left[W^{SBS}\right]}{2},\end{cases} \tag{3.32}$$

and

$$E\left[V^{SBS}\right]=\begin{cases}\dfrac{1-p}{pW_{\max}}+\dfrac{\overline{\delta}^2+\overline{\delta}-\overline{\delta}W_{\max}}{2W_{\max}}+\dfrac{b}{4}-\dfrac{3bW_{\max}}{8}, & \overline{\delta}\le\dfrac{W_{\max}}{2},\\ \dfrac{1-p}{pW_{\max}}+\dfrac{2b-W_{\max}}{8}+\dfrac{1}{4}-\dfrac{3bW_{\max}}{8}, & \overline{\delta}>\dfrac{W_{\max}}{2}.\end{cases}\tag{3.33}$$

In case of the Impatient (IMP) variant, we have:

$$B^{IMP}=\begin{cases}B^{SBS}, & \overline{\delta}<\tau+1,\\ \dfrac{1/p+E\left[W^{IMP}\right]-1}{\overline{\mathrm{RTT}}\left(\tau+1+\log_2\left(E\left[W^{IMP}\right]\right)+b\left(\dfrac{E\left[W^{IMP}\right]}{2}+1\right)\right)}, & \overline{\delta}\ge\tau+1, E\left[W^{IMP}\right]<W_{\max},\\ \dfrac{1/p+W_{\max}-1}{\overline{\mathrm{RTT}}\left(\tau+1+\log_2\left(W_{\max}\right)+\dfrac{bW_{\max}}{2}+E\left[V^{IMP}\right]\right)}, & \overline{\delta}\ge\tau+1, E\left[W^{IMP}\right]\ge W_{\max},\end{cases}\tag{3.34}$$

where

$$E\left[W^{IMP}\right]=$$

$$\begin{cases}-\left(\dfrac{3b+2\tau+4}{3b}\right)+\sqrt{\dfrac{8}{3bp}+\left(\dfrac{3b+2\tau+4}{3b}\right)^2+\dfrac{4\left(4\overline{\delta}+2\overline{\delta}\tau-\tau^2-3\tau-2\right)}{3b}}, & \overline{\delta}\le\dfrac{E\left[W^{IMP}\right]}{2},\\ -\left(\dfrac{3b+2\tau+3-2\overline{\delta}}{3b+1}\right)+\sqrt{\dfrac{8}{p\left(3b+1\right)}+\left(\dfrac{3b+2\tau+3-2\overline{\delta}}{3b+1}\right)^2+\dfrac{4\left(3\overline{\delta}+2\overline{\delta}\tau-\tau^2-\overline{\delta}^2-3\tau-2\right)}{3b+1}}, & \overline{\delta}>\dfrac{E\left[W^{IMP}\right]}{2},\end{cases}\tag{3.35}$$

and

$$E\left[V^{IMP}\right]=$$

$$\begin{cases}\dfrac{2-2p-3\tau p-\tau^2p+2\overline{\delta}\tau p+4\overline{\delta}p}{2pW_{\max}}-\dfrac{\tau}{2}-1-\dfrac{3bW_{\max}}{8}+\dfrac{b}{4}, & \overline{\delta}\le\dfrac{W_{\max}}{2},\\ \dfrac{2-2p-3\tau p-\overline{\delta}^2p+2\overline{\delta}\tau p+3\overline{\delta}p-\tau^2p}{2pW_{\max}}-\dfrac{\tau}{2}-\dfrac{3}{4}-\dfrac{3bW_{\max}}{8}+\dfrac{b}{4}-\dfrac{W_{\max}}{8}+\dfrac{\overline{\delta}}{2}, & \overline{\delta}>\dfrac{W_{\max}}{2}.\end{cases}\tag{3.36}$$

To the best of our knowledge, the only analytical study that explicitly addresses a comparison of the TCP NewReno variants was recently presented by Parvez *et al.* in [86]. The authors studied the TCP NewReno variants both analytically and using simulations and argued that the Slow-but-Steady variant is superior to the Impatient one in all but the most extreme network conditions and recommended it as the preferred variant of TCP NewReno, contrary to [49]. Unfortunately, whereas decision-making in protocol design requires very careful consideration and detailed analysis, there are several inaccuracies in the model developed in [86], which lead to wrong results and conclusions. These inaccuracies are as follows.

- Firstly, the authors assumed that within the fast recovery phase the Slow-but-Steady variant transmits segments at a constant rate of $W/4$ segments per round (see Fig. 3.18). However, the assumption about the constant sending rate contradicts the actual TCP behavior as described in (3.26) (see [87] for details).
- Secondly, the authors did not count new segment transmissions during the fast recovery phase of the Impatient variant (see Fig. 3.19). As it will be shown later, when the number of lost segments is large enough and $\overline{\mathrm{RTO}} \gg \overline{\mathrm{RTT}}$, such simplification results in underestimation of the performance of the Impatient variant.
- Thirdly, the authors supposed that upon occurrence of a loss event, the TCP sender enters the fast recovery phase and reduces the cwnd from W to $W/2$. Moreover, it sets the ssthresh to $W/2$. Assuming that the number of lost segments is greater than that can be recovered before a timeout event ($\delta \geq \tau + 1$), the TCP retransmission timer eventually expires, so the TCP sender enters the slow start phase and sets the ssthresh to half of the current cwnd size (i.e., $\mathrm{ssthresh} = W/4$). However, the assumption about the latter halving is incorrect because TCP computes the ssthresh only once upon entering the fast recovery phase. In fact, the value of the ssthresh is given by (3.19) (see [49] for details).

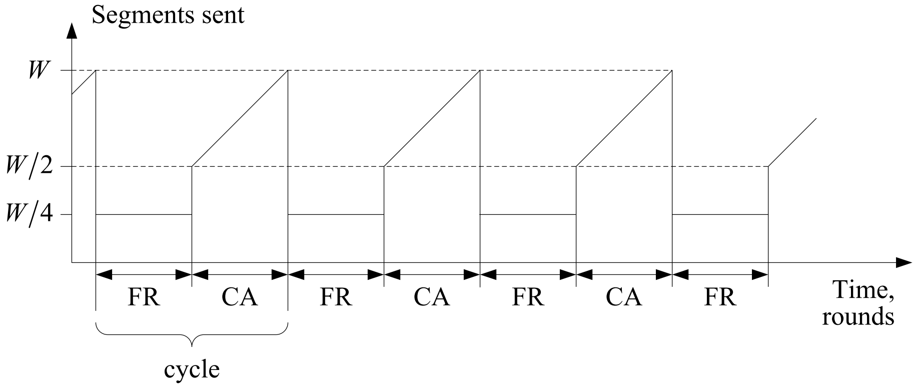


Fig. 3. 18 Cycles of the Slow-but-Steady variant (as assumed in [86])

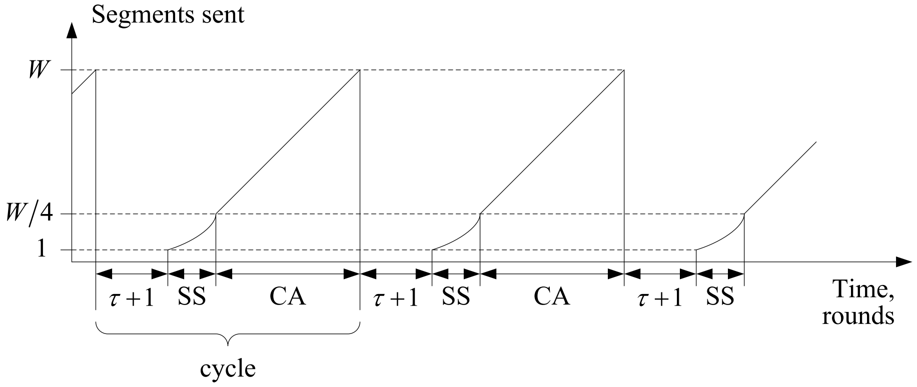
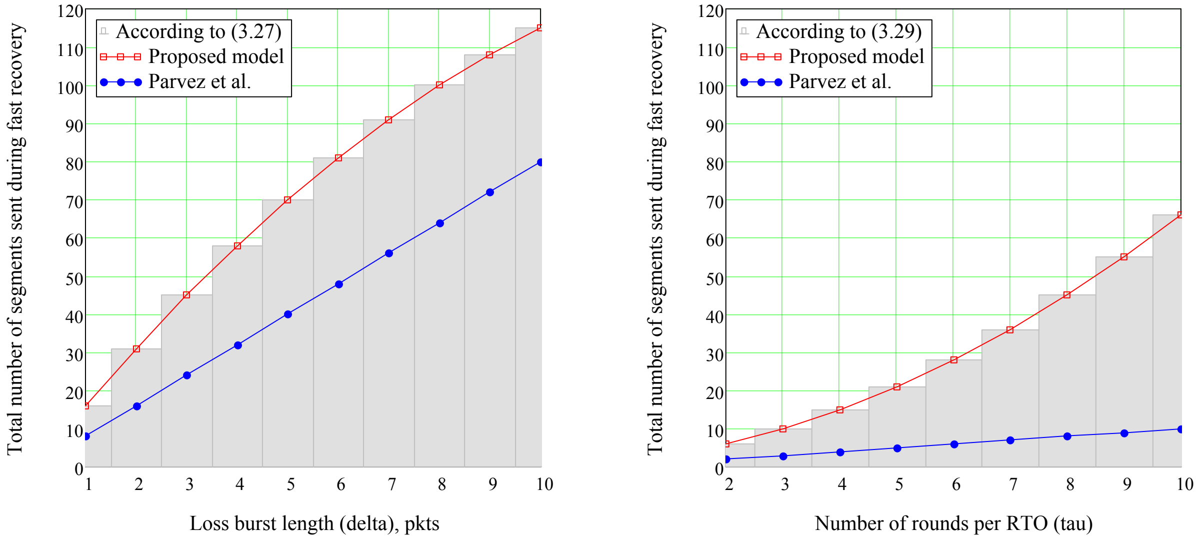


Fig. 3. 19 Cycles of the Impatient variant (as assumed in [86])

In order to compare the accuracy of the proposed model and the model developed in [86], we use the values listed in Table 3.6. Note that $\tau = 10$ can be considered as the case where $\overline{\text{RTO}} = 1$ s (as specified in [45]) and $\overline{\text{RTT}} = 100$ ms. As it follows from the results depicted in Fig. 3.20, the proposed model captures fast retransmit/fast recovery dynamics without any error, while the model presented in [86] significantly underestimates TCP NewReno performance during this phase, especially in case of multiple packet drops.

Table 3. 6 Input parameters for comparing the accuracy of the models from [86] [P3] [P4]

Input parameter	Slow-but-Steady	Impatient
Window size at the end of a cycle (W)	32	32
Average loss burst length ($\overline{\delta}$)	1, …, 10; step 1	16
RTO granularity (τ)	16	2, …, 10; step 1



a) The Slow-but-Steady variant

b) The Impatient variant

Fig. 3. 20 Total number of segments sent during the fast retransmit/fast recovery phase

3.3.2 Numerical Analysis and Conclusions

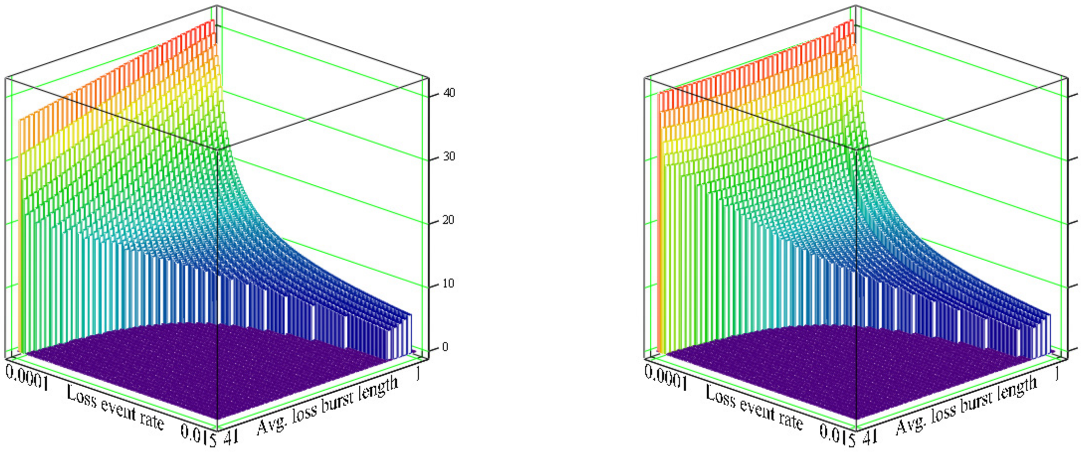
Armed with the expressions of the steady-state throughput of the Impatient and Slow-but-Steady variants, we can perform an analytical comparison of these variants over different values of $\overline{\delta}$,

p, b, τ, and $W_{\max}$. Note that the latter parameter not only captures the impact of the receive buffer size on TCP NewReno performance, but also allows us to place an upper bound on the maximum value of $\overline{\delta}$: since the number of full-sized segments transmitted in any RTT must be no more than $W_{\max}$ and we assume that all loss indications are exclusively due to "triple-duplicate" ACKs, then $\delta_i \le W_{\max} - 3$ and $W_i \ge 4$. Moreover, the fact that $E[W] \ge 4$ allows us to obtain an upper bound of the loss event rate p. As it follows from (3.32) and (3.35), $p \le 0.015$ (above this value it may not be possible to generate the required number of duplicate ACKs to trigger the fast retransmit algorithm when a loss occurs). Values of the default input parameters used in the numerical analysis are listed in Table 3.7. Here we assume that $\text{rwnd} = 65{,}535$ bytes (used by default in Microsoft Windows XP, the most popular OS [62]) and $\text{MSS} = 1460$ bytes. In practice, instead of using a hard-coded rwnd size, TCP adjusts it to even increments of the MSS announced during the connection establishment phase [89]. Then we get that $W_{\max} = 44$ full-sized segments and the maximum value of $\overline{\delta}$ is 41 segments. Fig. 3.21 shows the steady-state throughputs of the Slow-but-Steady and Impatient variants as a function of $\overline{\delta}$ and p.

Table 3. 7 Default parameters for estimating TCP NewReno throughput

Input parameter	**Value**
Maximum segment size (MSS)	1460 bytes
Maximum size of the rwnd ($W_{\max}$)	44 full-sized segments
Average loss burst length ($\overline{\delta}$)	1, …, 41; step 1
Loss event rate (p)	0.0001, …, 0.0150; step 0.0001
Number of segments acknowledged by one ACK (b)	2
RTO granularity (τ)	4

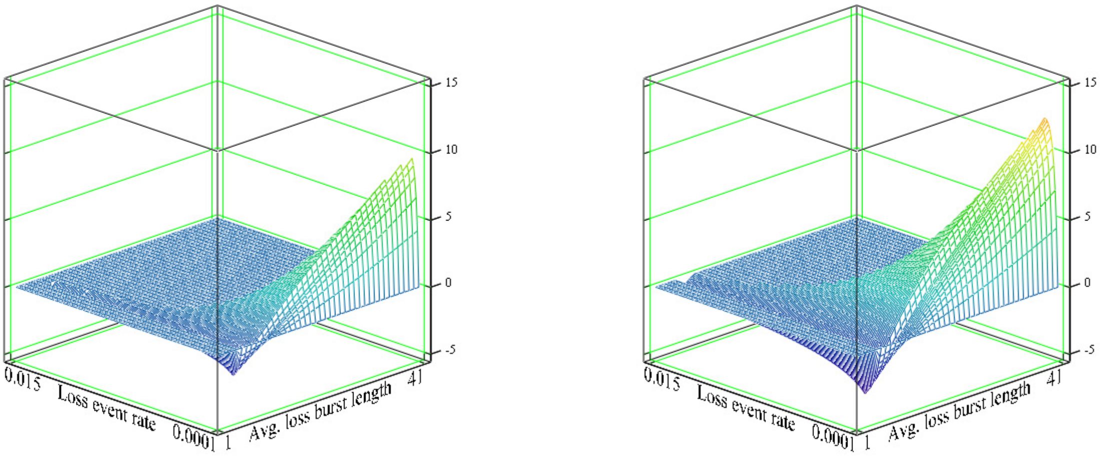
As it follows from Fig. 3.21, the TCP NewReno throughput exhibits quite complex behavior. First of all, we note that there is a set of input parameter values for which the TCP NewReno throughput is undefined and set to zero. This is either due to our assumption that all losses are detected via the receipt of three duplicate ACKs (this imposes an effective constraint on the parameter space), or due to the fact that such parameter combination is invalid (e.g., a large number of packets dropped from a single window of data implies a large size of the cwnd, which is impossible at high packet loss rates). Observing Fig. 3.21, we also notice that when the loss event rate is very small ($p \to 0$) and losses are predominantly single packet losses ($\overline{\delta} \to 1$), the TCP NewReno steady-state throughput tends to $W_{\max}/\text{RTT}$. Note that for convenience of representation all subsequent plots showing the difference between the TCP NewReno variants are rotated by 180 degrees from the position in Fig. 3.21.



a) The Slow-but-Steady variant

b) The Impatient variant

Fig. 3. 21 TCP NewReno steady-state throughput (in segm./RTT)



a) $b = 2$

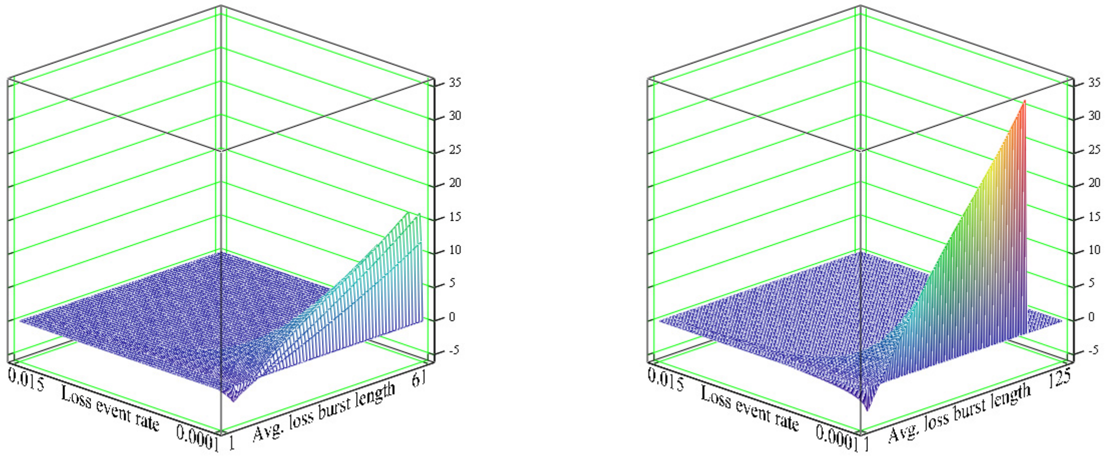
b) $b = 1$

Fig. 3. 22 Effect of the delayed ACK algorithm on the difference (in segm./RTT) between the steady-state throughputs of the Impatient and Slow-but-Steady variants

Now let us consider how the delayed ACK algorithm impacts the steady-state throughputs of the TCP NewReno variants. Fig. 3.22 shows the difference between the Impatient and Slow-but-Steady variants when the delayed ACK algorithm is enabled (Fig. 3.22a) or disabled (Fig. 3.22b). In the first place, it should be emphasized that when the average loss burst length is less than five packets, the steady-state throughputs of both variants are identical. This is because the TCP sender can recover all lost segments before TCP retransmission timer expiration ($\tau = 4$), which was reset after the first partial ACK. But when the average number of lost segments per congestion event is greater than five (i.e., $\tau + 1$), the difference between the variants takes place. As it was noted in [49], neither of the two variants is optimal. When the number of lost segments is small, the performance would have been better without invocation of the slow start algorithm. However, when the number of lost segments is sufficiently large, the Impatient variant provides a faster recovery and better performance, while the gain increases with the average loss burst length.

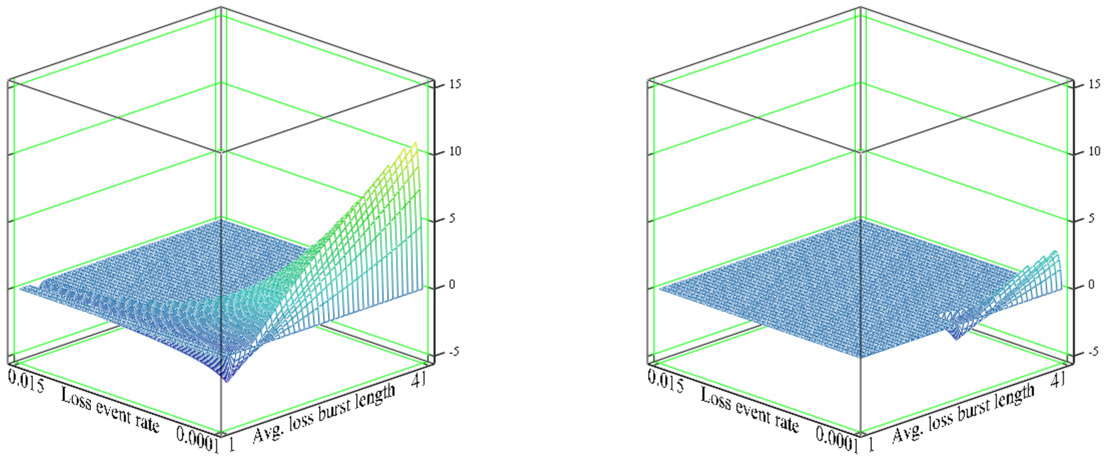
Secondly, the results show that when the TCP receiver does not use the delayed ACK algorithm and acknowledges every successfully received segment ($b = 1$), the Impatient variant provides better performance over a wider range of network conditions. This is due to the fact that the TCP receiver generates more ACKs, so the cwnd size grows faster, and more packets are injected into the network. As a result, TCP encounters segment loss more frequently, while the Impatient variant outperforms the Slow-but-Steady one when many segments are lost at once.

To quantify the impact of using the TCP window scale option on the steady-state throughputs of the Impatient and Slow-but-Steady variants, we consider two cases taken from Fig. 2.2h and Fig. 2.2g: $\text{rwnd} = 5840 \times 16 = 93{,}440$ bytes and $\text{rwnd} = 5840 \times 32 = 186{,}880$ bytes (the window scale factor is equal to 4 and 5, respectively). As it follows from Fig. 3.23, under given conditions the maximum gain of the Slow-but-Steady variant is almost constant and varies from 2.8 to 3.6 segm./RTT, while the maximum gain of the Impatient variant increases with the maximum size of the rwnd (up to 16.6 segm./RTT for $\text{rwnd} = 93{,}440$ bytes and 34.7 segm./RTT for $\text{rwnd} = 186{,}880$ bytes), since a larger rwnd means more segments in transit, which can be potentially dropped in case of congestion along the end-to-end network path. However, the Impatient variant outperforms the Slow-but-Steady one only when the average loss burst length is more then 10 segments. As was pointed out in [82], while the right tail of the loss burst length distribution can be fairly long (up to 100 packets and more), most losses are single packet losses. Then we can expect that in the majority of cases the Impatient variant will behave like the Slow-but-Steady one (since all lost segments can be recovered within several rounds before TCP retransmission timer expiration).



a) rwnd = 93,440 bytes, $W_{\max} = 64$ b) rwnd = 186,880 bytes, $W_{\max} = 128$

Fig. 3. 23 Effect of using the window scale option on the difference (in segm./RTT) between the steady-state throughputs of the Impatient and Slow-but-Steady variants



a) $\tau = \overline{\mathrm{RTO}} \big/ \overline{\mathrm{RTT}} = 2$ b) $\tau = \overline{\mathrm{RTO}} \big/ \overline{\mathrm{RTT}} = 20$

Fig. 3. 24 Effect of the TCP timer granularity on the difference (in segm./RTT) between the steady-state throughputs of the Impatient and Slow-but-Steady variants

Finally, let us consider the impact of the TCP timer granularity on the steady-state throughputs of the Impatient and Slow-but-Steady variants. Commonly, TCP implementations use a coarse-grained retransmission timer, having granularity of 500 ms. Moreover, the current standard [45] specifies the lower bound of the timeout value as 1 second. However, some TCP implementations use a fine-grained retransmission timer and do not follow the requirements of

[45] by allowing, for example, the minimum limit of 200 ms [90]. Thus, for the same network conditions the value of τ can vary greatly from one TCP implementation to another. Fig. 3.24 illustrates the difference between the steady-state throughputs of the Impatient and Slow-but-Steady variants for fine-grained (Fig. 3.24a) and coarse-grained (Fig. 3.24b) timers.

Fig. 3.24 shows that using a fine-grained retransmission timer, the Impatient variant provides a higher steady-state throughput in case of large transmission windows and multiple losses, while the gain of the Impatient variant with a coarse-grained retransmission timer in that case will be substantially smaller due to a very lengthy fast recovery phase. Taking into account the prevalence of single packet losses, we believe that in most cases the Impatient variant with a coarse-grained retransmission timer will behave in exactly the same way as the Slow-but-Steady one.

The results of the study allow us to draw the following conclusions.

- The Impatient variant provides approximately the same steady-state throughput as the Slow-but-Steady one in a wide range of network conditions and significantly outperforms the latter one in case of large windows and bursty losses.
- We can expect that under normal operating conditions there will be no difference between the Impatient and Slow-but-Steady variants, since in most cases all lost segments can be recovered in the Slow-but-Steady mode. Nevertheless, our recommendation is for the Impatient variant as a backup mechanism for extreme scenarios with multiple packet drops.

4. TCP PERFORMANCE IN WIRED-CUM-WIRELESS NETWORKS

This chapter introduces a performance evaluation model for a TCP connection running over a wired-cum-wireless network. The background and related work, as well as the obtained results, are also presented and discussed.

4.1 Background and Related Work

The basic cause of the degradation of TCP performance in wireless and wired-cum-wireless networks is that TCP does not distinguish between packet losses induced by network congestion and packet losses due to incorrect reception of channel symbols [16]. In fact, the latter case does not imply that the given path cannot support the current rate at which packets are injected into the network. Nevertheless, in both cases packet losses trigger the TCP congestion control algorithms and TCP reduces its sending rate in an attempt to alleviate the congestion.

Two basic trends exist to improve TCP performance over lossy wireless channels (e.g., see [15] [17] and references therein). The first approach is to make TCP aware of non-congestion losses in such a way that it would be able to differentiate between packet losses due to network congestion along the path of the TCP flow and packet losses due to data corruption. In other words, the goal is to decouple congestion control and error control in TCP, thus the TCP sender can then avoid invoking the congestion control procedures when non-congestion losses occur. The second approach is to hide non-congestion losses from TCP and recover them locally. Techniques that use this approach, such as Automatic Repeat reQuest (ARQ) and Forward Error Correction (FEC), attempt to decrease the packet loss rate seen by TCP, avoiding unnecessary execution of the TCP congestion control algorithms and subsequent reduction in the sending rate. As long as it allows TCP to operate efficiently (to some extent) over wireless channels without any modification of the TCP/IP protocol stack and without requiring proxies between source and destination, this second approach is now widely adopted.

In ARQ, the ARQ receiver uses an error-detecting code to check if a received frame is in error or not. If the received frame is error-free, the ARQ sender is notified by sending a positive acknowledgment. If an error is detected, the ARQ receiver drops the received frame and notifies the ARQ sender via the feedback channel by sending a negative acknowledgment (NAK) or by the lack of a positive acknowledgment. In response to the NAK or if the ARQ sender does not receive the positive acknowledgment before the timeout, the ARQ sender retransmits the corresponding frame. There are three types of ARQ in use [91]: Stop-and-Wait (SW), Go-Back-N (GBN), and Selective Repeat (SR). SW is the simplest ARQ scheme and ensures that each transmitted frame is correctly received before sending the next one, whereas GBN and SR ARQ allow transmitting a number of frames continuously without waiting for an immediate acknowledgement. In GBN ARQ, when a certain frame is received in error, the ARQ sender retransmits all frames, starting from the incorrectly received one. According to SR ARQ, only incorrectly received frames should be retransmitted.

When the wireless channel conditions are relatively “bad” (e.g., due to fading, shadowing, mobility of mobile terminals, surrounding obstacles), ARQ introduces significant delays in data delivery due to multiple retransmissions. The maximum number of transmission attempts is an important configurable parameter of an ARQ scheme that affects performance of data transmission over wireless channels [91]. Setting this parameter to infinity assures completely reliable operation at the expense of increased delay and jitter. That is, frames are always delivered irrespective of the number of retransmissions it takes. It is important to note that talking about “completely reliable operation” here, we neglect the possibility that one of the devices fails or connectivity is entirely lost for some reason. In practice, the number of transmission attempts is usually limited allowing some frames to be lost. In this case, the service provided by the data link layer is considered to be semi-reliable: lost frames result in loss of the corresponding IP packets and, therefore, TCP segments encapsulated into these packets.

In contrast to ARQ, FEC uses certain codes that are designed to be self-correcting for errors introduced in transmission. That is, FEC adds redundant information to the transmitted data to recover them in case they are received in error. As a result, the destination host can detect and correct errors (but only within certain limits) without requiring a retransmission. In practical applications, two basic types of FEC codes are usually employed [92, p.6]: block codes and convolutional codes. As the name implies, block codes are used for coding blocks of data with a predetermined size, while convolutional codes operate continuously on streams of data. Popular and widely used block codes are Reed-Solomon (RS) and Bose-Chaudhuri-Hocquenghem (BCH) codes, named after their inventors.

Unfortunately, both ARQ and FEC have their own drawbacks. In particular, the drawback of FEC is that it constantly consumes bandwidth to transmit the redundant information. On the other hand, ARQ uses bandwidth mainly when frames are retransmitted but introduces variable delay (jitter) due to these retransmissions. The shortcomings of these error control techniques could be overcome if they are used in combination (also known as hybrid ARQ/FEC or hybrid ARQ, HARQ, for short) [93, p.60].

The increasing popularity of wireless networks has attracted a lot of attention to performance modeling of TCP in wireless environments. According to the used approach, analytical models can be split into two types. In the first type of models (e.g., [66] [94] [95]), wireless channel behavior is directly modeled at the IP layer, then an appropriate TCP model is applied to obtain the performance of a TCP connection running over the wireless channel. For instance, this approach was adopted in [96] to demonstrate that TCP Tahoe performs better in the presence of correlated losses. However, using this approach, we abstract away from the details of ARQ and FEC operation that are implemented at the lower layers of the protocol stack. As a result, evaluating TCP performance under different settings of these error control techniques becomes time consuming as it requires extensive measurements in order to parameterize the model. Moreover, while the vast majority of TCP analytical models require the end-to-end packet delay and the packet loss rate to be given, a model capable to predict TCP performance from primary network characteristics (such as the raw data rate of the bottleneck link and the bottleneck link buffer size) would be more usable.

The other way to evaluate the performance of a TCP connection running over a wireless channel is to use the so-called cross-layer modeling. In this type of models, TCP performance is derived from physical channel characteristics and other low-level parameters. Recently, a big effort has been done to model TCP performance as a function of adaptive modulation, the amount of FEC, the ARQ persistency, etc. In [97], the authors studied the combined effect of FEC, SR ARQ, and power management on TCP throughput. It was shown that increasing the transmission power, the FEC redundancy, and the number of transmission attempts allowed for the ARQ scheme always improves TCP performance by reducing the number of non-congestion losses. The performance of TCP over a wireless channel with hybrid SR ARQ/FEC and shared by a number of long-lived TCP flows was considered in [98]. Using the Bernoulli loss model to model non-congestion losses induced by wireless channel impairments, the authors provided results for different physical characteristics of the wireless channel and for different traffic loads. It was shown that the throughput of the wireless channel is an increasing function of the persistency of ARQ. In [99], the authors analyzed TCP throughput over a wireless channel with a NAK-based SR ARQ scheme in the presence of correlated errors at the physical layer. They observed that the TCP throughput mainly depends on the throughput limit of the wireless data link layer, the residual packet loss rate after data link layer retransmissions, and the correlation degree of residual losses. An analytical model, capturing the joint effect of SR ARQ and FEC on TCP performance over a correlated Rayleigh wireless channel, was presented in [100]. One of the main advantages of this model is that it uses wireless channel related characteristics (the signal to noise ratio and the Doppler shift) to parameterize the service process of the wireless channel, while the other models [97] [98] [99] require the frame error rate to be given.

The model presented here also adopts the cross-layer modeling approach. We derive the long-term steady-state throughput of a TCP SACK connection running over a wireless channel as a function of the bit error rate (BER), the normalized autocorrelation function (NACF) of bit error observations at lag 1, the strength of the FEC block code, the persistency of ARQ, the size of protocol data units (PDUs) at different layers, the raw data rate of the wireless channel, and the bottleneck link buffer size. The novelty of the proposed model is two-fold. Firstly, the model allows to evaluate the joint effect of the performance characteristics of the wireless channel and various implementation-specific parameters on TCP performance over both correlated and uncorrelated wireless channels. Secondly, the model explicitly takes into account a high correlation between the TCP window size and the RTT when a bottleneck link is dedicated to a single host [16], as well as packet losses due to both buffer overflow at the IP layer and an excessive number of transmission attempts at the data link layer in case of semi-reliable ARQ.

4.2 System Model and Assumptions

The system model is presented in Fig. 4.1. We consider a TCP connection between two hosts such that the last link on the end-to-end path from the TCP sender to the TCP receiver is a wireless channel with hybrid ARQ/FEC. Taking into account that backbone networks are highly overprovisioned [101], we assume that there is only one bottleneck in the system: the wireless channel. Since such scenario is common in today’s networks (e.g., Internet access over mobile networks), we do not refer to a particular wireless technology.

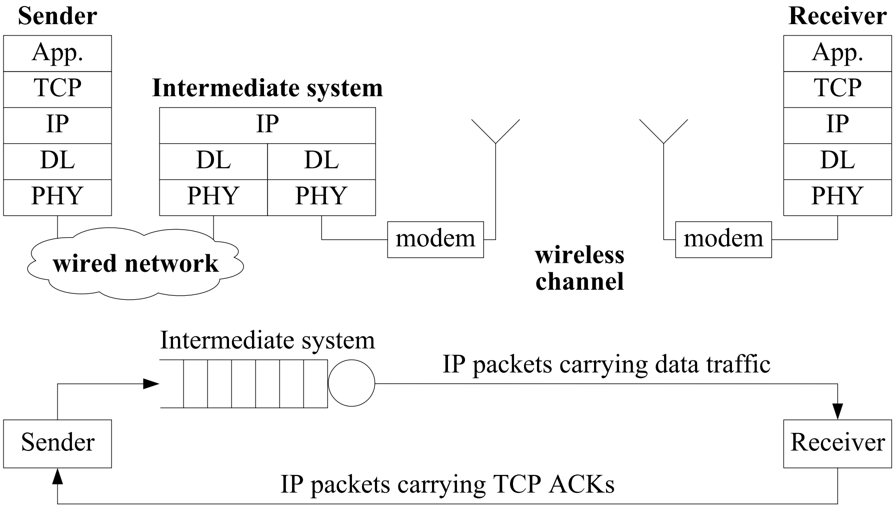


Fig. 4. 1 System model

The current status is that nearly all OSs support the SACK option (see Table 2.3 and [48]), so we assume that the TCP sender and the TCP receiver are both SACK-capable and use the SACK option under all permitted circumstances [42] [43] [44]. We consider a greedy application process which always has data to send, thus the TCP sender always sends full-sized segments (i.e., containing MSS bytes of application layer data) whenever the cwnd allows. It is assumed that the TCP receive buffer is sufficiently large, so the actual sending rate of the TCP connection is not limited by the rwnd size. These assumptions are justified for modern high-performance computers. As long as we are focusing on the performance of a single TCP connection running over a wireless channel, we do not consider here competing traffic effects and assume that application layer data are transmitted in one direction only: from the TCP sender to the TCP receiver (see Fig. 4.1).

Let the round-trip delay of the wired network be τ seconds, the raw data rate of the wireless channel be μ bits per second, and the buffer size of the intermediate system be B full-sized packets. Data packets are buffered at the IP layer of the intermediate system and passed one after another to the data link layer. The queue management algorithm is assumed to be FIFO Drop-Tail, so any packet arriving when the buffer is full will be lost. However, we do not model packet losses in the direction from the TCP receiver to the TCP sender and assume that TCP ACKs are always delivered to the TCP sender. In other words, packet losses happen only in the direction from the TCP sender to the TCP receiver. We believe that the impact of this omission is quite small because the cumulative nature of TCP ACKs ensures that the most recent ACK can cover all previously received data.

Between the IP and data link layers IP packets are segmented to a number of frames. Then FEC block coding is applied and the frames start to be transmitted. A data block composed of both data and FEC redundancy bits is called a codeword. For simplicity, we assume that the terms “frame” and “codeword” refer to the same entity and each frame consists of exactly one codeword. Additionally, we assume that exactly one bit is transmitted using a single channel symbol and use the terms “channel symbol” and “bit” interchangeably. Since extensions of the model to multiple codewords within a frame and multiple bits carried by a single channel symbol are straightforward, these assumptions are not fundamental and can be relaxed when needed.

The size of IP packets and frames carrying data traffic is assumed to be constant and equal to MTU bits and m bits, respectively. We denote the number of frames to which an IP packet is segmented as v. The process of encapsulation and segmentation of PDUs is shown in Fig. 4.2.

We assume that the ARQ receiver immediately sends back a feedback frame (carrying either a positive or a negative acknowledgement) for every incoming data frame it gets. This assumption allows to use a single model to capture different variants of ARQ, including the SW, GBN, and SR modes [102] [103]. Finally, we assume that the wireless channel in the reverse direction is completely reliable and that feedback frames are delivered instantaneously over the wireless channel. Indeed, feedback frames are usually small in size and well protected by a FEC code. Moreover, these assumptions were used in many studies and found to be appropriate for wireless links with a short propagation time [104] [105].

In [P5] [P6], we consider both types of ARQ operation: completely reliable and semi-reliable. When the ARQ scheme is completely reliable (i.e., perfectly-persistent), a frame is always delivered irrespective of the number of retransmissions it takes. However, if the number of data link retransmissions causes the RTT to exceed the current value of the TCP retransmission timeout (referred to as a delay spike in [16]), the TCP retransmission timer expires, leading to a spurious TCP timeout followed by unnecessary retransmission of the last window of data and invocation of the congestion control procedures. When the ARQ scheme is semi-reliable, the number of ARQ transmission attempts (including the original transmission and subsequent retransmissions) is limited to r. When a certain frame cannot be successfully delivered in r attempts, the ARQ sender drops this frame and the corresponding IP packet. The channel is then made free for the next packet waiting at the IP layer. Thus, we distinguish between packet losses occurring due to an excessive number of transmission attempts at the data link layer and packet losses resulting from buffer overflows at the IP layer. We denote the former ones as “non-congestion losses” and the latter as “congestion losses”.

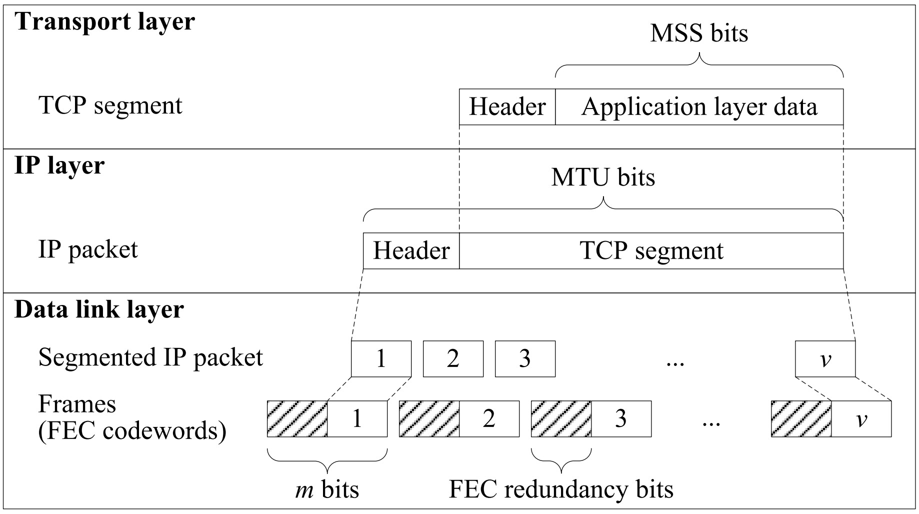


Fig. 4. 2 Encapsulation and segmentation

Table 4.1 summarizes the main notations used throughout this chapter.

Table 4. 1 Main notations used in the chapter

Notation	Meaning
$E[W_E]$	Bit error rate
$K_E(1)$	Normalized autocorrelation function of bit error observations at lag 1
$f(k)$	Probability function of the number of time slots the wireless channel is seized by transmitting an IP packet of MTU bits in size in case of completely reliable ARQ
$d(k)$	Probability function of the number of time slots the wireless channel is seized by transmitting an IP packet of MTU bits in size in case of semi-reliable ARQ
ε	Expected amount of time required to transmit an IP packet of MTU bits in size over the wireless channel in case of completely reliable ARQ (s)
δ	Expected amount of time the wireless channel is seized by transmitting an IP packet of MTU bits in size (regardless of its eventual fate) in case of semi-reliable ARQ (s)
MTU	IP maximum packet size (bits)
MSS	TCP maximum segment size (bits)
m	Frame size (bits)
v	Number of frames per IP packet of MTU bits in size
l	Number of errors per frame that can be corrected by the FEC scheme
r	Number of transmission attempts per frame in case of semi-reliable ARQ
B	Bottleneck link buffer size in IP packets of MTU bits in size
μ	Data rate of the wireless channel (bits/s)
τ	Round-trip delay of the wired network (s)
C	Average end-to-end path capacity in IP packets of MTU bits in size
p_C	Packet loss rate due to buffer overflow at the IP layer

p_L	Packet loss rate due to an excessive number of transmission attempts made at the data link layer in case of semi-reliable ARQ
b	Number of segments acknowledged by one ACK
$\overline{\mathrm{RTT}}$	Mean value of the total round-trip time (s)
$\overline{\mathrm{RTO}}$	Expected duration of a TCP retransmission timeout (s)
$\mathrm{RTO}_{\mathrm{min}}$	Minimum value of a TCP retransmission timeout (s)
x	By how much does $\overline{\mathrm{RTO}}$ exceed $\overline{\mathrm{RTT}}$

4.3 Cross-layer Model

Fig. 4.3 outlines the general structure of the model including the input/output parameters. The model is derived in two steps. At the first step, we consider the service process of the wireless channel and derive expressions for the following two parameters: the probability function of the number of time slots the wireless channel is seized by transmitting an IP packet and the packet loss probability due to an excessive number of transmission attempts made for one of its frames. Obviously, when the ARQ scheme is completely reliable, the first parameter corresponds to the probability function of the number of time slots required to transmit an IP packet over the wireless channel, while the second parameter is equal to zero. This is because the perfectly-persistent ARQ scheme will retransmit a corrupted frame an infinite number of times until the frame is successfully transmitted. When the ARQ scheme is semi-reliable, this probability function captures the duration of the packet transmission regardless of its eventual fate (i.e., whether a packet is successfully transmitted or dropped due to an excessive number of transmission attempts made for one of its frames). These parameters are used at the next step of the modeling, where we consider the performance of a TCP SACK connection running over the wireless channel. The wireless channel models for completely reliable and semi-reliable ARQ are presented in [P5] [P6], correspondingly.

Note that the cross-layer model has a modular structure, which provides the following advantages. Firstly, following the TCP/IP layering model makes the proposed model tractable. Secondly, modular design allows to extend the model by adding implementation- and protocol-specific details. Moreover, since the modules are independent of each other, they can be easily replaced or enhanced whenever is needed.

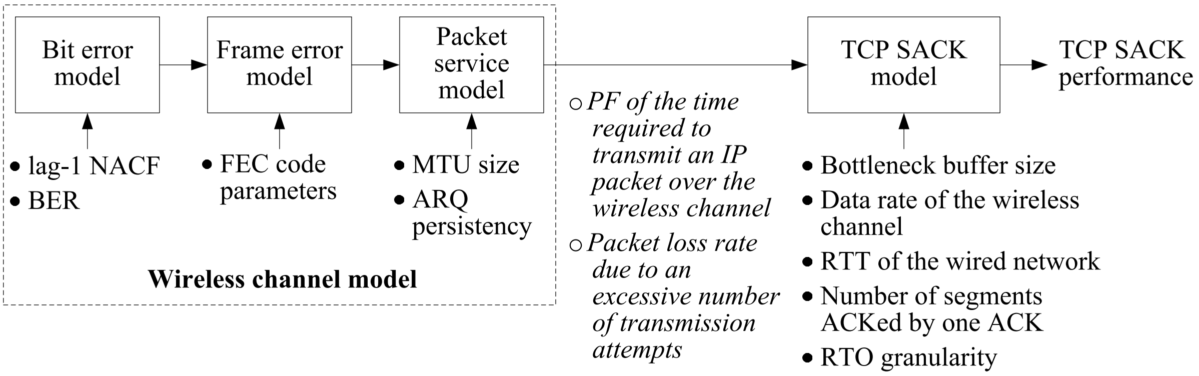

Bit error model
Frame error model
Packet service model
TCP SACK model
TCP SACK performance
• lag-1 NACF
• BER
• FEC code parameters
• MTU size
• ARQ persistency
Wireless channel model
○ PF of the time required to transmit an IP packet over the wireless channel
○ Packet loss rate due to an excessive number of transmission attempts
• Bottleneck buffer size
• Data rate of the wireless channel
• RTT of the wired network
• Number of segments ACKed by one ACK
• RTO granularity

Fig. 4. 3 Proposed cross-layer model

4.4 TCP SACK Model: Completely Reliable ARQ

In this section, we consider the evolution of a TCP SACK connection over a wireless channel with completely reliable ARQ/FEC and derive expressions for its long-term steady-state throughput and the spurious timeout probability. The developed model is based on the renewal-reward approach introduced in [65] [67].

To begin with, let us examine steady-state TCP SACK behavior in the absence of delay spikes caused by wireless channel impairments. Consider the system model depicted in Fig. 4.1. In long-distance and high-speed wired networks, the bit length of links is usually sufficiently large to allow multiple IP packets be on flight simultaneously. Here the bit length of a link stands for the number of bits present on the link at an instant of time when a stream of bits fully occupies this link [106, p.211]. The bit length of a link can be found as a product of the data rate of the link (in bits per second) and its propagation delay (in seconds). In terrestrial wireless networks, such as mobile networks, the bit length of wireless links is smaller than the length of a typical data packet. To illustrate this point, let us consider a 10 Mbit/s wireless channel where the distance between the transmitter and the receiver is 3 km. In free space, the propagation speed of radio waves is almost the same as that of light (approximately 300,000 km/s), so they would arrive at the receiver in about 0.01 ms. Then the bit length of the link is equal to 100 bits, which is much smaller than the length of a typical full-sized 1500-byte data packet (12,000 bits). At the same time, a 100 Mbit/s wired link, connecting the sender and the receiver located 200 km apart, has the bit length of 100,000 bits (note that wired media slow down signal propagation to about 200,000 km/s). Thus, in contrast to the wired network (see Fig. 4.1), where IP packets are usually sent back-to-back, the wireless channel cannot hold multiple packets at once. Moreover, assuming that the ARQ receiver immediately acknowledges the reception of every incoming data frame (either using a positive acknowledgement or a NAK), we get that all frames to which a previous packet was segmented should be successfully transmitted before starting a new transmission. Hence, at most one IP packet can be in transit on the wireless channel at a time.

Let the mean time required to transmit an IP packet over the wireless channel be ε seconds. Since the system bottleneck is solely determined by the wireless channel and at most one packet can be carried on the wireless channel at a time, the average network throughput is equal to MTU/ε bits per second, assuming there is always at least one packet waiting for transmission on the wireless channel (e.g., see Fig. 4.10b). Therefore, to fully utilize the wireless channel, the following requirements should be met. In the first place, we need to ensure that the bottleneck link buffer does not become empty to force the wireless channel to go idle. In addition, the TCP sender should not pause too long to make the buffer go empty. These conditions hold when the buffer size is at least as large as the end-to-end path capacity, while all packet losses can be detected within one RTT by the reception of three duplicate ACKs [18]. As it will be shown later, it does not practically limit the generality of our model and really holds in practice.

Let the average end-to-end path capacity (the wired network plus the wireless channel) be C full-sized IP packets (i.e., of MTU bits in size). To fully utilize the wireless channel, the buffer at the intermediate system should be sized as $B \geq C$, while ensuring that $B + C \geq 3$. This is caused by the fact that at least three duplicate ACKs are required to trigger the fast retransmit algorithm (if not enough duplicate ACKs arrive from the TCP receiver, the fast retransmit algorithm is never triggered and a timeout will be required to resend the lost segment) [29] [44]. Consequently, in case of $B \geq C$ and $B + C \geq 3$, the capacity of the wired network can be found as the bandwidth-delay product (expressed in packets of MTU size), while the capacity of the wireless channel is equal to just one packet:

$$C = \frac{1}{\mathrm{MTU}}\left(\frac{\mathrm{MTU}}{\varepsilon}\tau + \mathrm{MTU}\right) = \frac{\tau}{\varepsilon} + 1, \tag{4.1}$$

where ε is the mean time required to transmit an IP packet over the wireless channel.

To evaluate the order of magnitude of C, consider the difference in latency between wired and wireless networks (that is, between τ and ε). The one-way latency of a link can be defined as a sum of the propagation delay and the transmission delay. The propagation delay is the time required for a signal to propagate through the link. The signal propagates at the propagation speed, which depends on the physical medium of the link and is in the range of about 200,000 to 300,000 km/s. The propagation delay through the link can be calculated by dividing its length by the propagation speed. The transmission delay is the amount of time it takes to transmit a unit of data, such as a packet, at the data rate of the link. In turn, the transmission delay is equal to the packet size divided by the data rate of the link.

In high-speed wired networks, the transmission delay is generally negligible, whereas the propagation delay varies with distance. For instance, the transmission delay for a 1500-byte packet at 100 Mbit/s is about 0.12 ms. For fiber optics, the speed of signal propagation is about 2/3 of the speed of light in vacuum, which implies about 0.5 ms of the propagation delay for every 100 km. Typical wired wide area network (WAN) paths have the propagation delay on the order of tens of milliseconds (e.g., a USA coast-to-coast path has the propagation delay of approximately 25 ms). It should be emphasized that in practice the value of τ is larger than just the two-way propagation delay due to the queuing delay component caused by cross-traffic in the wired network. According to recent measurements [107] [108], RTTs of the vast majority of TCP connections are below 500 ms. Also note that τ will reach its maximum value (up to several hundreds of milliseconds) in case of one or more satellite hops along the path. However, here we do not consider the presence of both satellite links and a terrestrial wireless link on the same path of a given TCP connection and assume that the value of τ is mainly determined by the distance between the sender and the intermediate system (see Fig. 4.1).

The distance in wireless access networks is short enough compared to that in wired WANs because the distance does not exceed the cell size. Moreover, radio waves propagate at approximately the speed of light in vacuum. On the other hand, terrestrial wireless networks support much lower data rates than do today's wired networks. In addition, these data rates are inversely proportional to distance: as the data rate increases, the distance, at which this rate can be sustained, decreases. Besides, the implemented FEC and ARQ schemes can significantly increase the transmission delay, since adding FEC bits implies more bits to transmit, while retransmissions of erroneous frames delay delivery of correctly received frames. Consequently, a terrestrial wireless link introduces a larger transmission delay than most wired networks, but its propagation delay is negligibly small (in the range of a few microseconds). For example, assuming that the FEC code has the code rate of 1/2 and neglecting other types of overhead, the transmission delay for a 1500-byte packet at 10 Mbit/s is 2.4 ms. Suppose the distance between the base station and the mobile terminal is 3 km. Then the propagation delay is 0.01 ms. As a result of our calculations, we conclude that the values of τ and ε in (4.1) are commensurable quantities and, hence, the average end-to-end path capacity C is in the range of several to tens of full-sized packets. In practice, buffers are significantly overprovisioned in order to prevent packet losses due to buffer overflow [101]. Thus, our assumption that $B \geq C$ is quite reasonable.

The maximum number of IP packets that can be accommodated in the network is approximately equal to $C + B$, assuming that there are C packets in flight and the buffer at the intermediate system is fully occupied. Since $C \geq 2$ and $B \geq C$, it implies that $C + B \geq 4$. In this case, the TCP connection experiences periodic packet losses: each time the cwnd exceeds the maximum number of packets that can be accommodated in the network, the last packet in the transmitted window is dropped due to buffer overflow at the intermediate system. According to the TCP sliding window algorithm, after the lost segment, $C+B-1$ new segments are sent, triggering duplicate ACKs. As long as $C+B\geq 4$ and at least three duplicate ACKs are required in order to trigger a fast retransmission without the need to wait for a timeout event, the TCP sender receives enough duplicate ACKs to detect the segment loss. Since any lost segment can be recovered within a single RTT by using the SACK-based loss recovery algorithm [44], we neglect the details of the loss recovery phase as having only a minor effect on the long-term steady-state performance. Thus, a congestion avoidance phase starts after a segment loss is detected via three duplicate ACKs. Then the current value of the cwnd is set to approximately $(C+B)/2$ and the congestion avoidance phase begins. The TCP receiver sends one ACK for every b segment it gets, so the cwnd increases linearly with a slope of $1/b$ segments per RTT until $\text{cwnd}=C+B$ (see Fig. 4.4). The factor b depends on the acknowledgement policy: as specified in [28], TCP should use the delayed ACK algorithm, sending one ACK for every other received segment $(b=2)$, unless the delayed ACK timer expires; however, it is also allowed to send one ACK for every received segment $(b=1)$. The next additive increase of the cwnd leads to a new buffer overflow and the current congestion avoidance phase ends. Hence, considering the evolution of the cwnd between $(C+B)/2$ and $C+B$ (see Fig. 4.4), we get:

$$E\left[A^{CA0}\right]=b\left(\frac{C+B}{2}\right),\quad E\left[Y^{CA0}\right]=\frac{3b}{2}\left(\frac{C+B}{2}\right)^2,\quad p_C=\frac{1}{E\left[Y^{CA0}\right]},\tag{4.2}$$

where $E\left[A^{CA0}\right]$ is the expected duration of a congestion avoidance phase CA0 in rounds; $E\left[Y^{CA0}\right]$ is the expected number of packets sent during this phase; p_C is the packet loss rate due to buffer overflow at the IP layer.

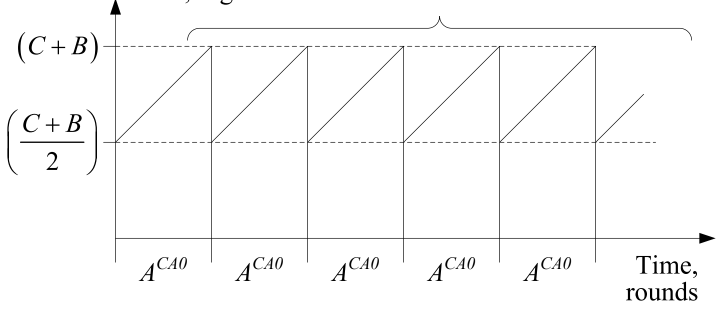
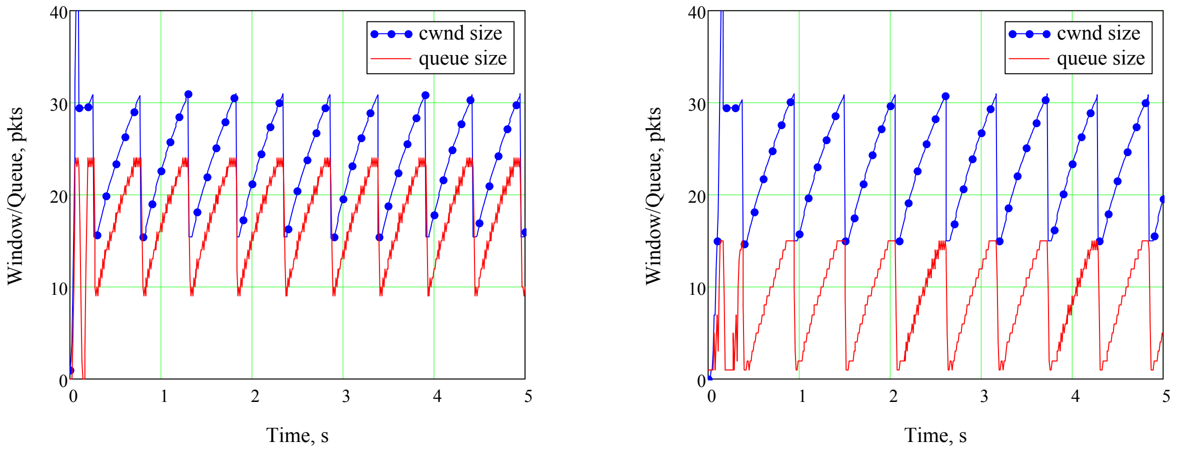


Fig. 4. 4 TCP SACK window evolution in the absence of delay spikes

In the considered scenario (Fig. 4.1), the RTT is given by three components: the round-trip delay τ of the wired network, the queuing delay at the intermediate system, and the round-trip delay of the wireless channel. As long as the wireless channel is the only bottleneck on the path of the TCP connection (i.e., the wired network has sufficient bandwidth and low enough total load, so it never encounters any queues), the value of τ is assumed to be static over time and mainly determined by the geographical spread of the wired network. Thus, to compute the mean value of the RTT, we must determine the mean queuing delay at the intermediate system and the mean round-trip delay of the wireless channel.

We begin by deriving an expression for the mean queue size $E[R]$. Note that the buffer occupancy tends to be periodic over time, following the saw-tooth TCP SACK window evolution. Fig. 4.5 shows the window size and buffer occupancy dynamics obtained using ns-2.



a) $B > C$

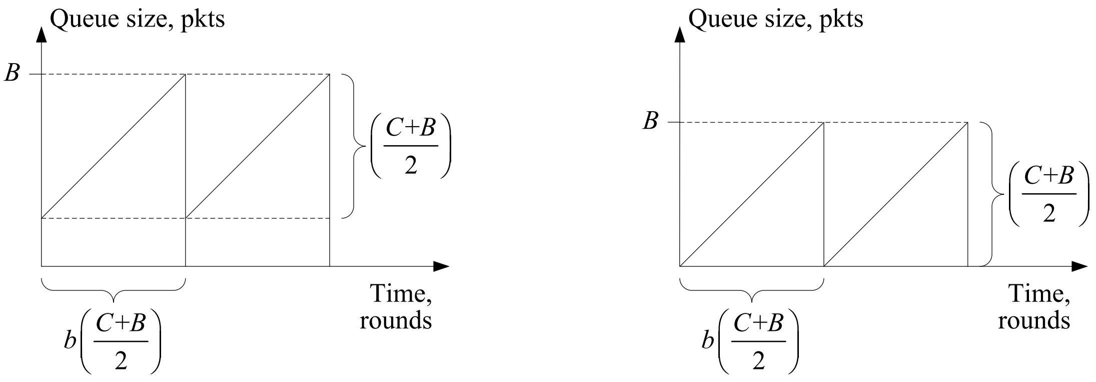
b) $B = C$

Fig. 4. 5 TCP SACK window evolution and buffer occupancy

One may note that the queue size depends on the current value of the cwnd and the ratio between the buffer size B and the average end-to-end path capacity C (Fig. 4.5). Taking into account that in the absence of delay spikes the minimum value of the cwnd is $(C+B)/2$ and the maximum is $C+B$ (see Fig. 4.4 and Fig. 4.6), we get the mean queue size during a congestion avoidance phase CA0 as

$$E\left[R^{CA0}\right]=\frac{\frac{b}{2}\left(\frac{C+B}{2}\right)^2+b\left(\frac{C+B}{2}\right)\left(B-\frac{C+B}{2}\right)}{b\left(\frac{C+B}{2}\right)}=\frac{3B-C}{4}. \tag{4.3}$$

The mean queuing delay can be obtained by multiplying the mean queue size $E\left[R\right]$ by the mean time ε required to transmit an IP packet over the wireless channel. In the absence of delay spikes, $E\left[R\right]=E\left[R^{CA0}\right]$.



a) $B > C$ b) $B = C$

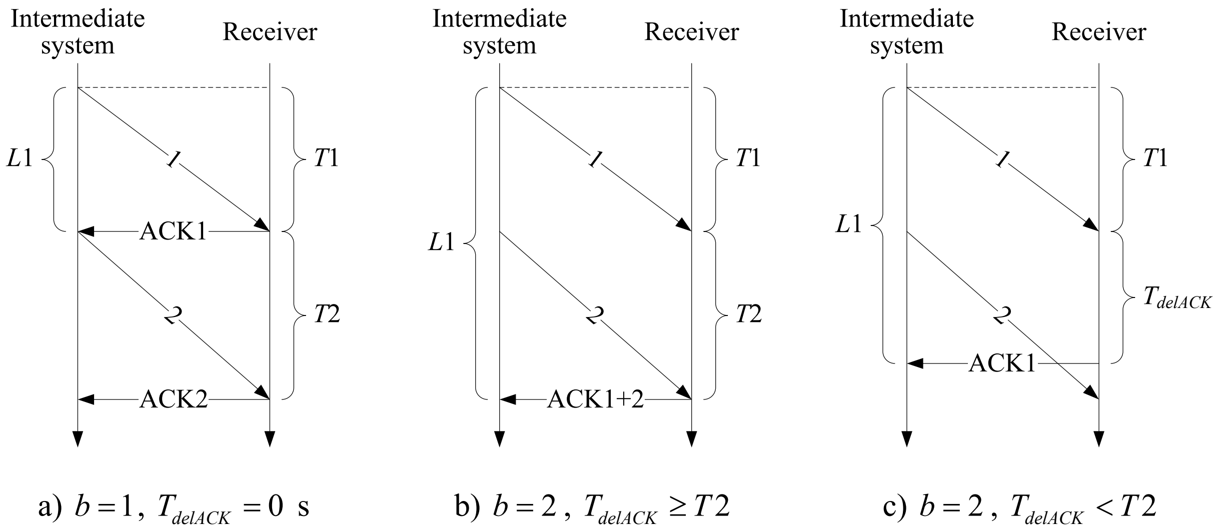
Fig. 4. 6 Queue size as a function of the ratio between the buffer size and the average end-to-end path capacity

The mean round-trip delay of the wireless channel can be found as a sum of the mean time required to transmit an IP packet over the wireless channel and the amount of time needed to get an ACK segment back (the last term depends on whether the delayed ACK algorithm is enabled or disabled). Recall that we neglect the transmission delay in the reverse direction. This is justified, since the size of IP packets carrying TCP ACKs is much smaller than the size of IP packets carrying TCP data segments (typically, a 40-byte ACK packet versus a 1500-byte data packet). Moreover, we do not take into account the propagation delays in both directions, because, as it was mentioned earlier, these delays are negligibly small compared to the time required to transmit a data packet over the wireless channel.

The delayed ACK algorithm determines how the TCP receiver sends ACKs: whether an ACK is sent for every received segment, or whether an ACK is delayed until either the next segment arrives or the delayed ACK timeout expires (i.e., after T_{delACK} seconds). The round-trip delay of the wireless channel as a function of the delayed ACK algorithm and the inter-packet arrival time is shown in Fig. 4.7, where $T1$ and $T2$ are the time intervals required to successfully transmit all frames to which packets 1 and 2 were segmented, respectively. Then the mean round-trip delay of the wireless channel can be obtained as

$$L=\begin{cases} b\varepsilon, & \varepsilon \le T_{delACK} \text{ or } b=1, \\ \varepsilon + T_{delACK}, & \varepsilon > T_{delACK}, \end{cases} \tag{4.4}$$

where $b=1$, if the TCP receiver immediately acknowledges every segment it gets; or $b=2$, if an ACK is sent for every other segment unless the delayed ACK timer expires.



a) $b=1$, $T_{delACK}=0$ s b) $b=2$, $T_{delACK} \geq T2$ c) $b=2$, $T_{delACK} < T2$

Fig. 4. 7 Round-trip delay of the wireless channel

Finally, the mean value of the RTT is given by

$$\overline{\text{RTT}} = \tau + E\left[R\right]\varepsilon + L, \tag{4.5}$$

where $E\left[R\right]$ is the mean queue size.

Now let us consider the effect of delay spikes on the long-term steady-state throughput of a TCP SACK connection. Delay spikes can be caused by a number of factors, including handovers, high priority traffic, etc. In the considered scenario, the completely reliable ARQ scheme can cause a sudden delay due to transmission errors on the wireless channel and a large number of subsequent retransmissions of the incorrectly received frames, resulting in a TCP spurious timeout. It is worthwhile to note that improving TCP performance in the presence of abrupt delay changes is an active research area. A number of algorithms have been proposed to avoid or detect spurious timeouts and to recover from them (e.g., [109] [110] [111] [112] [33]).

Consider the transmission of packets during the initial slow start phase when the delayed acknowledgement algorithm is disabled and the receiver acknowledges every segment it gets (Fig. 4.8). For simplicity of illustration, let us assume that the IW size is equal to two full-sized segments and the ssthresh is sufficiently high. Once the TCP connection has been established, the TCP sender begins sending data packets. When the TCP sender transmits the first packet, it starts the TCP retransmission timer so that it will expire after RTO seconds. After approximately $\tau/2$ seconds the first packet arrives at the intermediate system. Since the wireless channel is idle and the buffer is empty, the incoming packet will be serviced immediately: at the data link layer it will be segmented to a number of frames and these frames will be transmitted one after another over the wireless channel. Packet 2 arriving at the intermediate system will find the wireless channel busy and will be buffered for later transmission. Let $T1$ be the time required to successfully transmit all frames to which packet 1 was segmented. The TCP receiver then replies with an ACK segment (ACK1). As soon as the wireless channel becomes idle, the packet service process starts all over again: packet 2 is passed to the data link layer for segmentation and subsequent transmission over the wireless channel. Since we assume that the wireless channel in the reverse direction is completely reliable and that the feedback is almost instantaneous, it will take only $\tau/2$ seconds to deliver ACK1 to the TCP sender. The next ACK (ACK2) will arrive after $T2$ seconds, where $T2$ is the time required to successfully transmit all frames to which packet 2 was segmented. ACK3 will arrive at the TCP sender after $T3$ seconds and so on.

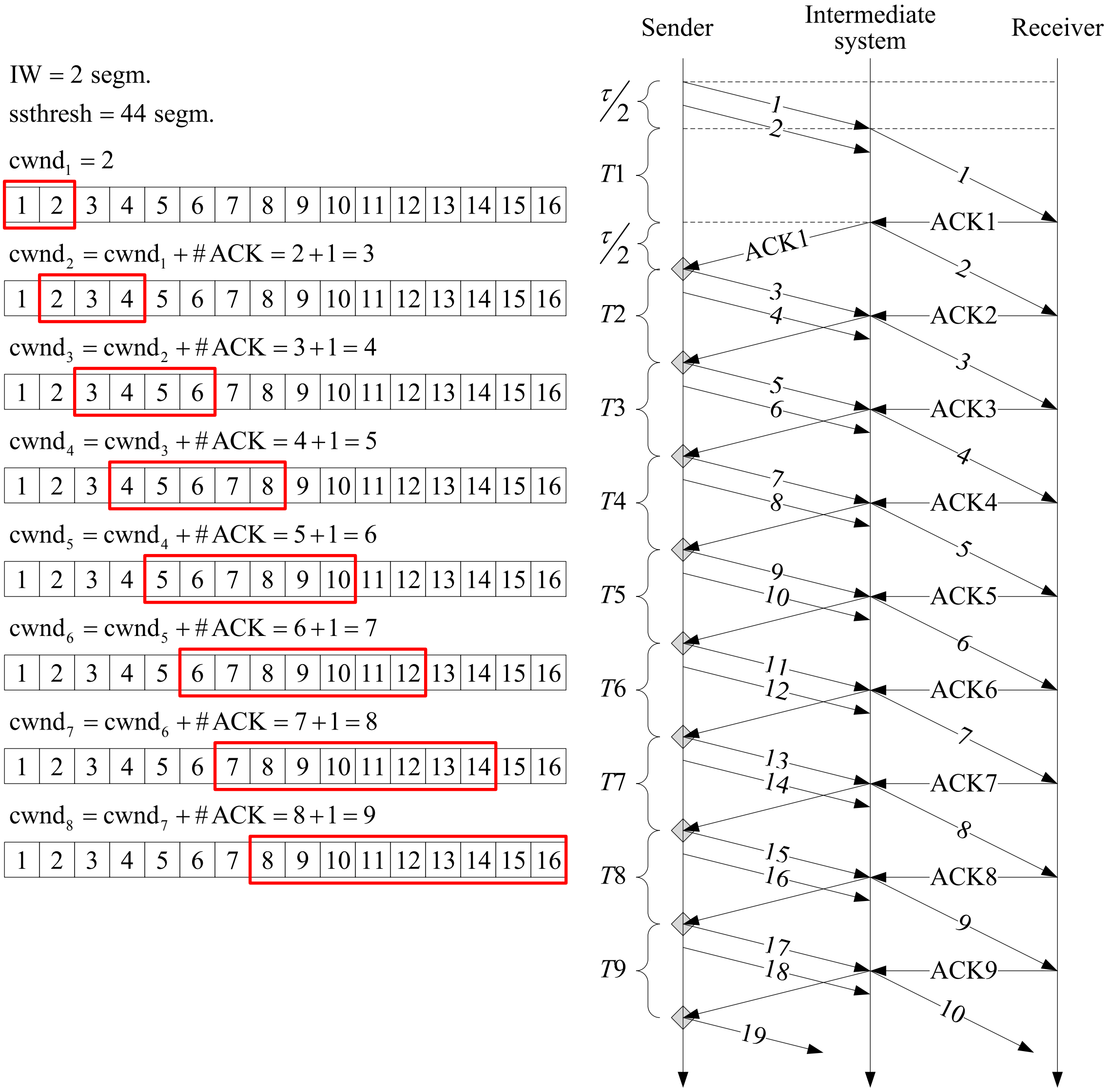
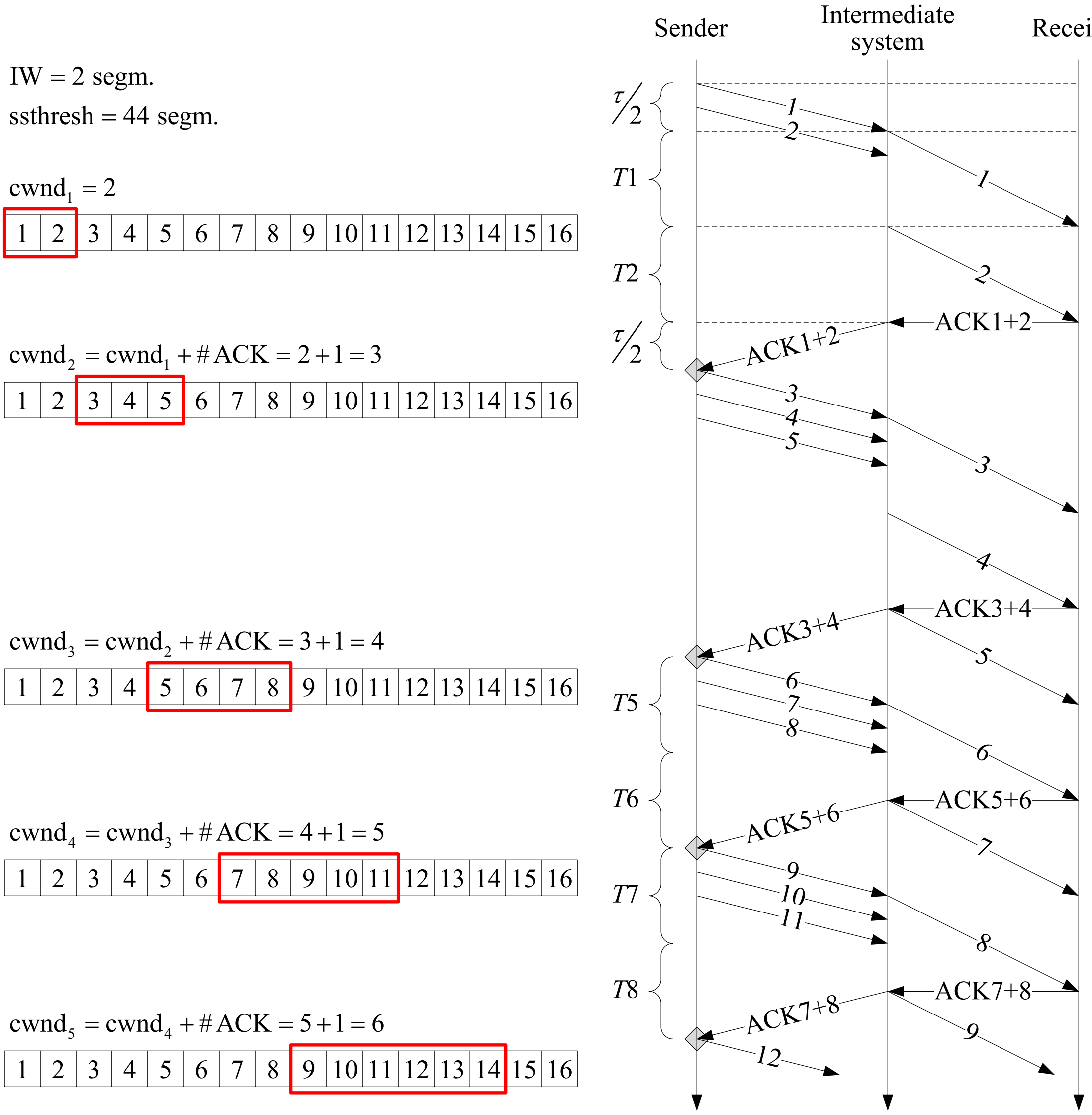

IW = 2 segm.
ssthresh = 44 segm.
cwnd$_1$ = 2
cwnd$_2$ = cwnd$_1$ + #ACK = 2 + 1 = 3
cwnd$_3$ = cwnd$_2$ + #ACK = 3 + 1 = 4
cwnd$_4$ = cwnd$_3$ + #ACK = 4 + 1 = 5
cwnd$_5$ = cwnd$_4$ + #ACK = 5 + 1 = 6
cwnd$_6$ = cwnd$_5$ + #ACK = 6 + 1 = 7
cwnd$_7$ = cwnd$_6$ + #ACK = 7 + 1 = 8
cwnd$_8$ = cwnd$_7$ + #ACK = 8 + 1 = 9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Sender
Intermediate system
Receiver
τ/2
T1
T2
T3
T4
T5
T6
T7
T8
T9
ACK1
ACK2
ACK3
ACK4
ACK5
ACK6
ACK7
ACK8
ACK9
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

Fig. 4. 8 End-to-end transmission of packets, $\mathrm{MSS} = 1460$ bytes, $\mathrm{ssthresh} = 65{,}535$ bytes, $b = 1$

Note that these arrivals are separated by the time required to transmit a corresponding IP packet over the wireless channel. When the delayed ACK algorithm is enabled, the TCP receiver acknowledges every other incoming segment or delays an ACK for T_{delACK} seconds (see Fig. 4.9). In this case, the mean inter-ACK gap is given by (4.4). In accordance to [45], when an ACK is received that acknowledges new data, the TCP retransmission timer should be restarted so that it will expire after RTO seconds (see step 5.3 in [45]). Thus, every time a new ACK arrives, the TCP retransmission timer will be restarted (denoted by grey diamonds in Fig. 4.8 and Fig. 4.9). Therefore, the only possibility for a TCP spurious timeout to occur is to transmit an IP packet over the wireless channel within RTO seconds or more.

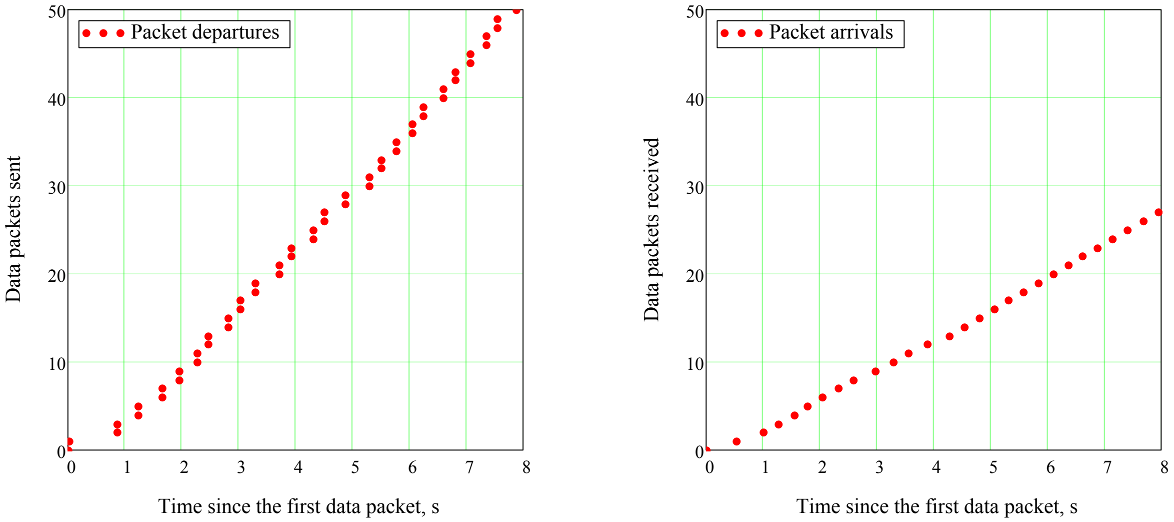


Sender
Intermediate system
Receiver
IW = 2 segm.
ssthresh = 44 segm.
$\text{cwnd}_1 = 2$
$\text{cwnd}_2 = \text{cwnd}_1 + \#\text{ACK} = 2+1 = 3$
$\text{cwnd}_3 = \text{cwnd}_2 + \#\text{ACK} = 3+1 = 4$
$\text{cwnd}_4 = \text{cwnd}_3 + \#\text{ACK} = 4+1 = 5$
$\text{cwnd}_5 = \text{cwnd}_4 + \#\text{ACK} = 5+1 = 6$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
$\tau/2$
$T1$
$T2$
$T5$
$T6$
$T7$
$T8$
ACK1+2
ACK3+4
ACK5+6
ACK7+8
1
2
3
4
5
6
7
8
9
10
11
12

Fig. 4. 9 End-to-end transmission of packets,

$\text{MSS} = 1460$ bytes, $\text{ssthresh} = 65,535$ bytes, $b = 2$

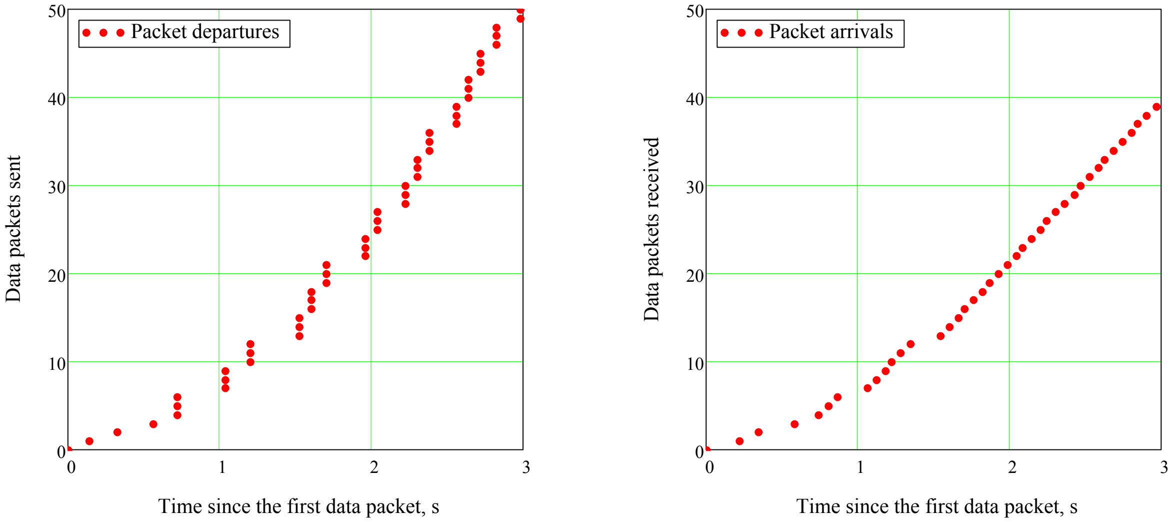
In order to illustrate and validate the above examples, we used Iperf [113], a standard tool for measuring network performance, to generate TCP flows between a server host in a high-speed wired domain (100 Mbit/s Ethernet) and a client host connected via a wireless last-hop link (EDGE). In our experiments, we used Wireshark to capture packet traces at both sender and receiver. Fig. 4.10 and Fig. 4.11 depict the obtained results for immediate and delayed ACKs, respectively. For the sake of simplicity, we do not show the delays caused by the three-way handshaking process.



a) The sending side

b) The receiving side

Fig. 4. 10 Time-sequence graph of a TCP connection over EDGE,
MSS = 1460 bytes, ssthresh = 65,535 bytes, $b = 1$



a) The sending side

b) The receiving side

Fig. 4. 11 Time-sequence graph of a TCP connection over EDGE,
MSS = 1460 bytes, ssthresh = 65,535 bytes, $b = 2$

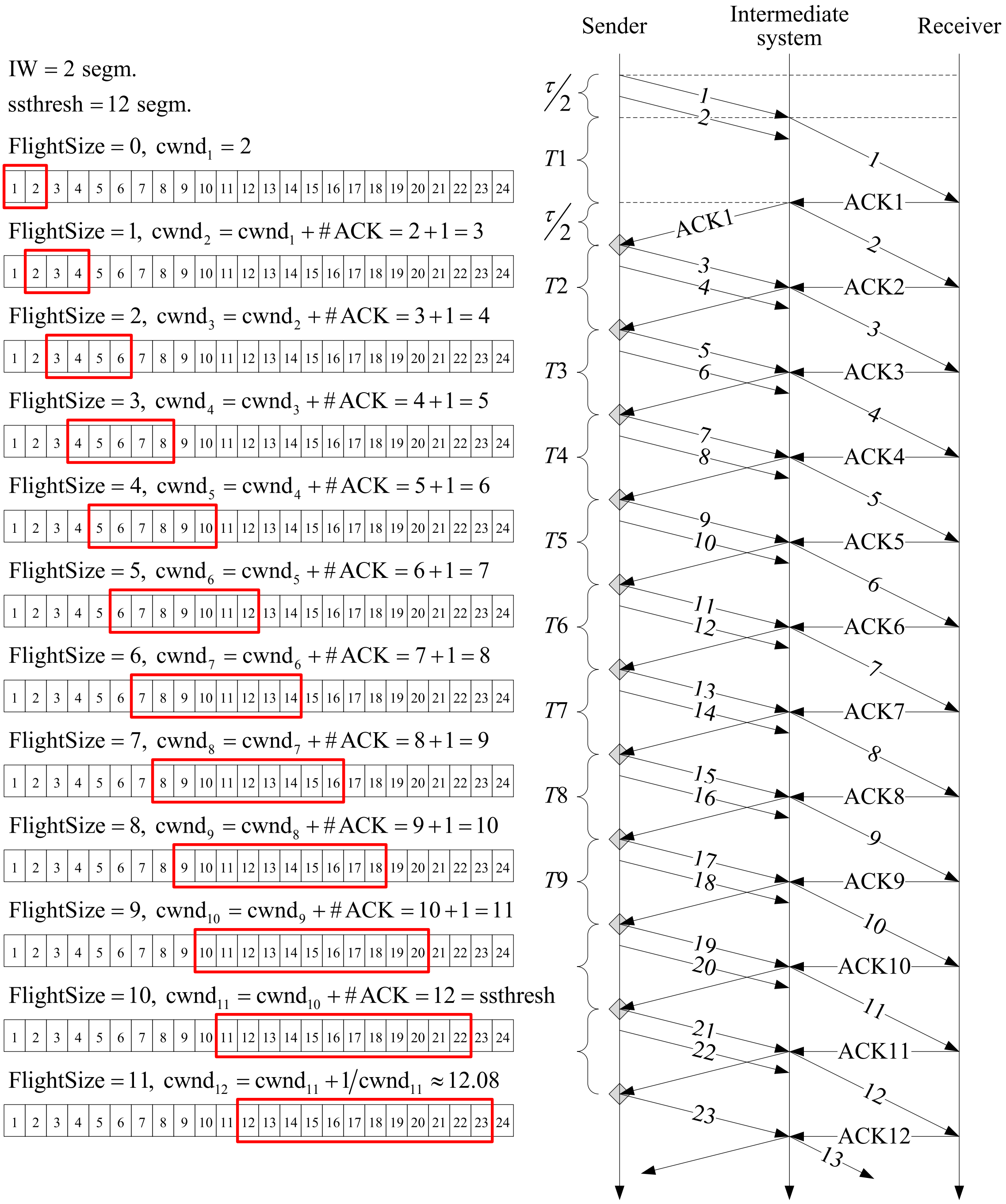
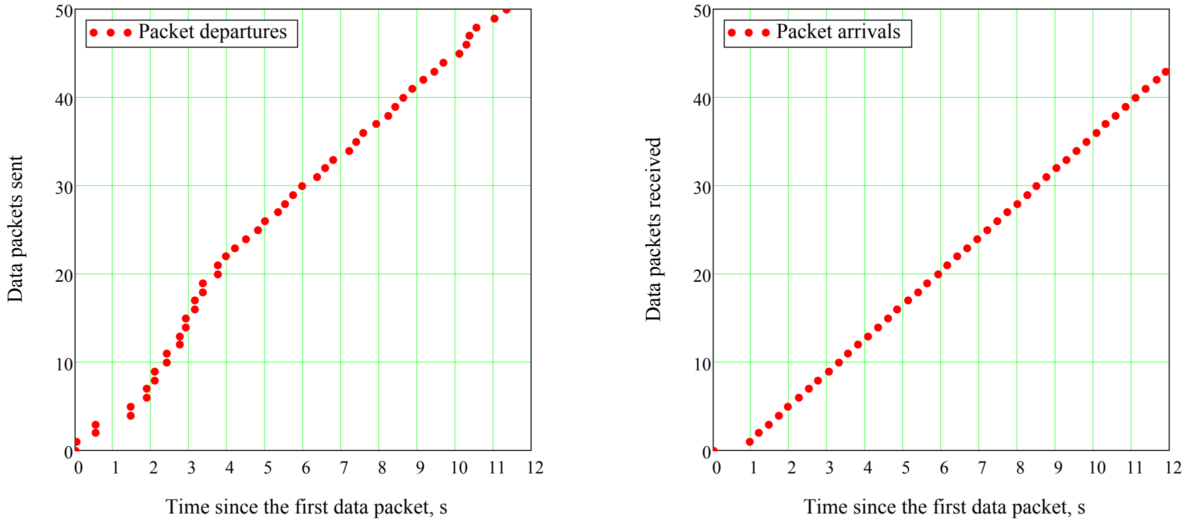

IW = 2 segm.
ssthresh = 12 segm.
FlightSize = 0, cwnd$_1$ = 2
FlightSize = 1, cwnd$_2$ = cwnd$_1$ + #ACK = 2 + 1 = 3
FlightSize = 2, cwnd$_3$ = cwnd$_2$ + #ACK = 3 + 1 = 4
FlightSize = 3, cwnd$_4$ = cwnd$_3$ + #ACK = 4 + 1 = 5
FlightSize = 4, cwnd$_5$ = cwnd$_4$ + #ACK = 5 + 1 = 6
FlightSize = 5, cwnd$_6$ = cwnd$_5$ + #ACK = 6 + 1 = 7
FlightSize = 6, cwnd$_7$ = cwnd$_6$ + #ACK = 7 + 1 = 8
FlightSize = 7, cwnd$_8$ = cwnd$_7$ + #ACK = 8 + 1 = 9
FlightSize = 8, cwnd$_9$ = cwnd$_8$ + #ACK = 9 + 1 = 10
FlightSize = 9, cwnd$_{10}$ = cwnd$_9$ + #ACK = 10 + 1 = 11
FlightSize = 10, cwnd$_{11}$ = cwnd$_{10}$ + #ACK = 12 = ssthresh
FlightSize = 11, cwnd$_{12}$ = cwnd$_{11}$ + 1/cwnd$_{11}$ ≈ 12.08
Sender
Intermediate system
Receiver
τ/2
T1
T2
T3
T4
T5
T6
T7
T8
T9
ACK1
ACK2
ACK3
ACK4
ACK5
ACK6
ACK7
ACK8
ACK9
ACK10
ACK11
ACK12

Fig. 4. 12 End-to-end transmission of packets, MSS = 1460 bytes, ssthresh = 17,520 bytes, $b = 1$

Examining the time-sequence graphs, we can make the following observations. TCP, being a window-based protocol, sends packets into the network in bursts. This is especially noticeable in the slow start phase (see Fig. 4.10a and Fig. 4.11a). However, when the bottleneck link is saturated with incoming traffic and is fully utilized, data packets arriving at the receiver tend to be well distributed in time (see Fig. 4.10b and Fig. 4.11b). Note that the gaps between packet arrivals at the beginning of the initial slow start phase in Fig. 4.11b are due to the fact that the

TCP sending rate at that time is less than that required to keep the bottleneck link full (i.e., to "fill the pipe"). Thus, in case of a single greedy source under steady-state conditions, the interarrival time distribution at the receiver is completely determined by the service process of the wireless channel. Since packet arrivals are spread in time, the corresponding ACKs will be sparsely issued as well. In turn, this smoothes out the transmission of packets (see Fig. 4.12 and Fig. 4.13).

A common approach to estimate the long-term steady-state throughput of a single TCP source is to compute the ratio between the expected transmission window size and the mean RTT. This concept is used, for instance, in [65] [67] [94] [95] [P2] [P3] [P4]. However, armed with a wireless channel model (such as the one introduced in [P5]) and assuming that the wireless channel is saturated with incoming traffic, we can define the TCP steady-state throughput as the ratio between the MSS and the mean time required to transmit an IP packet over the wireless channel. At this point, we only need to know the mean RTT to calculate the expected duration of the RTO and, thereby, to estimate the spurious timeout probability.



a) The sending side

b) The receiving side

Fig. 4. 13 Time-sequence graph of a TCP connection over EDGE, $\text{MSS} = 1460$ bytes, $\text{ssthresh} = 17{,}520$ bytes, $b = 1$

To calculate the mean duration of a TCP retransmission timeout, we use the following approximation. Commonly, TCP implementations use a coarse-grained retransmission timer, having granularity of 500 ms. Moreover, the current standard [45] requires that whenever the RTO is computed, if it is less than 1 second then it should be rounded up to 1 second. At the same time, some implementations use a fine-grained retransmission timer and do not follow the requirements of [45] by allowing, for example, the minimum limit of 200 ms [90]. Thus, we obtain the expected duration of the RTO as

$$\overline{\mathrm{RTO}} = \max\left(x\overline{\mathrm{RTT}}, \mathrm{RTO}_{\min}\right), \tag{4.6}$$

were x, $x > 1$, relates to the granularity of the TCP retransmission timer; $\mathrm{RTO}_{\min}$ is the minimum value of a TCP retransmission timeout; $\overline{\mathrm{RTT}}$ is from (4.5).

Since $x > 1$, $E\left[R\right] \geq 1$, and $L \geq \varepsilon$, we get that the expected duration of the RTO is at least several times (denoted as M) larger than ε :

$$\overline{\mathrm{RTO}} = \max\left(x\left(\tau + E\left[R\right]\varepsilon + L\right), \mathrm{RTO}_{\min}\right) = M\varepsilon, \quad M > 2. \tag{4.7}$$

Then the spurious timeout probability Q' can be obtained as the probability that the amount of time required to transmit an IP packet over the wireless channel and to get an ACK segment back will be at least M times larger than ε :

$$Q' = \begin{cases} \sum\limits_{k=\left\lceil \frac{\mu}{m} M\varepsilon \right\rceil}^{\infty} f\left(k\right), & b = 1, \\ \sum\limits_{k=\left\lceil \frac{\mu}{m} M\varepsilon \right\rceil}^{\infty} f_2\left(k\right), & b = 2,\ \varepsilon \leq T_{delACK}, \\ \sum\limits_{k=\left\lceil \frac{\mu}{m} M\varepsilon \right\rceil}^{\infty} f\left(k + \left\lceil \frac{\mu}{m} T_{delACK} \right\rceil\right), & b = 2,\ \varepsilon > T_{delACK}, \end{cases} \tag{4.8}$$

where $\lceil \ \rceil$ is the ceiling function; μ/m is the number of slots per second; $f\left(k\right)$, $k = v, v+1, \ldots$, is the probability function of the number of time slots required to transmit an IP packet over the wireless channel (see [P5] for details); $f_2\left(k\right)$ is the convolution of two functions, $f_2\left(k\right) = f\left(k\right) * f\left(k\right)$.

Thus, each data packet may be excessively delayed with probability Q' due to a large number of transmission attempts at the data link layer, causing a TCP spurious timeout. On the other hand, it may be delivered in time (i.e., before the TCP retransmission timer expires) with probability $1 - Q'$. Similarly to [94], we consider the evolution of a TCP SACK connection as a sequence of cycles, where a cycle is a period between two consecutive delay spikes (see Fig. 4.14). Then the expected number of packets sent during a cycle can be defined as

$$E\left[Y\right] = \sum_{k=1}^{\infty} \left(1 - Q'\right)^{k-1} Q'k = \frac{1}{Q'}. \tag{4.9}$$

The expected duration (in seconds) of a delay spike (DS) can be computed as

$$E\left[Z^{DS}\right]=\begin{cases}\dfrac{m}{\mu}\displaystyle\sum_{k=\left\lceil \frac{\mu}{m}M\varepsilon\right\rceil}^{\infty} f(k)k, & b=1,\\ \dfrac{m}{\mu}\displaystyle\sum_{k=\left\lceil \frac{\mu}{m}M\varepsilon\right\rceil}^{\infty} f_2(k)k, & b=2,\ \varepsilon\le T_{delACK},\\ \dfrac{m}{\mu}\displaystyle\sum_{k=\left\lceil \frac{\mu}{m}M\varepsilon\right\rceil}^{\infty} f\left(k+\left\lceil\dfrac{\mu}{m}T_{delACK}\right\rceil\right)k, & b=2,\ \varepsilon> T_{delACK},\end{cases} \tag{4.10}$$

where m/μ is the duration of a time slot.

When the variability in the wireless channel quality introduces a sudden delay in the service process of an IP packet, all the subsequent transmissions up to the end of the delay spike will be delayed as well. After TCP retransmission timer expiration, the TCP sender retransmits the first unacknowledged segment, and in the absence of any feedback from the TCP receiver it will continue trying to deliver this segment as specified in [45]. When the delay spike ends, the ACK for the original transmission returns to the TCP sender. On receipt of this ACK after the wireless channel outage, the TCP sender mistakenly interprets it as acknowledging the recently retransmitted segment and enters the slow start phase with unnecessary retransmission of other outstanding segments in the Go-Back-N strategy. Since none of the outstanding segments was actually lost, all these segments get retransmitted unnecessarily. These unnecessarily retransmitted segments arrive as duplicate at the TCP receiver, which in turn triggers a series of duplicate ACKs. In the absence of the SACK option or timestamps, a duplicate ACK carries no information to identify the segment that triggered that ACK, so TCP is unable to distinguish between a duplicate ACK that results from a lost segment and a duplicate ACK that results from an unnecessary retransmission of a segment that had already been received at the destination.

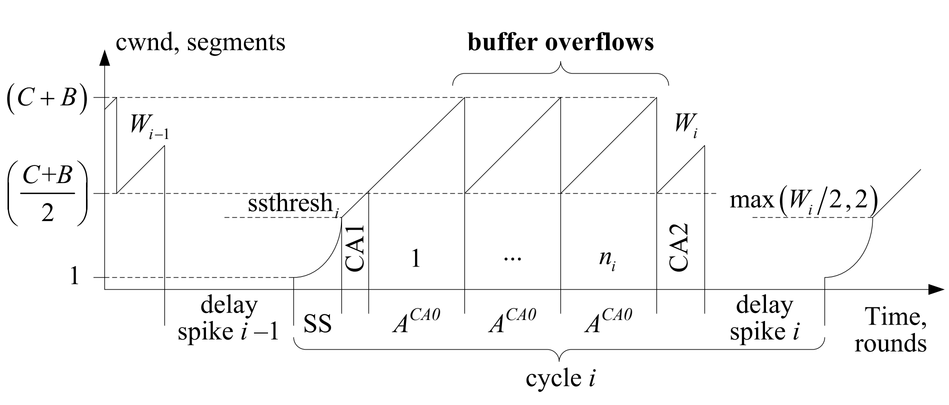


Fig. 4. 14 TCP SACK window evolution in the presence of delay spikes

In early TCP implementations, spurious timeouts usually lead to unnecessary multiple fast retransmits and, hence, multiple reductions of the cwnd. As was demonstrated in [114], TCP SACK is robust against false fast retransmits, since the SACK option with the D-SACK extension allows the TCP sender to infer when it has unnecessarily retransmitted a segment. Therefore, we assume that the slow start phase continues until the cwnd reaches the ssthresh and then a new congestion avoidance phase begins (see Fig. 4.14).

Let W_{i-1} denote the window size when delay spike $i-1$ occurs. After TCP retransmission timer expiration, the current values of the ssthresh and the cwnd will be set as $\mathrm{ssthresh}_i = \max\left(W_{i-1}/2, 2\right)$ and $\mathrm{cwnd} = 1$, respectively. Supposing that delay spikes are less frequent than packet losses due to buffer overflow ($p_C > Q'$), we can safely assume that the random variable W_i is uniformly distributed from $\left(C+B\right)/2$ to $C+B$. Hence

$$E\left[W\right] = \frac{1}{2}\left(\frac{C+B}{2} + C + B\right) = \frac{3}{2}\left(\frac{C+B}{2}\right), \quad E\left[\mathrm{ssthresh}\right] = \max\left(\frac{E[W]}{2}, 2\right). \tag{4.11}$$

The expected durations (in rounds) of phases CA1 and CA2 (see Fig. 4.14) can be found as

$$\begin{aligned} E\left[A^{CA1}\right] &= b\left(\frac{C+B}{2} - \frac{E\left[W\right]}{2}\right) = \frac{b}{4}\left(\frac{C+B}{2}\right), \\ E\left[A^{CA2}\right] &= b\left(E\left[W\right] - \frac{C+B}{2}\right) = \frac{b}{2}\left(\frac{C+B}{2}\right), \end{aligned} \tag{4.12}$$

and the expected number of segments sent during these phases can be defined as

$$\begin{aligned} E\left[Y^{CA1}\right] &= b\left(\frac{C+B}{2} - \frac{E\left[W\right]}{2}\right)\frac{E\left[W\right]}{2} + \frac{b}{2}\left(\frac{C+B}{2} - \frac{E\left[W\right]}{2}\right)^2 = \frac{7b}{32}\left(\frac{C+B}{2}\right)^2, \\ E\left[Y^{CA2}\right] &= b\left(E\left[W\right] - \frac{C+B}{2}\right)\left(\frac{C+B}{2}\right) + \frac{b}{2}\left(E\left[W\right] - \frac{C+B}{2}\right)^2 = \frac{5b}{8}\left(\frac{C+B}{2}\right)^2. \end{aligned} \tag{4.13}$$

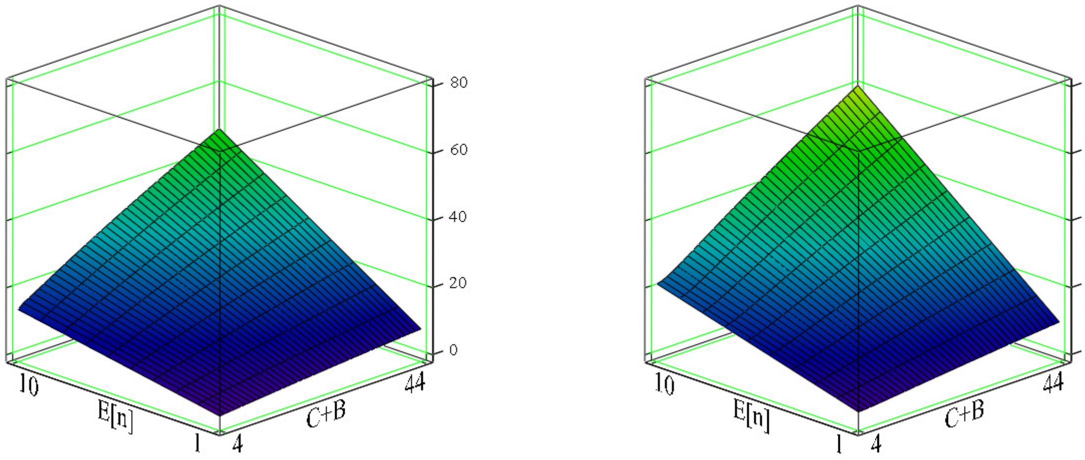
The number of segments transmitted during a slow start phase can be closely approximated as a geometric series $Y_i^{SS} = 1 + \gamma + \gamma^2 + \ldots + \gamma^{N_i - 1} = \left(\gamma^{N_i} - 1\right)/\left(\gamma - 1\right)$, where $\gamma = 1 + 1/b$ [68]. Taking into account that in the slow start phase of the i-th cycle the cwnd growths exponentially from one segment to $\mathrm{ssthresh}_i$, we get that $\gamma^{N_i - 1} = \max\left(W_{i-1}/2, 2\right)$. Hence, the expected duration (in rounds) of the slow start phase and the expected number of segments sent during this phase can be expressed as

$$E\left[A^{SS}\right] = \max\left(\log_\gamma\left(\frac{\gamma E[W]}{2}\right), 2\right), \quad E\left[Y^{SS}\right] = \max\left(\frac{\gamma E\left[W\right] - 2}{2\left(\gamma - 1\right)}, 3\right). \tag{4.14}$$

Neglecting those segments that were unnecessarily retransmitted during the delay spike, the total number of segments sent during the i-th cycle is $Y_i = Y_i^{SS} + Y_i^{CA1} + n_i Y^{CA0} + Y_i^{CA2}$ (see Fig. 4.14). Then the expected number of buffer overflows within a cycle is given by

$$E\left[n\right] = \frac{E\left[Y\right] - E\left[Y^{SS}\right] - E\left[Y^{CA1}\right] - E\left[Y^{CA2}\right]}{E\left[Y^{CA0}\right]}. \tag{4.15}$$

Let us compare the durations of the slow start phase after a delay spike and the subsequent congestion avoidance phase. Fig. 4.15 shows the ratio between the expected durations (in rounds) of the congestion avoidance phase, which can be defined as $E\left[A^{CA1}\right] + E\left[n\right]E\left[A^{CA0}\right] + E\left[A^{CA2}\right]$ (see Fig. 4.14), and the slow start phase (4.14). It is easy to see that the slow start phase is much shorter than the subsequent congestion avoidance phase over a wide range of network conditions.



a) $b = 1$

b) $b = 2$

Fig. 4. 15 Ratio between the expected durations of the congestion avoidance phase and the slow start phase after a delay spike

Then, neglecting the duration of the slow start phase after a delay spike, we can find the mean queue size during a cycle:

$$E\left[R\right] = \frac{E\left[R^{CA0}\right]E\left[n\right]E\left[A^{CA0}\right] + E\left[R^{CA1}\right]E\left[A^{CA1}\right] + E\left[R^{CA2}\right]E\left[A^{CA2}\right]}{E\left[n\right]E\left[A^{CA0}\right] + E\left[A^{CA1}\right] + E\left[A^{CA2}\right]}. \tag{4.16}$$

Similarly to (4.3), the expected queue size during phases CA1 and CA2 can be computed as

$$E\left[R^{CA1}\right]=\begin{cases}\dfrac{7B-9C}{16}, & B>\dfrac{5C}{3},\\ \dfrac{(B-C)^2}{C+B}, & B\in\left[C,\dfrac{5C}{3}\right],\end{cases}\qquad E\left[R^{CA2}\right]=\frac{5B-3C}{8}. \tag{4.17}$$

Taking into consideration that the expected number of packets sent during a cycle is given by (4.9) and the mean time required to transmit an IP packet over the wireless channel is equal to ε seconds, we define the TCP SACK long-term steady-state throughput as

$$T=\frac{E[Y]\mathrm{MSS}}{E[Y]\varepsilon+E\left[Z^{DS}\right]}=\frac{\mathrm{MSS}}{\varepsilon+Q'E\left[Z^{DS}\right]}, \tag{4.18}$$

where $\mathrm{MSS}=\mathrm{MTU}-40$ bytes; ε is from (4.20); Q' is from (4.8); $E\left[Z^{DS}\right]$ is from (4.10).

Evidently, when Q' is small, expression (4.18) can be simplified as follows:

$$T\approx\lim_{Q'\to 0}\left(\frac{\mathrm{MSS}}{\varepsilon+Q'E\left[Z^{DS}\right]}\right)=\frac{\mathrm{MSS}}{\varepsilon}. \tag{4.19}$$

4.5 TCP SACK Model: Semi-reliable ARQ

In this section, we consider the evolution of a TCP SACK connection over a wireless channel with semi-reliable ARQ/FEC and derive an expression for its long-term steady-state throughput. We suppose that each data packet may be dropped with probability p_L due to an excessive number of transmission attempts made for one of its frames at the data link layer or, consequently, successfully delivered to the IP layer with probability $1-p_L$. Let p_C be the packet loss rate due to buffer overflow at the IP layer in the absence of non-congestion losses. We develop our model in two steps:

- when non-congestion losses are sufficiently rare and, on average, there is at least one buffer overflow between two consecutive non-congestion losses ($p_C > p_L$);
- when non-congestion losses are frequent enough to keep the cwnd below the maximum number of packets that can be accommodated in the network ($p_C \le p_L$). In this case, non-congestion losses prevent the TCP sender from overloading the bottleneck buffer.

Note that we do not consider here TCP spurious timeouts caused by delay spikes due to wireless channel impairments. As it will be demonstrated later, TCP spurious timeouts do not occur when wireless channel conditions are covariance-stationary and their presence in some practical studies should be attributed to non-stationary behavior of wireless channel characteristics (caused by handovers, resource preemption due to higher priority traffic, etc.), which is, however, out of the scope of this thesis and remains for future studies. Moreover, according to the results obtained in [P5], the TCP spurious timeout probability is negligibly small even for very severe wireless channel conditions and a perfectly-persistent (i.e., completely reliable) ARQ scheme in use.

In order to distinguish between the models corresponding to completely reliable and semi-reliable ARQ, we denote the mean time within which the wireless channel is seized by transmitting an IP packet as ε in case of the completely reliable ARQ scheme and as δ in case of the semi-reliable ARQ scheme. It should be emphasized that when the ARQ scheme is completely reliable, the time required to transmit an IP packet over the wireless channel is potentially unlimited, while the lower bound is equal to the amount of time needed to successfully transmit all v frames to which the packet was segmented from the first try. When the ARQ scheme is semi-reliable, the time during which an IP packet is transmitted over the wireless channel is either the amount of time to successfully transmit the frames or the time till the packet is dropped due to an excessive number of transmission attempts made for one of its frames. Then the lower bound can be defined as $\min(r,v)$, while the maximum time is bounded by rv (i.e., every frame out of the total v requires exactly r attempts to be successfully transmitted). Again, we note that we define δ as the amount of time the wireless channel is seized by transmitting an IP packet irrespective of whether this packet is successfully transmitted or not. Thus, we have:

$$\begin{aligned}
\varepsilon &= \frac{m}{\mu}\sum_{k=v}^{\infty} f(k)k, \qquad k = v, v+1, \ldots, \\
\delta &= \frac{m}{\mu}\sum_{k=\min(r,v)}^{rv} d(k)k, \quad k = \min(r,v), \min(r,v)+1, \ldots, rv,
\end{aligned} \tag{4.20}$$

where $f(k)$ and $d(k)$ are the probability functions of the number of time slots the wireless channel is seized by transmitting an IP packet in case of completely reliable and semi-reliable ARQ, respectively (see [P5] [P6] for details); v is the number of frames to which an IP packet is segmented; r is the number of ARQ transmission attempts (including the original transmission and subsequent retransmissions).

4.5.1 Step 1: Buffer Overflows Dominate the Data Transfer

As before, we assume that the buffer at the intermediate system is sized in such a way to provide full utilization of the wireless channel, which implies that $B \geq C$ and $B + C \geq 3$ (see section 4.4

for details). Then, the average network throughput is equal to MTU/δ bits per second and the average end-to-end path capacity (expressed in packets of MTU size) is given by

$$C = \frac{1}{\mathrm{MTU}}\left(\frac{\mathrm{MTU}}{\delta}\tau + \mathrm{MTU}\right) = \frac{\tau}{\delta} + 1. \tag{4.21}$$

Now let us consider the evolution of a TCP SACK connection as a sequence of cycles, where a cycle is a period between two consecutive non-congestion losses separated by at least one packet loss due to buffer overflow (Fig. 4.16). The expected duration of a congestion avoidance phase in rounds ($E\left[A^{CA0}\right]$), the expected number of packets sent during this phase ($E\left[Y^{CA0}\right]$), and the packet loss rate due to buffer overflow at the IP layer (p_C) can be found as (4.2). Note that more than one non-congestion loss can occur within a single window of data. Let W_{i-1} denote the window size at the end of cycle $i-1$. After the loss detection, the ssthresh will be set as $\mathrm{ssthresh}_i = \max\left(W_{i-1}/2, 2\right)$. For $p_C > p_L$, we assume that the random variable W_i is uniformly distributed from $(C+B)/2$ to $C+B$. Then the values of $E[W]$ and $E[\mathrm{ssthresh}]$ can be obtained from (4.11).

Depending on W_{i-1} and the number of losses within a single window of data, the lost segment(s) may be detected either via three duplicate ACKs or via expiration of the TCP retransmission timer. As was demonstrated in [115], taking into account the probability of j losses out of a window of w packets, the timeout probability for TCP SACK is

$$\widehat{Q}(w) = \begin{cases} 1, & w < 4, \\ \displaystyle\sum_{j=w-2}^{w} \binom{w}{j} p_L^j \left(1-p_L\right)^{w-j} + \sum_{j=1}^{w-3} \binom{w}{j} p_L^j \left(1-p_L\right)^{w-j} \left(1-\left(1-p_L\right)^j\right), & w \geq 4. \end{cases} \tag{4.22}$$

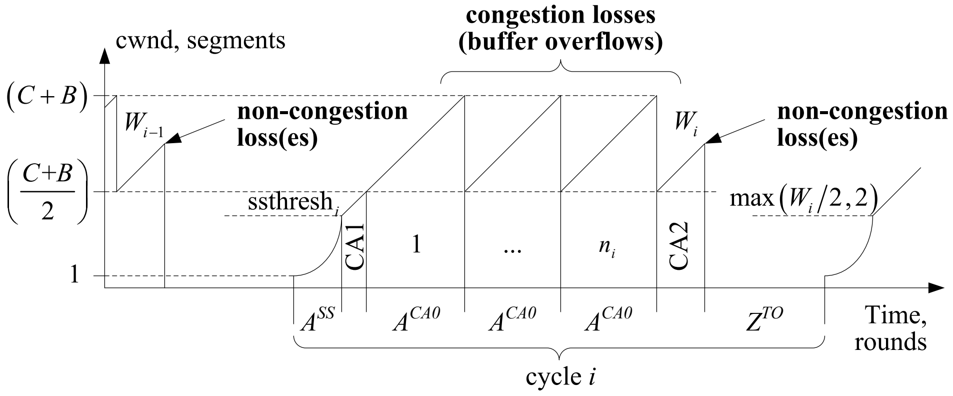


Fig. 4. 16 TCP SACK window evolution in the presence of congestion and non-congestion losses

As in [67], we approximate:

$$Q = \sum_{w=1}^{\infty} \widehat{Q}(w)\Pr[W = w] \approx \widehat{Q}(E[W]), \tag{4.23}$$

where Q is the probability that a non-congestion loss (or losses) will be detected via expiration of the TCP retransmission timer.

The expected durations of phases CA1 and CA2 (see Fig. 4.16) and the expected number of segments sent during these phases can be calculated as (4.12) and (4.13), respectively.

When the TCP retransmission timer expires (i.e., after RTO seconds), the cwnd is reduced to one segment and the first unacknowledged segment is retransmitted. The TCP sender also doubles the timeout value so that it will expire after 2RTO seconds. This doubling is repeated for each unsuccessful retransmission until the maximum value of 64RTO is reached. After that, the timeout value remains constant and equal to 64RTO seconds [45]. According to [67], the expected duration of a sequence of timeouts (TO) can be defined as follows:

$$E\left[Z^{TO}\right] = \overline{\mathrm{RTO}} \frac{\left(1 + p_L + 2p_L{}^2 + 4p_L{}^3 + 8p_L{}^4 + 16p_L{}^5 + 32p_L{}^6\right)}{1 - p_L}, \tag{4.24}$$

where $\overline{\mathrm{RTO}} = f\left(\overline{\mathrm{RTT}}\right)$.

Considering that each data packet may be dropped with probability p_L due to an excessive number of transmission attempts made for one of its frames at the data link layer or successfully delivered to the IP layer with probability $1 - p_L$, the expected number of packets sent during a cycle can be defined as

$$E[Y] = \sum_{k=1}^{\infty} (1 - p_L)^{k-1} p_L k = \frac{1}{p_L}. \tag{4.25}$$

Then we define the TCP SACK long-term steady-state throughput as

$$T = \begin{cases} \mathrm{MSS}/\delta, & p_L = 0, \\ \dfrac{\mathrm{MSS}(1/p_L)}{\delta(1/p_L) + QE\left[Z^{TO}\right]}, & p_L > 0, \end{cases} \tag{4.26}$$

where $\mathrm{MSS} = \mathrm{MTU} - 40$ bytes; δ is from (4.20); Q is from (4.23); $E\left[Z^{TO}\right]$ is from (4.24).

As it follows from Fig. 4.16, the total number of segments sent during the i-th cycle can be approximated as $Y_i = Y_i^{SS} + Y_i^{CA1} + n_i Y^{CA0} + Y_i^{CA2}$, where the presence of the slow start phase depends on the detection of non-congestion loss(es) (i.e., either via three duplicate ACKs or via expiration of the TCP retransmission timer). Therefore, the derivation of the mean value of the

RTT is similar to that presented in section 4.4, except the fact that the expected number of buffer overflows within a cycle is given by

$$E[n]=\frac{E[Y]-QE\left[Y^{SS}\right]-E\left[Y^{CA1}\right]-E\left[Y^{CA2}\right]}{E\left[Y^{CA0}\right]}. \tag{4.27}$$

4.5.2 Step 2: Non-congestion Losses Dominate the Data Transfer

Consider now the case when non-congestion losses are frequent enough ($p_C \le p_L$) to restrict the evolution of the cwnd below the sum of the end-to-end path capacity C and the buffer size B (see Fig. 4.17).

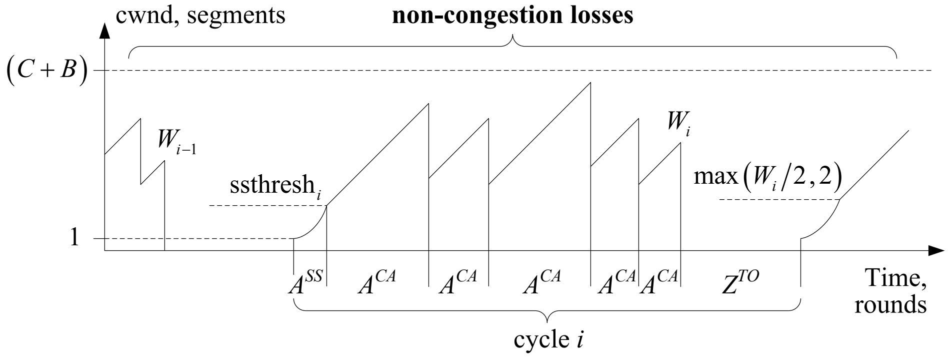


Fig. 4. 17 TCP SACK window evolution in the presence of non-congestion losses only

For small values of p_L, the derivation of $E[W]$ and $E\left[A^{CA}\right]$ is similar to [65]:

$$E[W]=\sqrt{\frac{8}{3bp_L}},\quad E\left[A^{CA}\right]=b\frac{E[W]}{2}. \tag{4.28}$$

Neglecting the slow start phase after a timeout, we get the expected queue size during the subsequent congestion avoidance phase(s) as

$$E[R]=\begin{cases}\left(\dfrac{3E[W]}{4}-C\right), & E[W]\ge 2C,\\[2ex] \dfrac{\left(E[W]-C\right)^2}{E[W]}, & E[W]\in(C,2C),\\[2ex] 0 & E[W]\le C.\end{cases} \tag{4.29}$$

Once the mean value of the RTT is obtained, we can calculate the mean duration of the first timeout as (4.6) and substitute it into (4.24).

Note that expressions (4.28) only hold when $E[W] \geq 4$ and, hence, $p_L \leq 1/6b$. In this case, the TCP sender gets enough duplicate ACKs to trigger the SACK-based loss recovery algorithm. However, when $p_L > 1/6b$, not enough duplicate ACKs arrive from the TCP receiver and a timeout event is required to detect a lost segment (or segments). Thus, on average, every non-congestion loss will be followed by a timeout. Then $Q = 1$ and $E[R] \to 0$. When $p_L = 1$, we approximate the window size at loss events as $E[W] = 1$.

4.6 Numerical Analysis

In this section, we estimate various metrics characterizing TCP performance over wireless channels with completely reliable and semi-reliable ARQ/FEC. To demonstrate the effect of different FEC codes, we use the following BCH codes: (255,131,18), (511,250,31), (255,87,26), and (511,157,51), where a triplet (m,n,l) denotes that in a codeword of size m bits and containing n data bits up to l errors can be corrected. The code rate is equal to n/m, so the code rate of the first two FEC codes is approximately 1/2 and the code rate of the last two FEC codes is roughly 1/3. Note that the number of frames per packet can be defined as

$$v = \left\lceil \frac{\text{MTU}}{n} \right\rceil, \tag{4.30}$$

where $\lceil \quad \rceil$ is the ceiling function.

Table 4.2 summarizes the values of v for the given FEC codes and $\text{MTU} = 1500$ bytes.

Table 4. 2 Number of frames per packet

	(255,131,18)	**(511,250,31)**	**(255,87,26)**	**(511,157,51)**
Frames per packet (v)	92	48	138	77

The BER of the wireless channel is set to vary between 0.01 and 0.10. Note that the latter value corresponds to a very noisy wireless channel. The lag-1 NACF varies from 0.00 (the bit error process has no autocorrelation at lag 1) to 0.95 (a high degree of autocorrelation at lag 1). Values of the default system parameters used in the numerical analysis are listed in Table 4.3.

Table 4. 3 Default system parameters

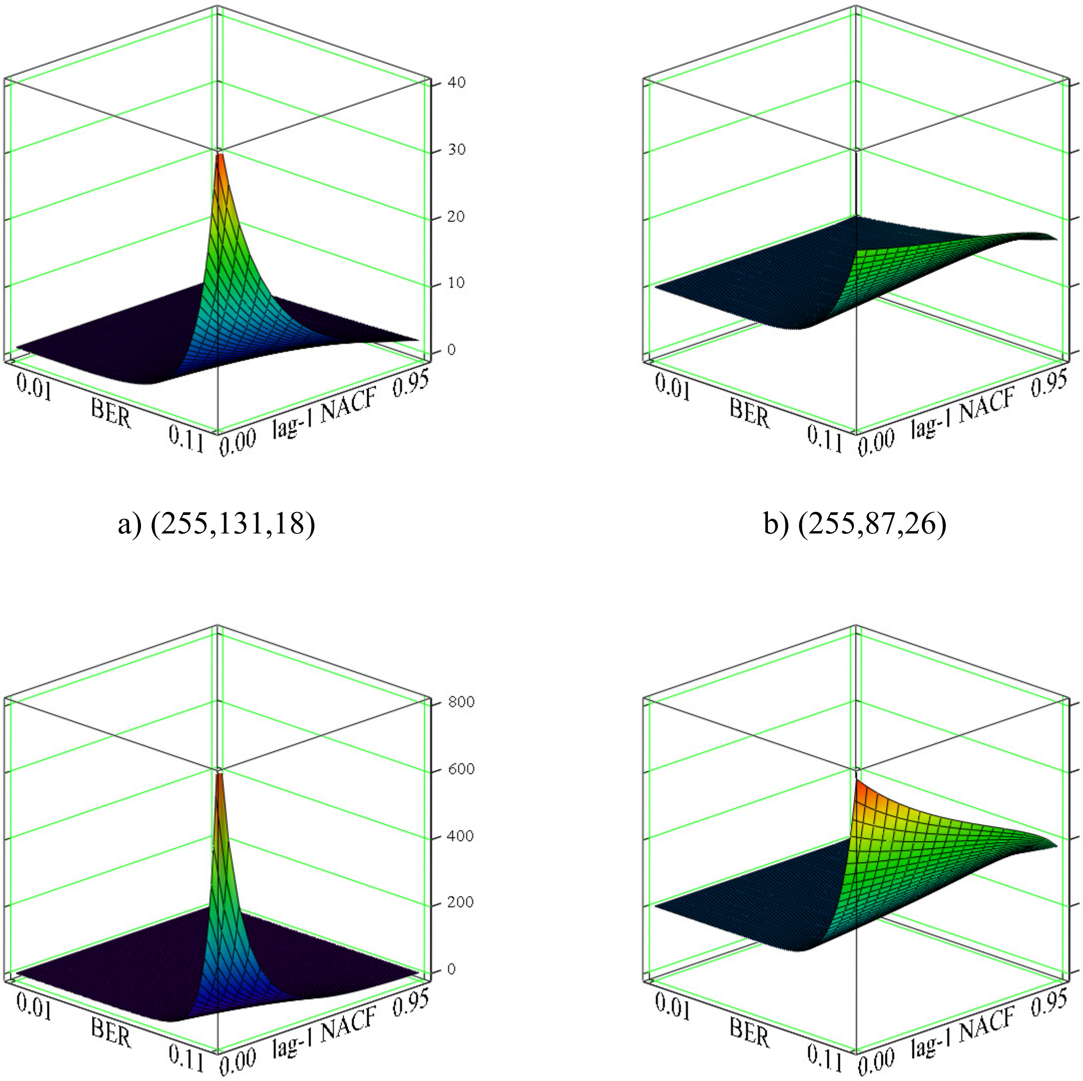
Input parameter	**Value**
BER $(E[W_E])$	0.01, …, 0.11; step 0.001
lag-1 NACF $(K_E(1))$	0.00, …, 0.95; step 0.05
FEC code	(255,131,18), (511,250,31), (255,87,26), (511,157,51)
MTU	1500 bytes
MSS	1460 bytes
Bottleneck link buffer size (B)	20 packets of MTU size
Data rate of the wireless channel (μ)	2 Mbit/s
Round-trip delay of the wired network (τ)	10 ms
Number of segments acknowledged by one ACK (b)	1
By how much does $\overline{\text{RTO}}$ exceed $\overline{\text{RTT}}$ (x)	2
Minimum value of the RTO ($\text{RTO}_{\min}$)	1 s
Number of transmission attempts per frame* (r) * This parameter applies to semi-reliable ARQ only	3, 6, 9, 30

4.6.1 Completely Reliable ARQ

4.6.1.1 Service process of the wireless channel

First of all, let us consider the mean number of transmission attempts per frame (including failed and successful transmissions) as a function of the BER, the lag-1 NACF, and different FEC codes (Fig. 4.18). When the BER is sufficiently small, the FEC code can correct all errors in a frame without requiring a retransmission. As a result, all the frames in an IP packet will be successfully transmitted in their first attempts, which implies one transmission attempt per frame. However, when the BER increases and not all transmission errors can be corrected, the erroneous frame is discarded and a retransmission is requested by the ARQ receiver. Obviously, when the channel quality is relatively "bad", more powerful FEC codes provide better performance, requiring less transmission attempts (see Fig. 4.18b and Fig. 4.18d).

Observing Fig. 4.18, we conclude that FEC codes with different codeword lengths but the same code rate perform similarly. The difference between the performance corresponding to the codeword length of 255 and 511 bits is due to slight deviations in the code rate and the error correcting capability of these FEC codes.



a) (255,131,18)

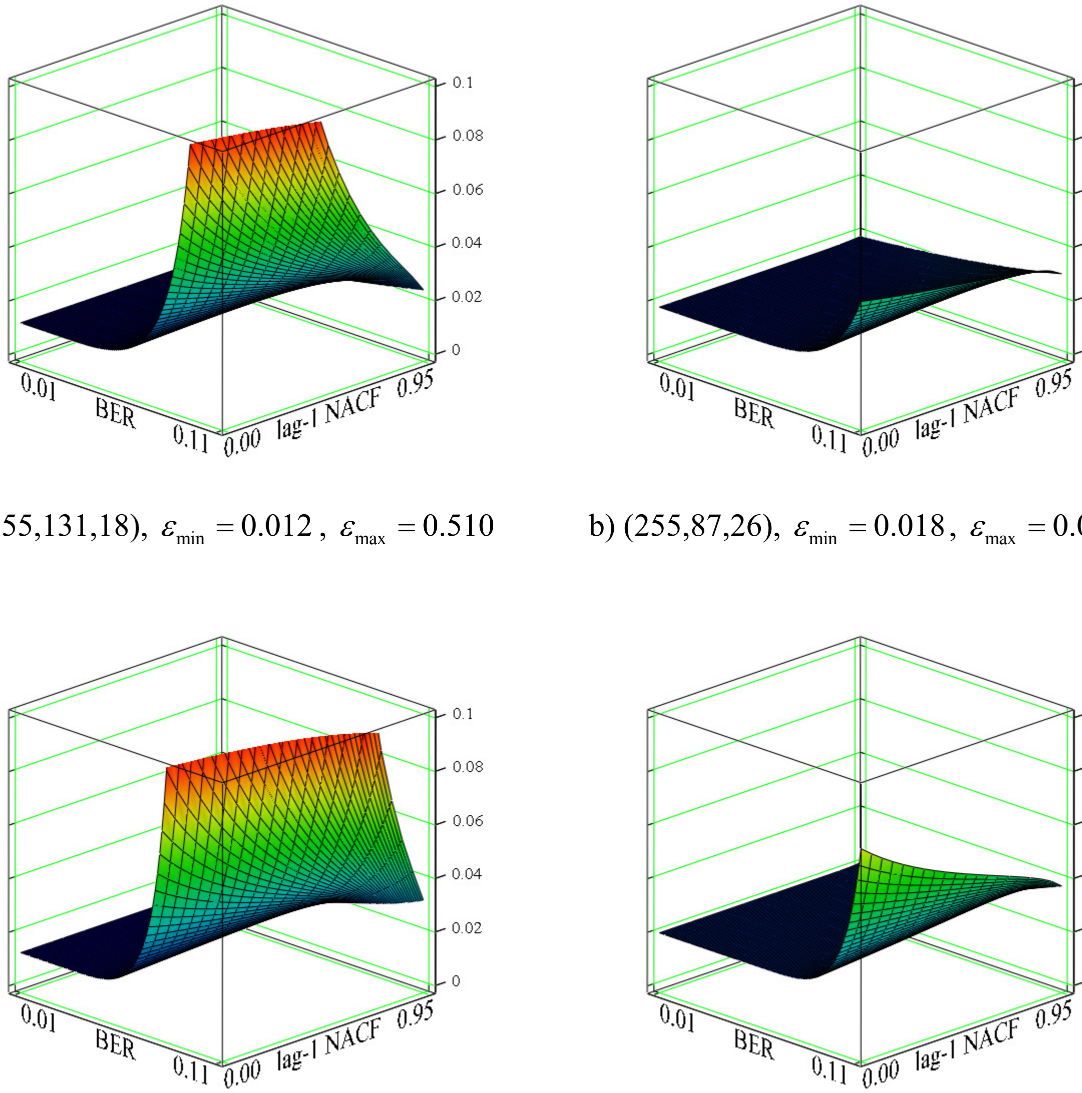
b) (255,87,26)

c) (511,250,31)

d) (511,157,51)

Fig. 4. 18 Mean number of transmission attempts per frame, including failed and successful transmissions

The mean time required to transmit an IP packet over the wireless channel as a function of the BER, the lag-1 NACF, and different FEC codes is shown in Fig. 4.19. We note that the increase in the strength of the FEC code results in higher delays when the BER is small. Of course, FEC codes with greater redundancy result in more frames and, thus, more bits to transmit. On the other hand, when the BER increases, more powerful FEC codes perform better. We also note that the increase in the BER results in higher delay values, since those frames that are dropped due to a large number of incorrectly received channel symbols require retransmission and, consequently, increase the total time needed to successfully transmit all frames to which the packet was segmented.



a) (255,131,18), $\varepsilon_{\min} = 0.012$, $\varepsilon_{\max} = 0.510$

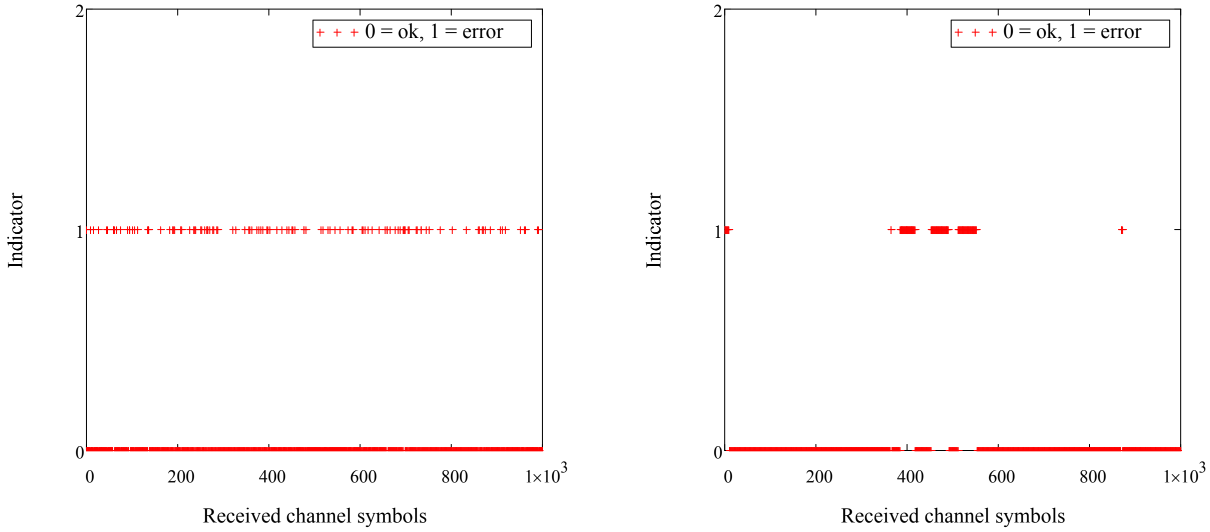
b) (255,87,26), $\varepsilon_{\min} = 0.018$, $\varepsilon_{\max} = 0.045$

c) (511,250,31), $\varepsilon_{\min} = 0.012$, $\varepsilon_{\max} = 11.334$

d) (511,157,51), $\varepsilon_{\min} = 0.020$, $\varepsilon_{\max} = 0.077$

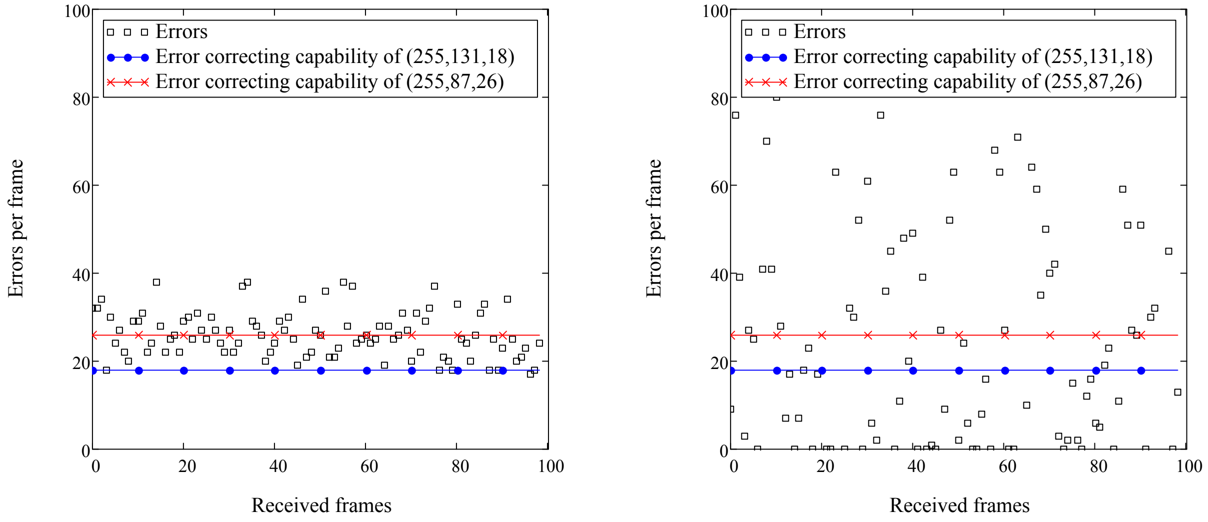
Fig. 4. 19 Mean time (in seconds) required to transmit an IP packet over the wireless channel

The effect of autocorrelation is more complex. When the BER is small, the lag-1 autocorrelation of the bit error process almost does not affect the performance of the wireless channel. But when the BER increases, the lag-1 autocorrelation of bit errors results in a smaller delay. This behavior can be explained as follows. The lag-1 autocorrelation of the bit error process manifests itself in clumping of errors (Fig. 4.20). Thus, higher values of the lag-1 NACF result in a less deterministic process with a high variance around the mean number of errors per frame (see Fig. 4.21b). On the other hand, lower values of the lag-1 NACF lead to a more uniform distribution of errors over transmitted data, thus decreasing the spread in the number of errors per frame (see Fig. 4.21a).



a) $K_E(1) = 0.00$, $E[W_E] = 0.10$ b) $K_E(1) = 0.95$, $E[W_E] = 0.10$

Fig. 4. 20 Effect of autocorrelation on the bit error process

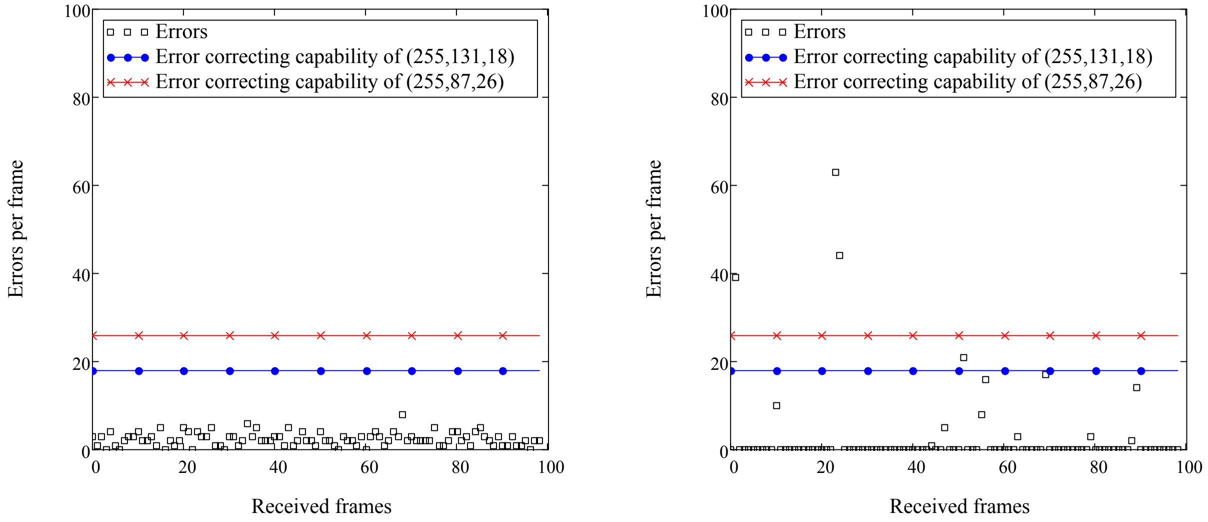


a) $K_E(1) = 0.00$, $E[W_E] = 0.10$ b) $K_E(1) = 0.95$, $E[W_E] = 0.10$

Fig. 4. 21 Number of errors per frame versus the strength of the FEC codes under high BER

As an example, let us consider the (255,131,18) FEC code. For large values of the BER and small values of the lag-1 NACF, bit errors, even being well distributed, result in more incorrectly received channel symbols per frame than the FEC code can correct, so almost all frames are received incorrectly (Fig. 4.21a). As a result, this increases the amount of time required to transmit an IP packet over the wireless channel. However, when the lag-1 autocorrelation of bit errors is sufficiently high, bit errors tend to occur in groups. Given the same BER, this unequal distribution leads to more frames received correctly (Fig. 4.21b). This effect remains the same for all FEC codes.

At the same time, the lag-1 autocorrelation of the bit error process may affect performance of the wireless channel even when the BER is small (this is more visible in Fig. 4.24a when $K_E(1) = 0.95$ and $E[W_E] = 0.01$). More specifically, it leads to slightly worse performance for FEC codes with a small error correcting capability. Indeed, large values of the lag-1 NACF lead to more lengthy bursts of errors within a frame that the FEC code cannot correct (see Fig. 4.22b). This, in turn, increases the number of transmission attempts required to successfully transmit an IP packet over the wireless channel.



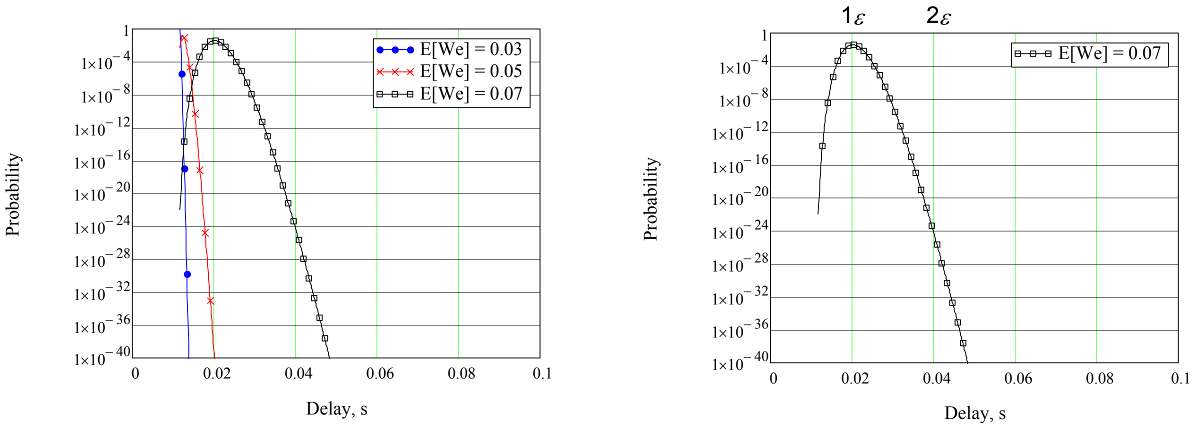
a) $K_E(1) = 0.00$, $E[W_E] = 0.01$ b) $K_E(1) = 0.95$, $E[W_E] = 0.01$

Fig. 4. 22 Number of errors per frame versus the strength of the FEC codes under low BER

4.6.1.2 TCP spurious timeout probability

Now let us estimate the spurious timeout probability for a TCP SACK connection over a wireless channel behaving in a covariance-stationary manner. As it was demonstrated in section 4.4, a TCP spurious timeout occurs when the variability in the wireless channel quality introduces a sudden delay in the service process of an IP packet which is equal to or larger than RTO seconds.

The expected duration of the RTO can be defined as (4.7). Our numerical analysis shows that the spurious timeout probability is negligibly small for $M > 2$. For instance, Fig. 4.23a illustrates the probability functions (in log scale) of the time required to transmit an IP packet over the wireless channel using the (255,131,18) FEC code for $K_E(1) = 0.00$ and different values of $E[W_E]$. It is easy to observe that the worst possible scenario for TCP is when the BER is high. Fig. 4.23b shows the probability function for $E[W_E] = 0.07$, as well as the mean of the distribution (ε) and the quantity required to estimate the probability of a spurious retransmission timeout for $M = 2$. According to the first case in (4.8), the probability mass beyond 2ε provides an estimate for the TCP spurious timeout probability. Note that already for $M = 2$ and $E[W_E] = 0.07$ the probability of a spurious retransmission timeout is less than 10E-24. Considering the other two cases in (4.8), we get that the probability of a spurious timeout is only insignificantly larger. The obtained results allow us to conclude that completely reliable ARQ does not lead to TCP spurious timeouts when wireless channel conditions are covariance-stationary. This conclusion well agrees with the findings in [97] [98] [100].



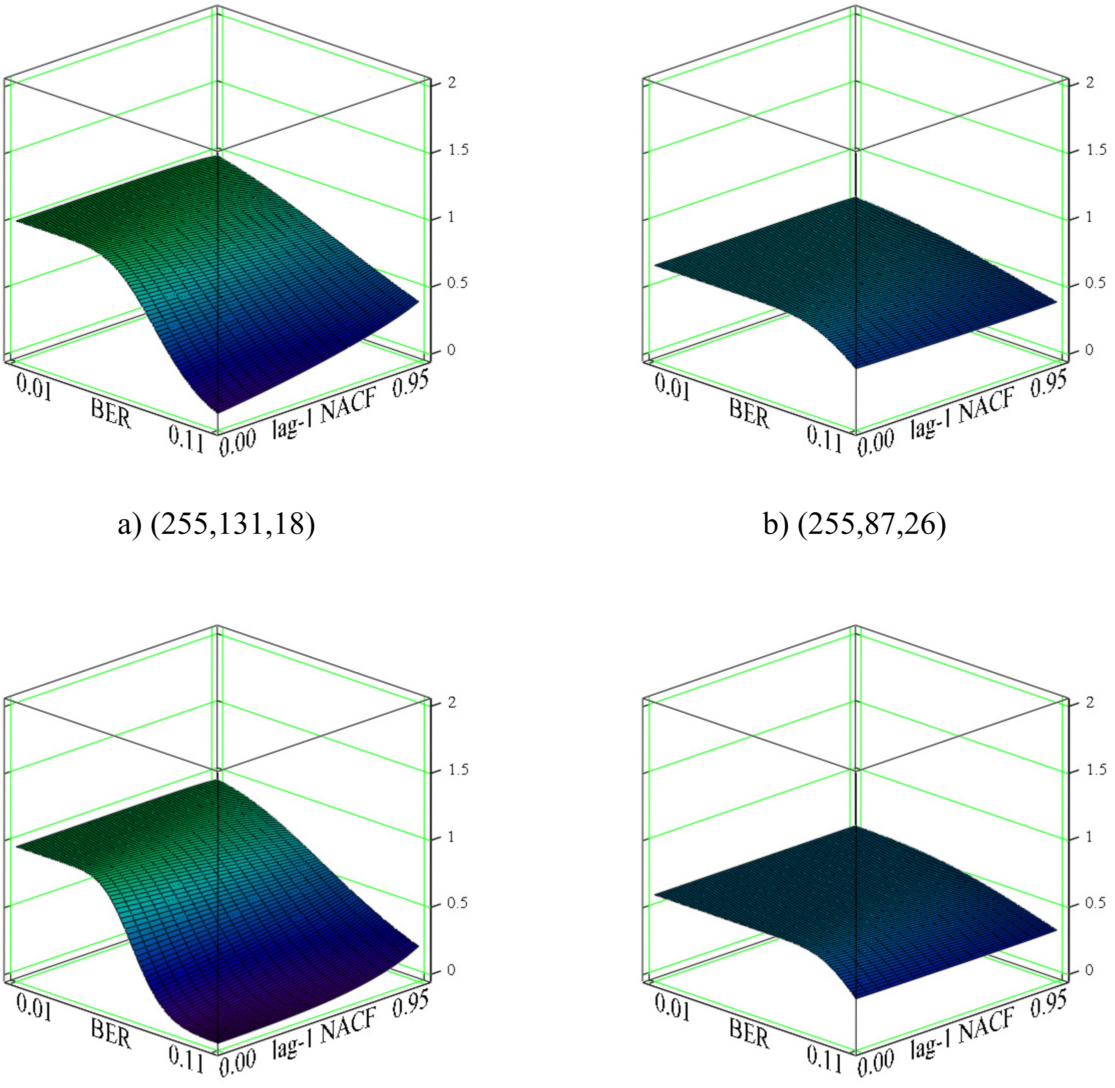
a) (255,131,18) FEC code, $K_E(1) = 0.00$

b) (255,131,18) FEC code, $\varepsilon = 0.02$ s, $2\varepsilon = 0.04$ s

Fig. 4. 23 Probability function (in log scale) of the time required to transmit an IP packet over the wireless channel, $K_E(1) = 0.00$

4.6.1.3 TCP SACK steady-state throughput

The TCP SACK long-term steady-state throughput as a function of the BER, the lag-1 NACF, and different FEC codes is shown in Fig. 4.24. As one may note, it is inverse proportional to the mean time required to transmit an IP packet over the wireless channel (Fig. 4.19). It is easy to see that depending on the BER and the lag-1 autocorrelation of bit errors, different FEC codes provide better performance in terms of TCP throughput. As it was noted in [97] [98] [100], there is a clear trade-off between the bandwidth consumed by FEC and the gain archived in TCP performance. Therefore, an adaptive FEC scheme, allowing to adjust code parameters on the fly as a function of the wireless channel quality, is the best choice for both TCP performance and efficient resource utilization.



a) (255,131,18)

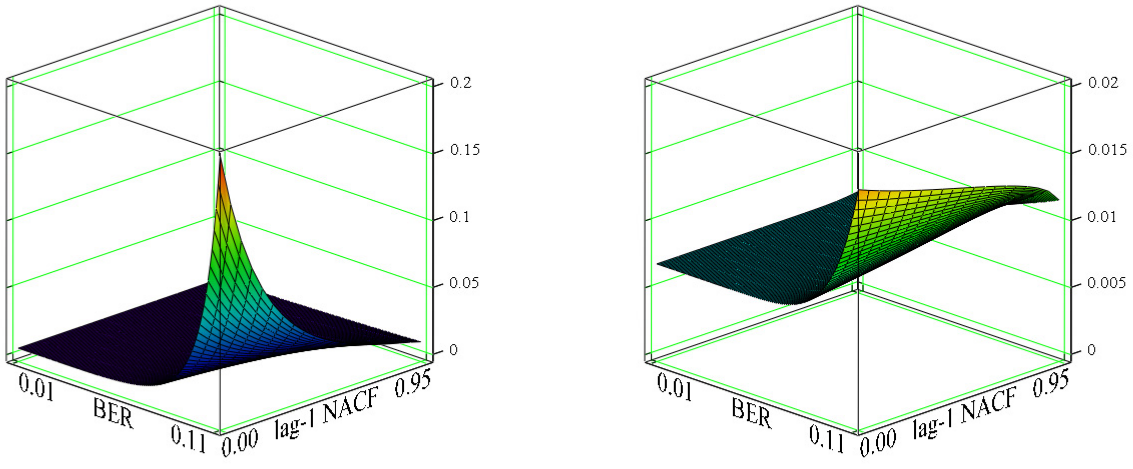
b) (255,87,26)

c) (511,250,31)

d) (511,157,51)

Fig. 4. 24 TCP SACK steady-state throughput (in Mbit/s), $\mathrm{MTU} = 1500$ bytes

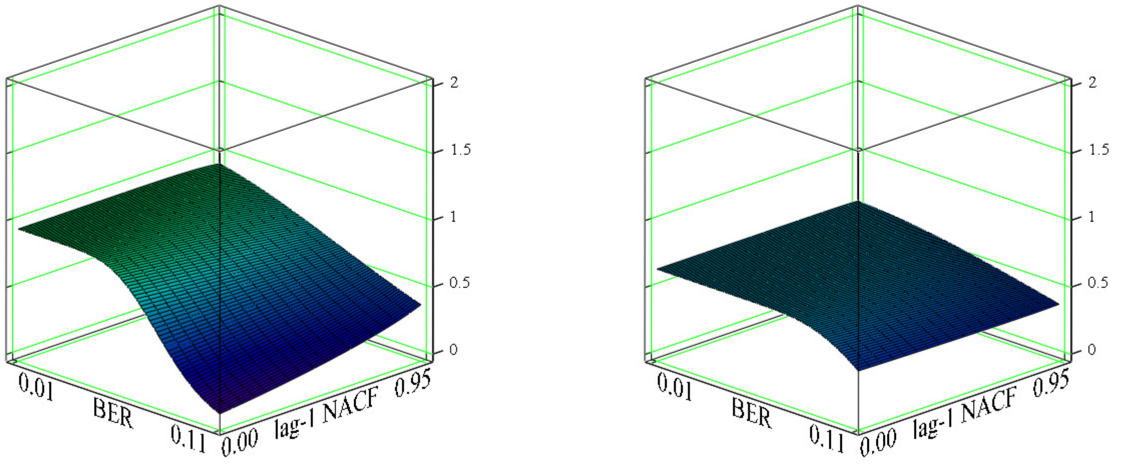
Fig. 4.25 and Fig. 4.26 demonstrate the impact of the MTU size on the mean time required to transmit an IP packet over the wireless channel and the TCP steady-state throughput, respectively. For the sake of briefness, we present the results for the (255,131,18) and (255,87,26) FEC codes only. The results for the (511,250,31) and (511,157,51) FEC codes are qualitatively similar. Comparing Fig. 4.19 and Fig. 4.25, it is easy to see that the change in the MTU size simply affects the magnitude of the mean time required to transmit an IP packet over the wireless channel, since the decrease in the IP packet size results in less bytes to transmit. On the other hand, the change in the MTU size almost does not affect the TCP throughput because a smaller MTU size implies a smaller MSS value. The small reduction in the TCP throughput is due to the increased protocol header overhead: from $40/1500 \approx 0.027$ to $40/576 \approx 0.069$.



a) (255,131,18)

b) (255,87,26)

Fig. 4. 25 Mean time (in seconds) required to transmit an IP packet over the wireless channel, $\mathrm{MTU} = 576$ bytes



a) (255,131,18)

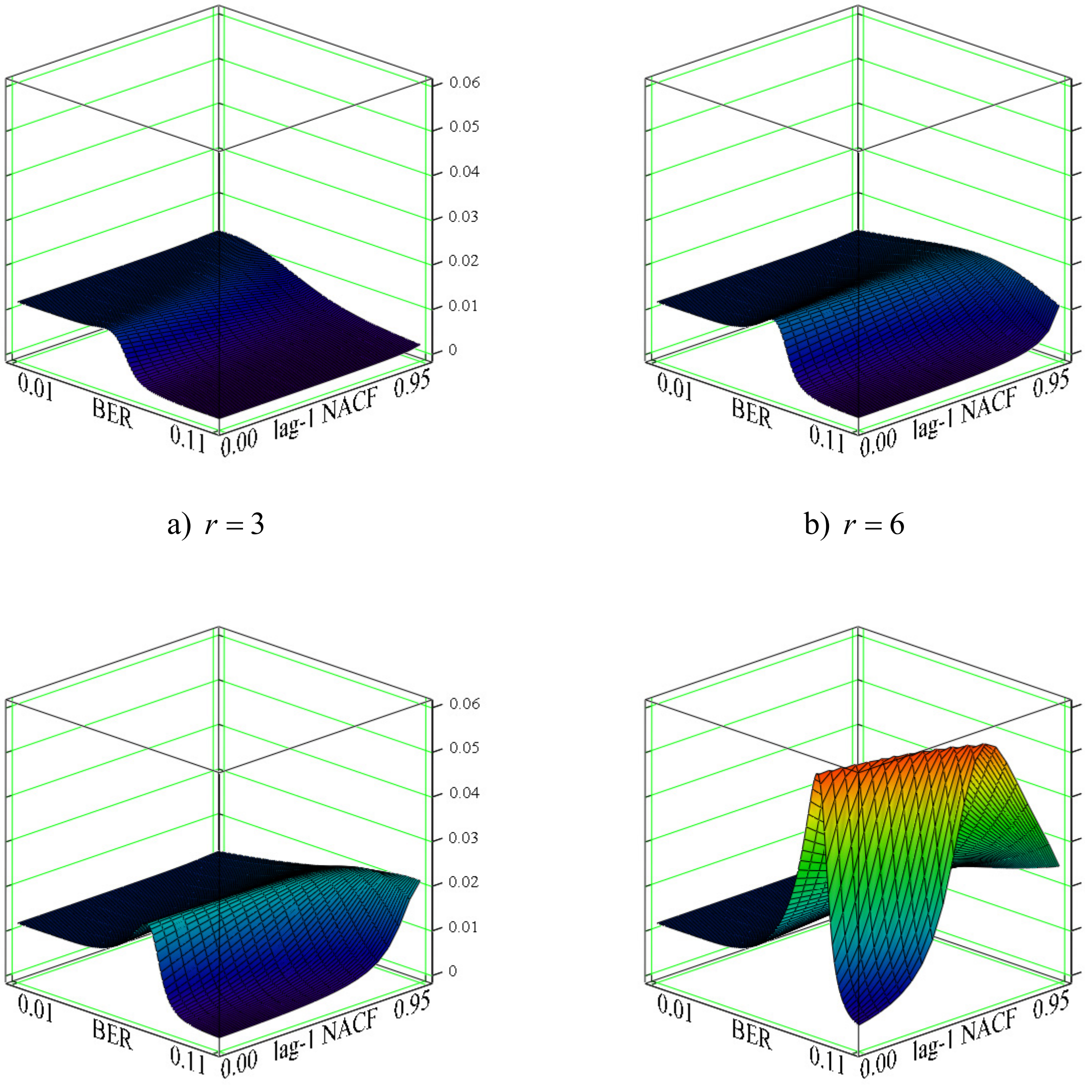
b) (255,87,26)

Fig. 4. 26 TCP SACK steady-state throughput (in Mbit/s), $\mathrm{MTU} = 576$ bytes

4.6.2 Semi-reliable ARQ

4.6.2.1 Service process of the wireless channel

Now let us consider how the BER and the lag-1 autocorrelation of bit errors affect packet transmission over wireless channels with semi-reliable ARQ. As it was demonstrated in section 4.6.1, FEC codes with different codeword lengths but the same code rate perform similarly, so for the sake of briefness, we present the results only for one pair of the FEC codes: (255,131,18) and (255,87,26). The results for the other pair of FEC codes are qualitatively similar.



a) $r = 3$

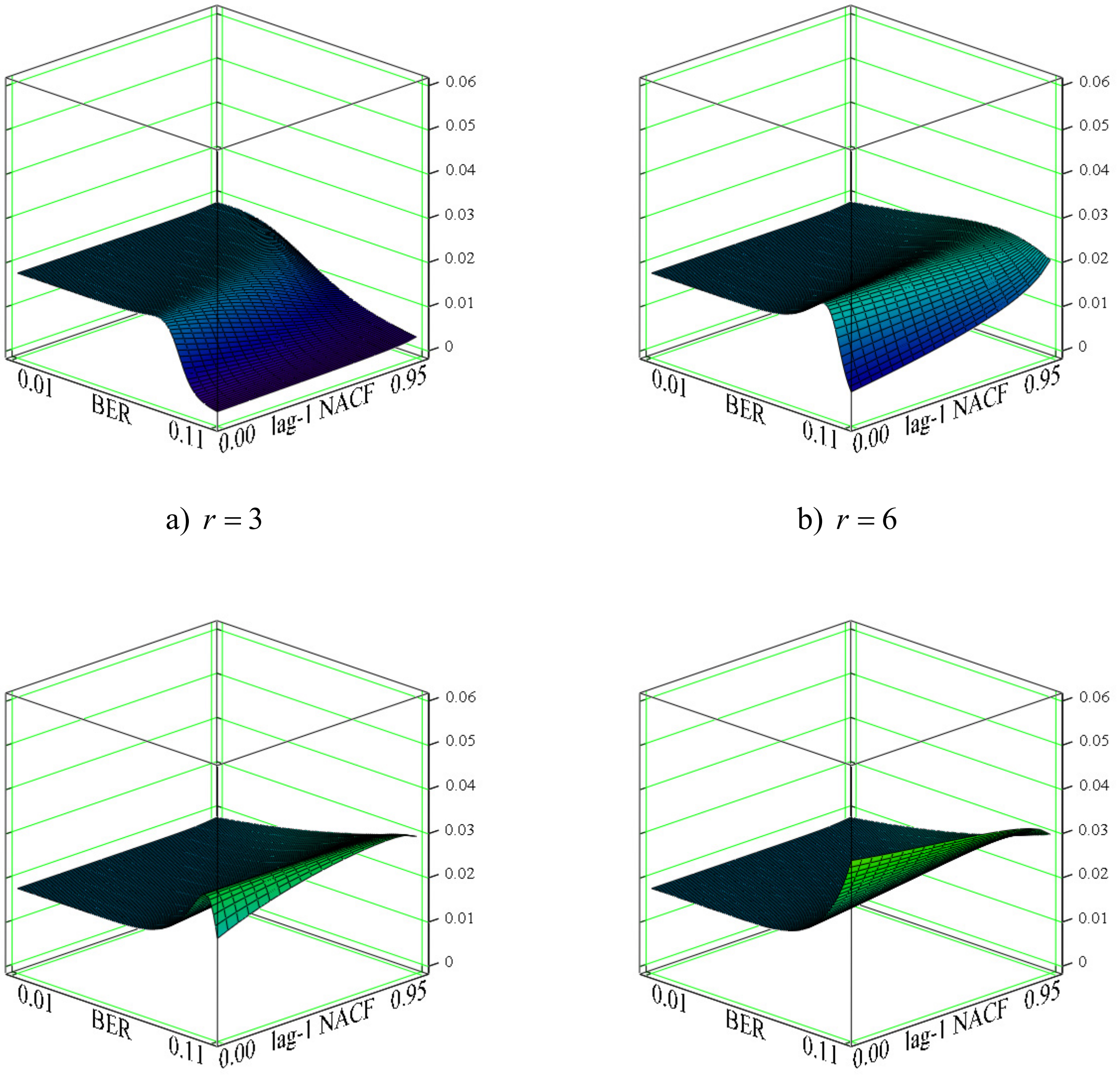
b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 27 Mean time (in seconds) during which an IP packet is transmitted over the wireless channel, (255,131,18)

The mean packet transmission delay (including failed and successful transmissions) for the (255,131,18) and (255,87,26) FEC codes is shown in Fig. 4.27 and Fig. 4.28, respectively. Note that this metric represents the average amount of time the wireless channel is seized by transmitting an IP packet regardless of whether it is successfully transmitted or not. The packet loss rate due to an excessive number of transmission attempts for the (255,131,18) and (255,87,26) FEC codes is shown in Fig. 4.29 and Fig. 4.30, correspondingly.



a) $r = 3$

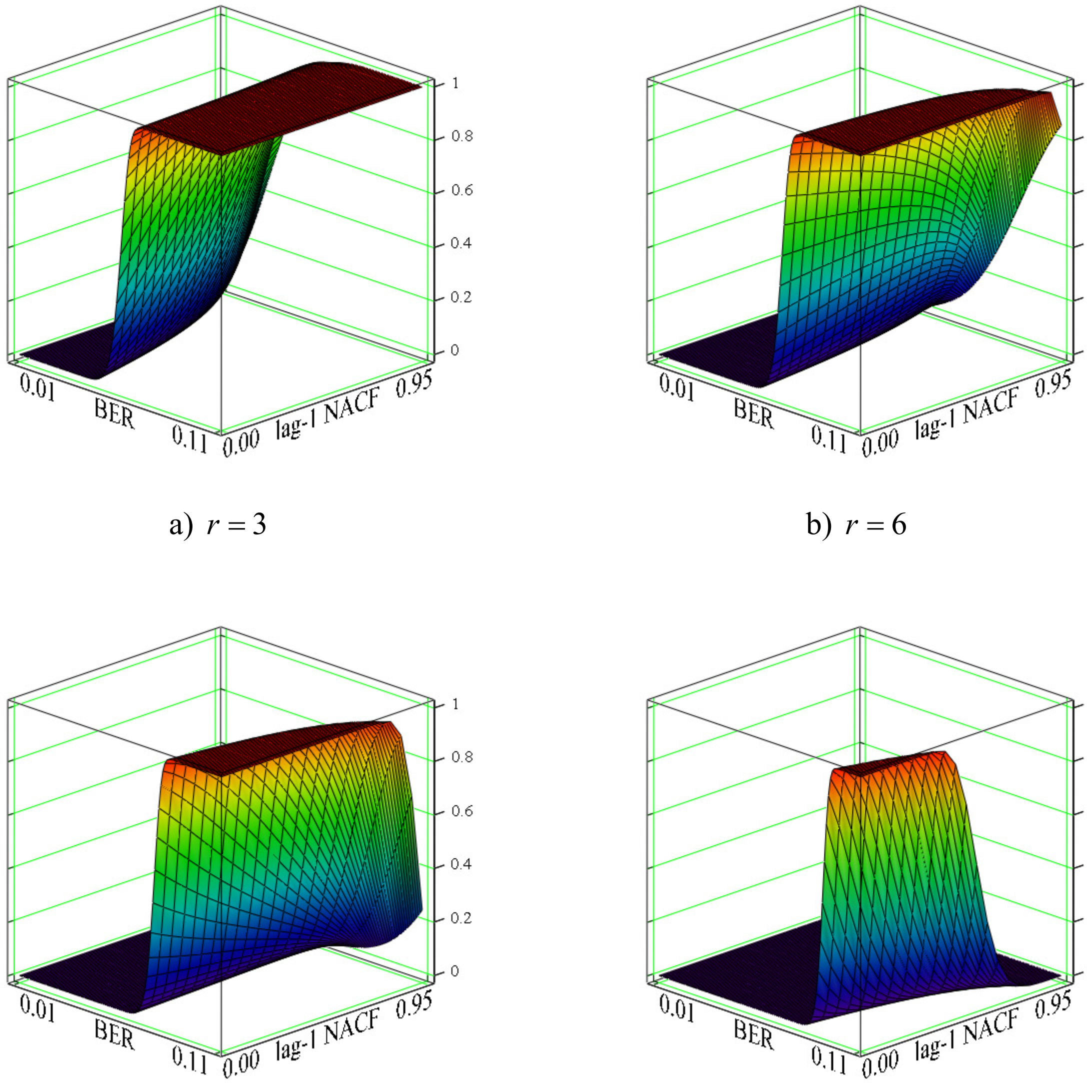
b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 28 Mean time (in seconds) during which an IP packet is transmitted over the wireless channel, (255,87,26)

As expected, when the BER is small, the FEC code can correct almost all errors in a frame without requiring a retransmission. Therefore, nearly all frames to which an IP packet was segmented will be successfully transmitted at the first try and the packet transmission delay will be approximately v time slots (the duration of a time slot is assumed to be constant and equal to m/μ seconds). Next, when the BER increases and not all transmission errors in a received frame can be corrected, the erroneous frame is dropped and a new copy is retransmitted, increasing the total number of transmission attempts and, consequently, the packet transmission delay. However, once r successive times any frame fails to be successfully transmitted, the whole IP packet is dropped irrespective of the number of frames that have already been correctly received.



a) $r = 3$

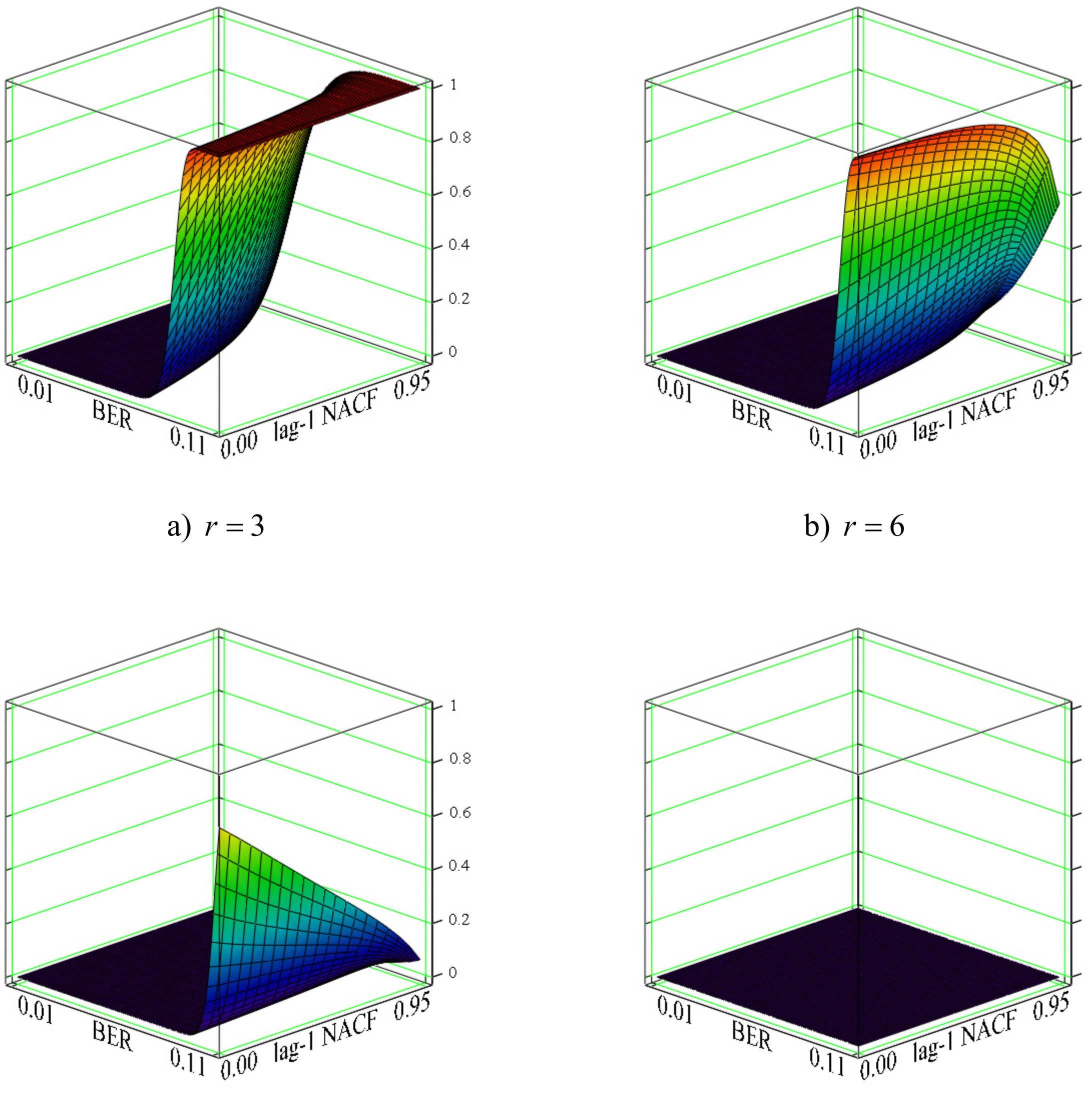
b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 29 Packet loss rate due to an excessive number of transmission attempts, (255,131,18)

Therefore, a further increase in the BER results in a less number of transmission attempts per packet, since a frame being dropped due to an excessive number of transmission attempts implies that the ARQ sender will drop all the subsequent frames belonging to that packet even not trying to transmit them. Obviously, when the wireless channel conditions are extremely "bad", so the error correcting capability of the FEC code and the persistency of the ARQ scheme do not allow to recover all transmission errors, the packet transmission delay tends to r time slots. In other words, the first frame of every transmitted packet fails to be successfully transmitted in r attempts, which means that every packet will be lost due to an excessive number of transmission attempts and $p_L = 1$.



a) $r = 3$

b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 30 Packet loss rate due to an excessive number of transmission attempts, (255,87,26)

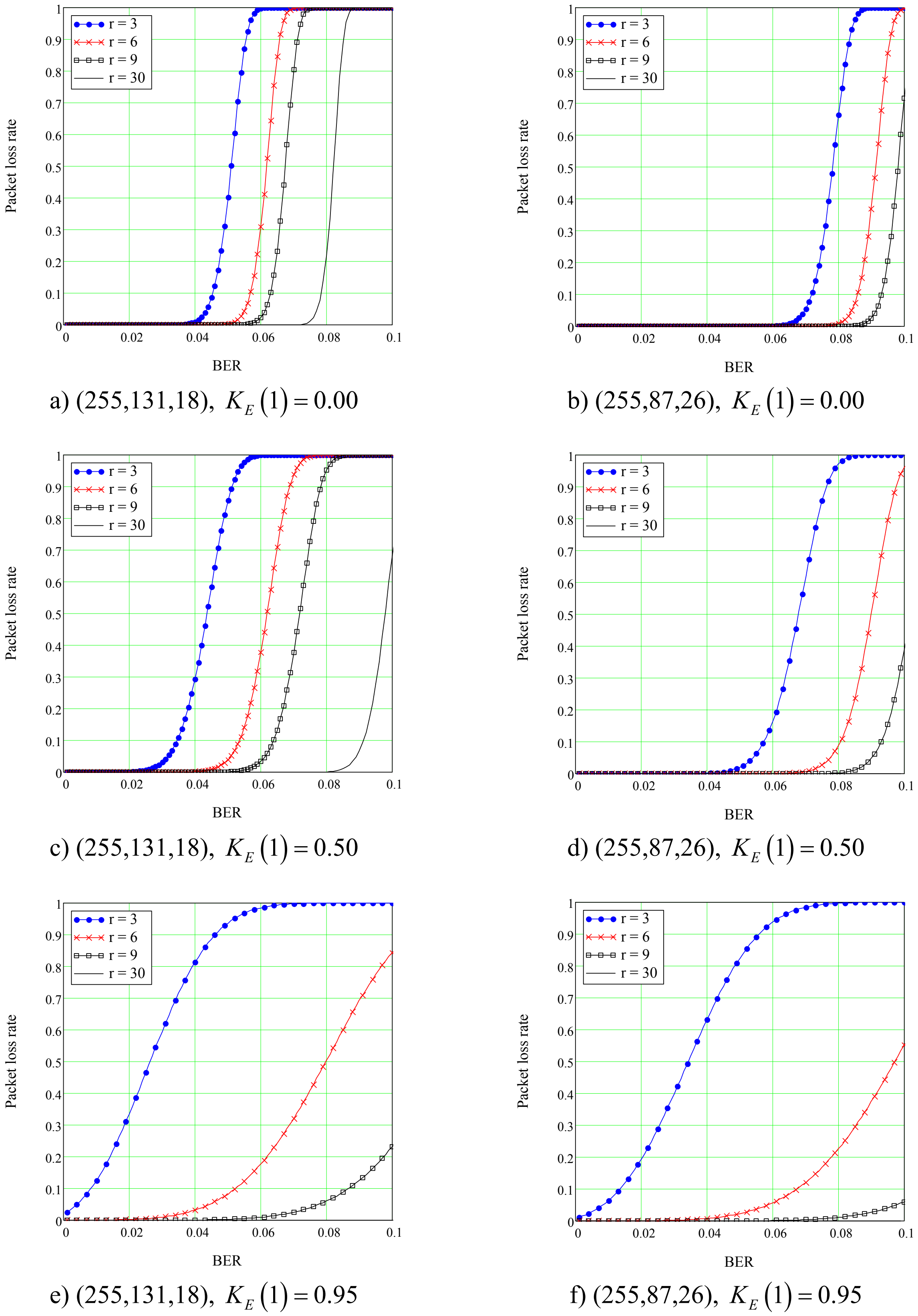
It is worthwhile to note that for a given range of BER values, the (255,87,26) FEC code combined with the high-persistence ARQ scheme ($r = 30$) provides completely reliable operation of the data link layer, so $p_L = 0$ (see Fig. 4.30d).

To demonstrate how the lag-1 autocorrelation of the bit error process affects the performance of the wireless channel, let us consider three different bit error processes, assuming that the first process has no autocorrelation at lag 1 ($K_E(1) = 0.00$), the second exhibits a moderate degree of autocorrelation at lag 1 ($K_E(1) = 0.50$), and the last one exhibits a very high degree of autocorrelation at lag 1 ($K_E(1) = 0.95$). Observing Fig. 4.31, we note that the lag-1 NACF significantly affects the packet loss rate due to an excessive number of transmission attempts, while the magnitude of this effect greatly depends on the ARQ persistency.

First of all, when the BER is small, the lag-1 autocorrelation of the bit error process results in higher packet loss rates due to an excessive number of transmission attempts. This effect is most noticeable in case of a low-persistence ARQ scheme (e.g., $r = 3$). The observed behavior can be explained as follows. The lag-1 autocorrelation of the bit error process manifests itself in clumping of errors (see Fig. 4.20b). Therefore, higher values of the lag-1 NACF result in a less deterministic process with a high variance around the mean number of errors per frame (see Fig. 4.22b). On the other hand, lower values of the lag-1 NACF lead to a more uniform distribution of errors over transmitted data (see Fig. 4.20a), thus decreasing the spread in the number of errors per frame (see Fig. 4.22a). As a result, large values of the lag-1 NACF lead to more lengthy bursts of errors within a frame than the FEC code can correct. This, in turn, increases the number of transmission attempts required to successfully transmit an IP packet over the wireless channel and, hence, increases the probability that a packet will be dropped due to an excessive number of transmission attempts made for one of its frames.

At the same time, when the BER is high, the lag-1 autocorrelation of the bit error process results in a lower packet loss rate due to an excessive number of transmission attempts. This effect mainly occurs for high-persistence ARQ schemes (e.g., $r = 6,9,30$). As shown in Fig. 4.20a and Fig. 4.21a, when the BER is high and the lag-1 NACF is small, bit errors, even being well distributed over transmitted data, result in more incorrectly received channel symbols per frame than the FEC code can correct, so almost all frames are received incorrectly. Consequently, this increases the total number of transmission attempts required to successfully transmit an IP packet over the wireless channel and naturally increases the probability that a packet will be dropped due to an excessive number of transmission attempts made for one of its frames. When the lag-1 autocorrelation of bit errors is sufficiently high, bit errors tend to occur in groups. Given the same BER, it leads to more frames received correctly (see Fig. 4.21b).

However, to take advantage of the error grouping effect, the ARQ scheme should be highly-persistent in order to be able to cope with a large spread in the number of errors per frame.



a) (255,131,18), $K_E(1) = 0.00$

b) (255,87,26), $K_E(1) = 0.00$

c) (255,131,18), $K_E(1) = 0.50$

d) (255,87,26), $K_E(1) = 0.50$

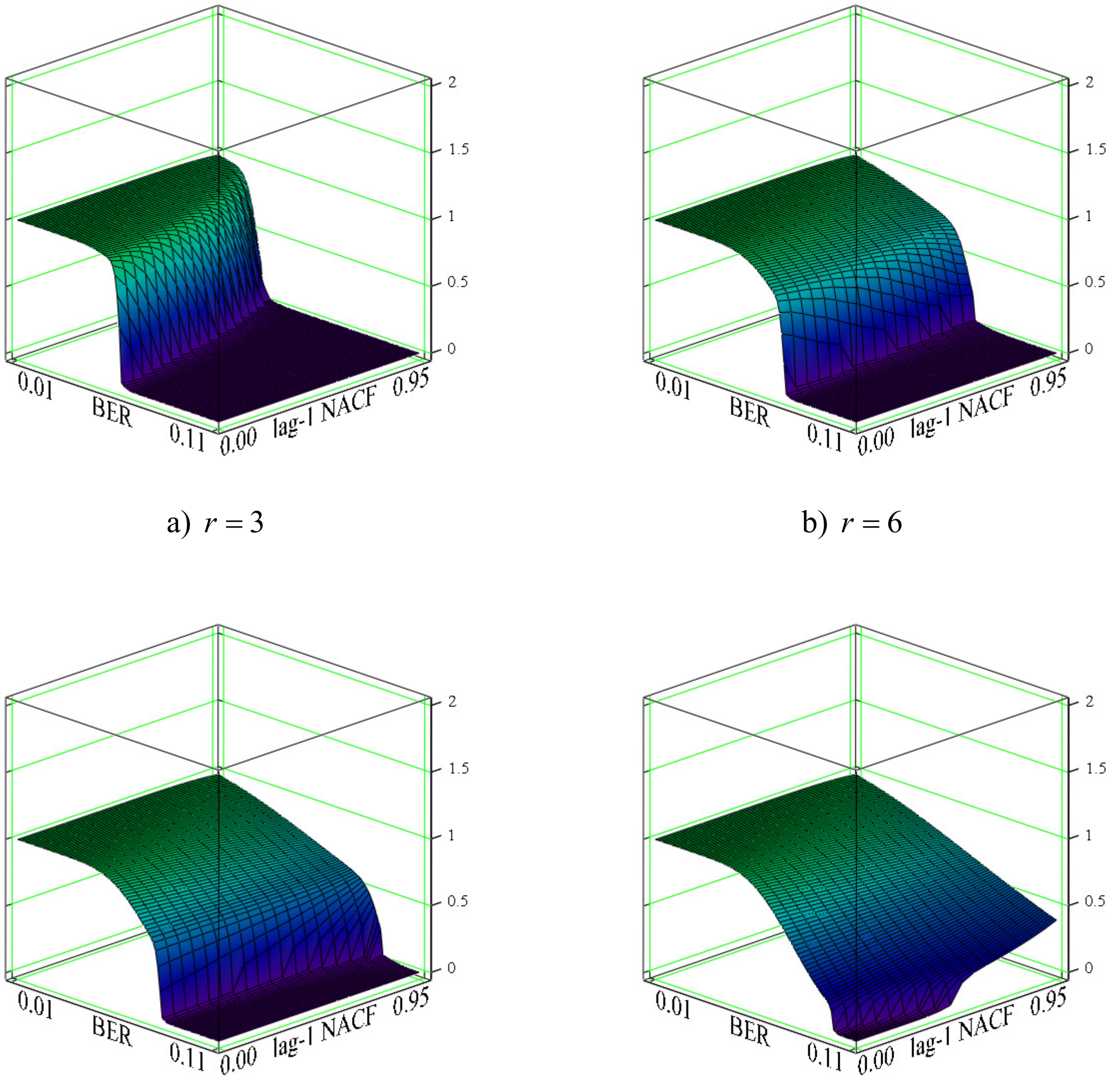
e) (255,131,18), $K_E(1) = 0.95$

f) (255,87,26), $K_E(1) = 0.95$

Fig. 4. 31 Packet loss rate due to an excessive number of transmission attempts

4.6.2.2 TCP SACK steady-state throughput

The TCP SACK long-term steady-state throughput as a function of the BER, the lag-1 NACF, and the persistency of the ARQ scheme for the (255,131,18) and (255,87,26) FEC codes is shown in Fig. 4.32 and Fig. 4.33, respectively. The major effect on the TCP steady-state throughput is produced by the BER. However, influence of the lag-1 autocorrelation of the bit error process is also noticeable. When only three transmission attempts are allowed for a single frame ($r = 3$), the TCP steady-state throughput drops significantly even for a moderate BER. As it follows from Fig. 4.32 and Fig 4.33, the steady-state throughput of a TCP SACK connection running over a noisy wireless channel is an increasing function of the persistency of the ARQ scheme.



a) $r = 3$

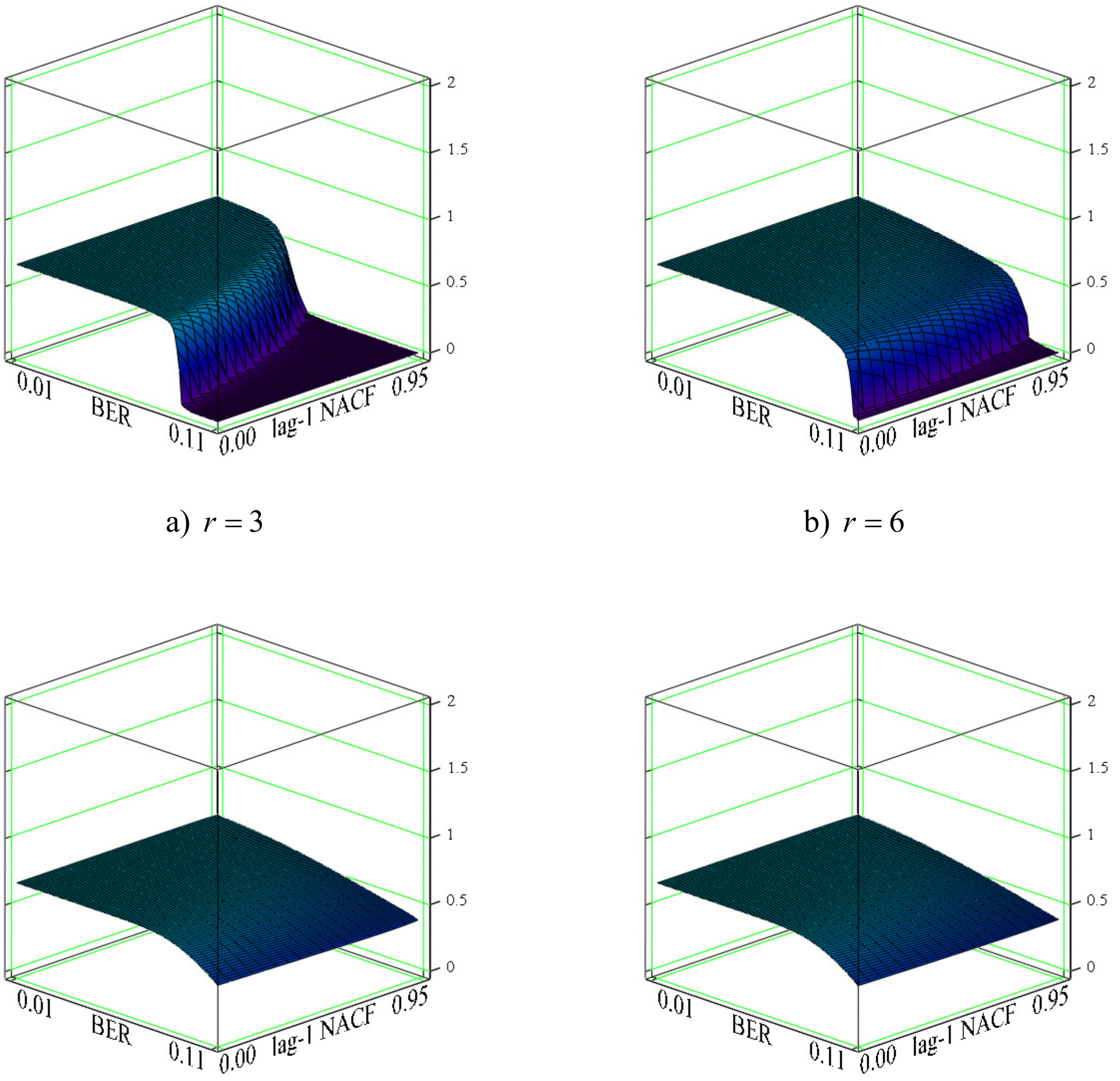
b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 32 TCP SACK steady-state throughput (in Mbit/s), (255,131,18)

As expected, the (255,131,18) FEC code results in better performance for small values of the BER. However, when the BER increases, the (255,87,26) FEC code leads to a higher TCP throughput compared to the (255,131,18) FEC code at the expense of greater code redundancy.



a) $r = 3$

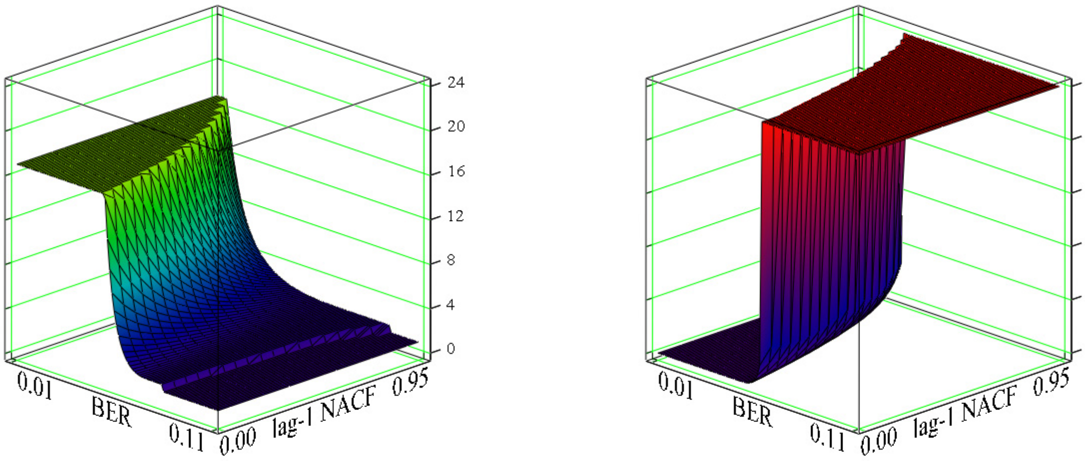
b) $r = 6$

c) $r = 9$

d) $r = 30$

Fig. 4. 33 TCP SACK steady-state throughput (in Mbit/s), (255,87,26)

Considering Fig. 4.32 and Fig. 4.33, we notice a sharp drop in the TCP SACK steady-state throughput which occurs when the BER is high and the ARQ scheme is low-persistent. This behavior can be explained as follows. Firstly, the observed decrease in the TCP throughput corresponds one-to-one to the growth in the packet loss rate due to an excessive number of transmission attempts (see Fig. 4.29 and Fig. 4.30). Indeed, when the packet loss rate is very high ($p_L \to 1$), so almost every transmitted packet will be lost due to an excessive number of transmission attempts made for one of its frames, the TCP throughput tends to zero. Secondly, when non-congestion losses are sufficiently rare ($p_C > p_L$), the window size at loss events belongs to the range from $(C+B)/2$ to $C+B$ and, on average, the TCP sender gets enough duplicate ACKs to trigger the SACK-based loss recovery algorithm (see Fig. 4.34a). However, when non-congestion losses dominate congestion losses ($p_C \le p_L$), the window size at loss events is inversely proportional to p_L and the probability that a non-congestion loss will be detected via a timeout event rapidly increases (see Fig. 4.34b). In extreme case, every non-congestion loss will be followed by a timeout. Thus, high values of the packet loss rate due to an excessive number of transmission attempts together with lengthy TCP timeouts contribute a lot to the degradation of TCP performance over wireless channels with semi-reliable ARQ/FEC.



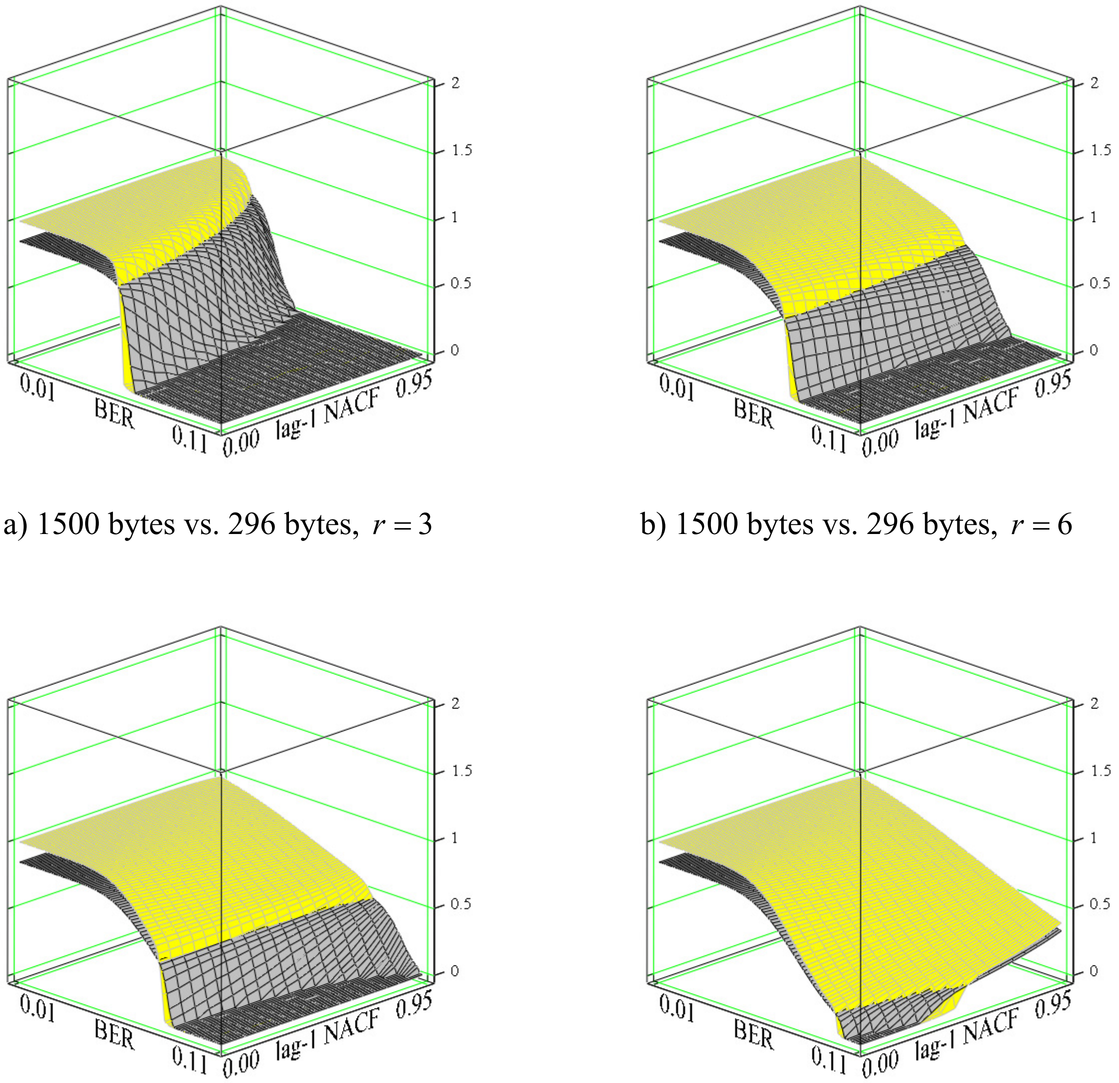
a) The expected window size at loss events

b) The timeout probability

Fig. 4. 34 TCP SACK performance, (255,131,18), $r = 3$

As it was mentioned in section 4.4, the buffer at the intermediate system should be sized at least as large as the end-to-end path capacity ($B \ge C$), while ensuring that the window size at loss events due to buffer overflow is large enough to trigger a fast retransmission without the need to wait for a timeout event ($B + C \ge 3$). At the same time, setting the size of the buffer to be much larger than the end-to-end path capacity results in very long queuing delays, since TCP does its best to fill the end-to-end path capacity and the bottleneck link buffer, thus increasing the queue size and, consequently, the queuing delay. In turn, it leads to large values of the RTT and the RTO. It should be also emphasized that such overprovisioning does not increase the TCP throughput, which is mainly determined by the packet service process of the wireless channel (see Fig. 4.13).

Fig. 4.35 and Fig. 4.36 demonstrate the impact of the MTU size on the TCP SACK steady-state throughput for the (255,131,18) and (255,87,26) FEC codes, respectively. Here the yellow (light) planes denote the case of $\mathrm{MTU} = 1500$ bytes and the grey (dark) planes correspond to the case of $\mathrm{MTU} = 296$ bytes. It is easy to see that when the BER is small, the decrease in the MTU size results in a smaller TCP throughput due to increased protocol header overhead: from $40/1500 \approx 0.027$ to $40/296 \approx 0.135$.



a) 1500 bytes vs. 296 bytes, $r = 3$

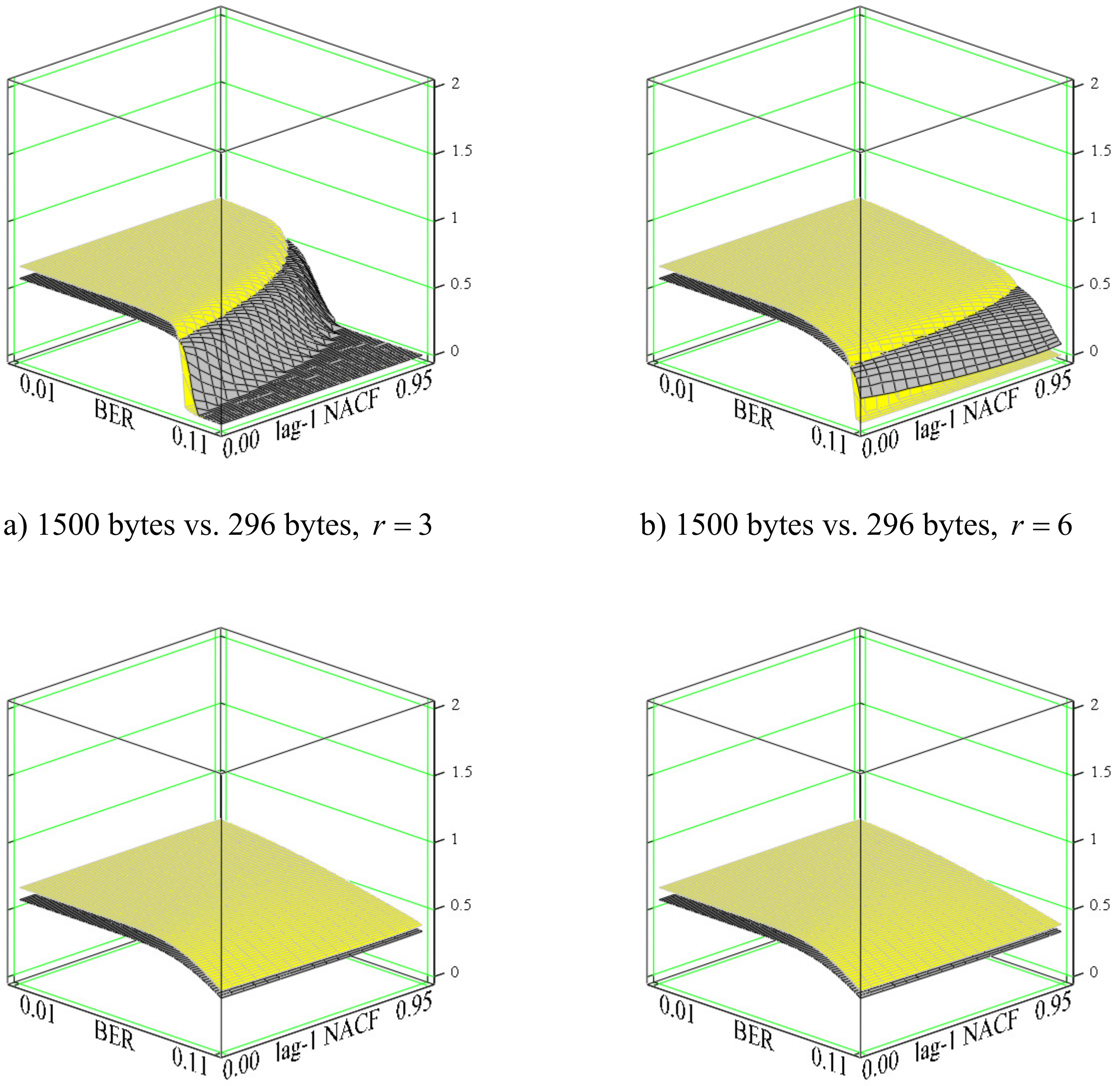
b) 1500 bytes vs. 296 bytes, $r = 6$

c) 1500 bytes vs. 296 bytes, $r = 9$

d) 1500 bytes vs. 296 bytes, $r = 30$

Fig. 4. 35 TCP SACK steady-state throughput (in Mbit/s) for different MTU sizes, (255,131,18)

On the other hand, as it was shown in [P6], the packet loss rate due to an excessive number of transmission attempts (p_L) depends on the number of frames per packet (v). Thus, when the wireless channel quality is relatively “bad”, by decreasing the packet size and, consequently, the number of frames to which an IP packet is segmented, we can slightly reduce non-congestion losses and improve TCP performance. Of course, this effect does not take place in case of a high-persistence ARQ scheme combined with a powerful FEC code (see Fig. 4.36c and Fig. 4.36d), since such a combination ensures that the service provided by the data link layer is highly reliable and the packet loss rate due to an excessive number of transmission attempts tends to zero.



a) 1500 bytes vs. 296 bytes, $r = 3$

b) 1500 bytes vs. 296 bytes, $r = 6$

c) 1500 bytes vs. 296 bytes, $r = 9$

d) 1500 bytes vs. 296 bytes, $r = 30$

Fig. 4. 36 TCP SACK steady-state throughput (in Mbit/s)
for different MTU sizes, (255,87,26)

4.7 Conclusions

In this chapter, we presented an analytical cross-layer model for a TCP SACK connection running over a wireless channel either with completely reliable or semi-reliable ARQ/FEC. The proposed model allows to evaluate the combined effect of many implementation-specific parameters on TCP performance over both correlated and uncorrelated wireless channels, which makes it suitable for performance optimization studies. These parameters include the BER, the lag-1 NACF of bit error observations, the size of PDUs at different layers, the strength of the FEC code, the persistency of the ARQ scheme, the raw data rate of the wireless channel, and the bottleneck link buffer size. It should be emphasized that the developed model is a general framework rather than a model for a particular wireless technology. To use it for practical evaluation of different technologies, this framework should be extended by adding specific details of state-of-the-art wireless systems.

The results of the study allow us to make the following conclusions:

- Although the major effect on the performance of a wireless channel with ARQ/FEC is produced by the BER, the lag-1 autocorrelation of the bit error process can significantly alter the reliability of the channel. Specifically, a bit error process with a high degree of autocorrelation at lag 1 results in better performance of a high-persistence or perfectly-persistent ARQ scheme, while considerably degrading the wireless channel quality for ARQ schemes with low persistency.

- Since a large number of transmission attempts allowed for a frame greatly improves the reliability of a wireless channel by reducing the number of non-congestion losses, a high-persistence or perfectly-persistent ARQ scheme is the best choice for TCP data flows.

- The amount of FEC, required to maximize TCP performance, depends on both BER and lag-1 NACF of the bit error process. However, FEC codes with different codeword lengths but the same code rate provide almost similar performance.

- To ensure that the window size at loss events due to buffer overflow is large enough to trigger a fast retransmission, the buffer at the intermediate system should be sized at least as large as the end-to-end path capacity. However, setting the size of the buffer to be much larger than this value does not increase the TCP steady-state throughput, which is mainly determined by the packet service process of the wireless channel.

- When all packet losses (due to both buffer overflow at the IP layer and an excessive number of transmission attempts at the data link layer) can be recovered using the SACK-based loss recovery algorithm, the TCP throughput is mainly determined by the time required to transmit an IP packet over the wireless channel. But once the window size at loss events becomes less than four segments, lengthy TCP timeouts contribute a lot to the degradation of TCP performance.

- Using small packet sizes reduces the TCP throughput by increasing the protocol header overheard but can slightly improve TCP performance when the wireless channel quality is relatively “bad” and the ARQ scheme is low-persistent. This is because a small packet size implies fewer frames per packet and, consequently, a smaller packet loss probability due to an excessive number of transmission attempts made for one of these frames.

5. SUMMARY OF PUBLICATIONS

This chapter summarizes the publications incorporated in this dissertation and describes the author's contribution to them. In accordance with the structure of the thesis, all the publications can be divided into two groups. The first group contains publications concerning TCP performance evaluation and modeling over wired networks [P1] [P2] [P3] [P4]. The second group includes publications about TCP performance over wired-cum-wireless networks [P5] [P6].

5.1 Overview of the Publications

[P1] R. Dunaytsev, Y. Koucheryavy, J. Harju, The impact of RTT and delayed ACK timeout ratio on the initial slow start phase, in: Proceedings of IPS-MoMe 2005, Warsaw, Poland, March 2005, pp. 171-176.

Description

According to recent measurements, the Internet traffic is dominated by short-lived flows, i.e., flows that are short enough to experience any losses and that spend the most part of their lifetime in the initial slow start phase. Therefore, it is important to estimate how the TCP initial slow start affects the performance of short data transfers. In this paper, the impact of the RTT and delayed ACK timeout ratio on the cwnd increase pattern is analyzed. Then, a comprehensive analytical model of the initial slow start phase is introduced.

[P2] R. Dunaytsev, Y. Koucheryavy, J. Harju, The PFTK-model revised, Computer Communications 29 (13-14) (2006) 2671-2679.

This is an extended version of the following paper:

> R. Dunaytsev, Y. Koucheryavy, J. Harju, Refined PFTK-model of TCP Reno throughput in the presence of correlated losses, in: Proceedings of WWIC 2005, Xanthi, Greece, May 2005, pp. 42-53.

Description

This paper presents an analytical model of the steady-state throughput of a TCP Reno connection as a function of the loss event rate, the mean RTT, the expected duration of the RTO, and the rwnd size based on the model proposed by Padhye *et al.* (widely known as the PFTK-model in correspondence with the initials of the authors) [67]. The presented model refines the previous work by careful examination of fast retransmit/fast recovery dynamics in the presence of correlated losses and taking into consideration the slow start phase after a timeout event.

[P3] R. Dunaytsev, Y. Koucheryavy, J. Harju, TCP NewReno throughput in the presence of correlated losses: the Slow-but-Steady variant, in: Proceedings of IEEE INFOCOM Global Internet Workshop 2006, Barcelona, Spain, April 2006, pp. 115-120.

Description

Recent studies [48] show that the most widely used TCP implementation in today's Internet is TCP NewReno and its deployment has increased significantly in the last few years. However, the majority of the proposed analytical models were developed for the TCP Reno implementation. This paper presents an analytical model of the steady-state throughput of a TCP connection based on the Slow-but-Steady variant of TCP NewReno as a function of the loss event rate, the average loss burst length, the mean RTT, and the expected duration of the RTO.

[P4] R. Dunaytsev, K. Avrachenkov, Y. Koucheryavy, J. Harju, An analytical comparison of the Slow-but-Steady and Impatient variants of TCP NewReno, in: Proceedings of WWIC 2007, Coimbra, Portugal, May 2007, pp. 30-42.

Description

The current standard [49] defines two variants of TCP NewReno: Slow-but-Steady and Impatient. While the behavior of various TCP implementations has been extensively studied over the last years, little attention has been paid to performance analysis of these variants of TCP NewReno. In this paper, an analytical model of the Impatient variant is presented, which, being combined with the earlier proposed model of the Slow-but-Steady variant [P3], gives a comprehensive analytical model of TCP NewReno performance. This model provides the possibility to study the steady-state throughput of both variants over the entire range of operating conditions and protocol settings.

[P5] R. Dunaytsev, D. Moltchanov, Y. Koucheryavy, J. Harju, Modeling TCP SACK performance over wireless channels with completely reliable ARQ/FEC, *Submitted for publication in International Journal of Communication Systems*.

This is a revised and substantially extended version of the following paper:

D. Moltchanov, R. Dunaytsev, Y. Koucheryavy, Cross-layer modeling of TCP SACK performance over wireless channels with completely reliable ARQ/FEC, in: Proceedings of WWIC 2008, Tampere, Finland, May 2008, pp. 13-26.

Description

In this paper, an analytical cross-layer model for a TCP SACK connection running over a covariance-stationary wireless channel with completely reliable ARQ/FEC is introduced. The model allows to evaluate the joint effect of many parameters of wireless channels on TCP performance making it suitable for performance optimization studies. These parameters include the performance characteristics of the wireless channel, the size of PDUs at different layers, the strength of the FEC code, the use of ARQ, the raw data rate of the wireless channel, and the bottleneck link buffer size.

[P6] D. Moltchanov, R. Dunaytsev, Modeling TCP SACK performance over wireless channels with semi-reliable ARQ/FEC, *Accepted for publication in Wireless Networks* (DOI: 10.1007/s11276-009-0231-9).

This is a revised and substantially extended version of the following paper:

D. Moltchanov, R. Dunaytsev, Modeling TCP performance over wireless channels with a semi-reliable data link layer, in: Proceedings of IEEE ICCS 2008, Guangzhou, China, November 2008, pp. 912-918.

Description

Most analytical models that studied the effect of ARQ and FEC on TCP performance assumed that the ARQ scheme is perfectly-persistent (i.e., completely reliable), thus a frame is always successfully transmitted irrespective of the number of transmission attempts it takes. This paper presents an analytical cross-layer model for data transmission over a wireless channel that explicitly takes into account the effect of a semi-reliable data link layer. Packet losses are allowed to occur either due to buffer overflow at the IP layer or due to an excessive number of transmission attempts at the data link layer. The performance metric of interest is the steady-state throughput of a TCP SACK connection running over a wireless channel with semi-reliable ARQ/FEC. The input parameters include the BER, the lag-1 NACF of bit error observations, the strength of the FEC code, the persistency of ARQ, the size of PDUs at different layers, the raw data rate of the wireless channel, and the bottleneck link buffer size.

5.2 Author's Contribution to the Publications

The author's contribution to all the publications included in this dissertation is significant. He has been the primary author in publications [P1] [P2] [P3] [P4] and has contributed a lot to the content of publications [P5] [P6]. The main contribution of the author to these publications is as follows.

In [P1], the author analyzed the impact of different RTT and delayed ACK timeout ratios on the cwnd increase pattern and proposed a comprehensive analytical model of the TCP initial slow start phase. The developed model takes into consideration both cases: when the RTT is smaller than or equal to the doubled value of the delayed ACK timeout and when the RTT is bigger than the last one.

In [P2], the author pointed out several mistakes in the PFTK-model and revised the model. The ns-2 simulation results show that the new model gives a more accurate estimate of TCP Reno throughput in the presence of correlated losses than the original one. Since new TCP models are often compared with the PFTK-model and use its resultant formula, such inaccuracy in throughput estimation can potentially lead to incorrect results or wrong conclusions.

In [P3], the author developed an analytical model of the steady-state throughput of a long-lived TCP connection based on the Slow-but-Steady variant of TCP NewReno.

In [P4], the author proposed an analytical model of the Impatient variant of TCP NewReno and performed an analytical comparison of the Impatient and Slow-but-Steady throughputs in the presence of correlated losses.

In [P5], the author analyzed the evolution of a long-lived TCP SACK connection running over a wireless channel with completely reliable ARQ/FEC and derived expressions for its steady-state throughput, the mean RTT, and the spurious timeout probability.

In [P6], the author derived an expression for the steady-state throughput of a long-lived TCP SACK connection running over a wireless channel with semi-reliable ARQ/FEC.

It should be emphasized that the wireless channel models used in [P5] [P6] have been developed by Dmitri Moltchanov.

6. CONCLUSIONS

From the early days of BSD Unix systems to desktop and server platforms of today, the Internet Protocol Suite, and, hence, TCP is an integral part of any OS. Moreover, from the very beginning of the Internet, it has been widely used by the most popular applications and services: from the File Transfer Protocol (FTP) and Usenet before the dot com era, and up to the World Wide Web (WWW) and peer-to-peer (P2P) file sharing nowadays. As a consequence, TCP controls about 90% of all bytes and packets carried over the Internet and there are no indications that its share will significantly decline in the nearest future. Due to its widespread use, TCP performance has been extensively studied over the last decade. Analytical modeling has proven to be a powerful and cost-effective method for examining the behavior of TCP. To be useful, TCP analytical models should be realistic and capture the most important TCP algorithms such as slow start, congestion avoidance, fast retransmit and fast recovery, etc. In this dissertation, we made an effort towards a better understanding of various aspects of TCP performance under different conditions and in different environments. The contribution of the thesis includes the development of the following models:

- a model for the TCP initial slow start phase. In networks with large bandwidth and long delay, if the initial ssthresh is set too low relative to the bandwidth-delay product, a TCP connection exits slow start and switches to the congestion avoidance phase with a linear increase of the cwnd size prematurely, resulting in poor utilization of the available bandwidth. Recently, a number of methods have been suggested for improving TCP startup performance. In order to evaluate the efficiency of different proposals, an accurate analytical model of the initial slow start was developed, which can predict the duration of the initial slow start phase and the number of transmitted segments over a wide range of operating conditions.
- a model of TCP Reno throughput under correlated losses. This model is based on the model proposed by Padhye *et al*. (also known as the PFTK-model) and refines it by careful examination of fast retransmit/fast recovery dynamics in case of multiple packet losses within a window of data. We show that though the PFTK-model is very popular and widely referenced, it contains a number of logical contradictions, which can result in significant overestimation of the steady-state throughput of a long-lived TCP Reno connection when packet losses occur in bursts.
- a model of the steady-state throughput of a long-lived TCP NewReno connection. Although TCP performance has been widely investigated in the literature, the majority of the analytical models were developed for TCP Reno. In the absence of sufficient

analytical background, the current standard recommends the Impatient variant of TCP NewReno based only on ns-2 simulations. The proposed model allows to evaluate the performance of both TCP NewReno variants (Slow-but-Steady and Impatient) over a wide range of operating conditions and different protocol settings.

- a model of TCP SACK performance over wireless channels with completely reliable or semi-reliable ARQ/FEC. TCP was initially developed to operate over wired networks, where the packet loss rate due to data corruption is very small, and is known to have poor performance over noisy wireless channels, since packet losses due to transmission errors are misinterpreted as congestion-induced packet drops. To improve communication reliability and reduce packet losses by detecting and recovering corrupted bits, modern wireless technologies take advantage of different error control techniques including ARQ, FEC, and hybrid ARQ/FEC. Being combined with a wireless channel model, the developed model allows to quantify the joint effect of different parameters of the ARQ/FEC scheme in use on TCP performance over both correlated and uncorrelated wireless channels, which makes it suitable for performance optimization studies. Ultimately, the model can be used as the basis for a cross-layer performance control system.

Future work includes extending the presented cross-layer model to incorporate different ARQ and FEC schemes used in modern wireless communication systems. Moreover, it involves experimental work to validate the model by simulations and real-life measurements. We also plan to evaluate the performance of new TCP implementations, such as those developed specifically for high-speed and wireless networks. Another important direction that is currently under investigation is applying a fixed-point method to model the interaction between the network and a number of long-lived TCP flows sharing a wireless bottleneck link.

APPENDIX A

In [P1], to find the number of slow start rounds required to transfer a given number of segments, we used the approach proposed in [79]. In fact, it is a new method of finding solutions of higher degree and transcendental equations developed by M.A. Eremin. Unfortunately, this book is available in Russian only, and, therefore, is mostly unknown outside of the Russian Federation. Thus, in order to provide some insight into this method, we give a short introduction to it, followed by examples of its use from [79].

A.1 Solving Polynomials Equations of Higher Degree

Let us consider a higher degree equation in the canonical form:

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \ldots + a_{n-2} x^2 + a_{n-1} x + a_n = 0, \tag{A.1}$$

where $n \geq 2$; $a_{n-1} \neq 0$; $a_n \neq 0$.

According to [79], this equation has the following determinant:

$$p = \frac{m^2}{m + \frac{a_n}{a_{n-1}}}, \quad p = \frac{-a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}}, \tag{A.2}$$

where $m + \frac{a_n}{a_{n-1}} \neq 0$; $m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} \neq 0$.

Theorem 1: If m is the real root of equation (A.1), then

$$p = \frac{m^2}{m + \frac{a_n}{a_{n-1}}} = \frac{-a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}}. \tag{A.3}$$

Proof: As it follows from (A.3)

$$\begin{aligned} &\frac{m^2}{m + \frac{a_n}{a_{n-1}}} - \frac{-a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}} = \\ &= \frac{m^2}{m + \frac{a_n}{a_{n-1}}} + \frac{a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}} = 0. \end{aligned} \tag{A.4}$$

Then

$$m^2 \left(m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} \right) + a_{n-1} \left(m + \frac{a_n}{a_{n-1}} \right) = 0. \tag{A.5}$$

Finally, we get:

$$m^n + a_1 m^{n-1} + a_2 m^{n-2} + \ldots + a_{n-2} m^2 + a_{n-1} m + a_n = 0. \tag{A.6}$$

Since at $x = m$ equation (A.1) becomes zero, then m is the root of (A.1).

Corollary: Based on Theorem 1, we can find the intervals containing real roots of equation (A.1). In order to so, we need to solve the following sets of inequalities for $p > 0$

$$\begin{cases} m + \dfrac{a_n}{a_{n-1}} > 0, \\ m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} < 0, \quad \text{if}\, a_{n-1} > 0 \end{cases}$$

$$\begin{cases} m + \dfrac{a_n}{a_{n-1}} > 0, \\ m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} > 0, \quad \text{if}\, a_{n-1} < 0, \end{cases} \tag{A.7}$$

and for $p < 0$

$$\begin{cases} m + \dfrac{a_n}{a_{n-1}} < 0, \\ m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} > 0, \quad \text{if}\, a_{n-1} > 0, \end{cases}$$

$$\begin{cases} m + \dfrac{a_n}{a_{n-1}} < 0, \\ m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} < 0, \quad \text{if}\, a_{n-1} < 0. \end{cases} \tag{A.8}$$

Example 1: Let us define the interval containing real roots of the following equation [79, p.36]:

$$x^4 + 2x^3 - 4x^2 - 5x - 6 = 0. \tag{A.9}$$

Observing that $n = 4$ and using (A.2), we obtain:

$$p = \frac{m^2}{m + \dfrac{-6}{-5}} = \frac{m^2}{m + \dfrac{6}{5}}, \quad p = \frac{-(-5)}{m^{4-2} + 2m^{4-3} + (-4) m^{4-4}} = \frac{5}{m^2 + 2m - 4}. \tag{A.10}$$

Then we get two sets of inequalities:

$$p > 0, \qquad \begin{cases} m + \dfrac{6}{5} > 0, \\ m^2 + 2m - 4 > 0, \end{cases} \qquad \begin{cases} m > -\dfrac{6}{5}, \\ m > -1 + \sqrt{5}, \end{cases} \qquad m > -1 + \sqrt{5}, \tag{A.11}$$

and

$$p < 0, \qquad \begin{cases} m + \dfrac{6}{5} < 0, \\ m^2 + 2m - 4 < 0, \end{cases} \qquad \begin{cases} m < -\dfrac{6}{5}, \\ -1 - \sqrt{5} < m < -1 + \sqrt{5}, \end{cases} \qquad -1 - \sqrt{5} < m < -\frac{6}{5}. \tag{A.12}$$

Hence, the roots of equation (A.9) lie in the intervals $-3.2361 < m < -1.2$ and $m > 1.2$ (Fig. A.1).

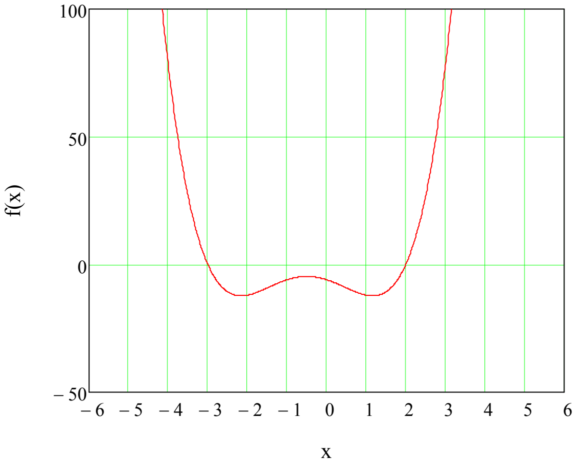


Fig. A. 1 $f(x) = x^4 + 2x^3 - 4x^2 - 5x - 6$

To find the exact values of the roots, let us compute p in these intervals. For $m = -2$, we have:

$$p = \frac{(-2)^2}{-2 + \frac{6}{5}} = -\frac{20}{4} = -5, \quad p = \frac{5}{(-2)^2 + 2(-2) - 4} = -\frac{5}{4}. \tag{A.13}$$

And for $m = -3$, we get:

$$p = \frac{(-3)^2}{-3 + \frac{6}{5}} = -\frac{45}{9} = -5, \quad p = \frac{5}{(-3)^2 + 2(-3) - 4} = -5. \tag{A.14}$$

Thus, according to Theorem 1, the first real root is $m = -3$.

Now let us check the interval $m > -1 + \sqrt{5}$. For $m = 2$, we obtain the second real root of equation (A.9):

$$p = \frac{2^2}{2 + \frac{6}{5}} = \frac{20}{16} = \frac{5}{4}, \quad p = \frac{5}{2^2 + 2(2) - 4} = \frac{5}{4}. \tag{A.15}$$

Substituting $m = 3, 4, 5, \ldots$, we get that while $p = \dfrac{m^2}{m + 6/5}$ increases with m, $p = \dfrac{5}{m^2 + 2m - 4}$ decreases. Therefore, we conclude that there are no real roots in the interval $m > 2$. This can be proved by decomposing (A.9) as follows:

$$x^4 + 2x^3 - 4x^2 - 5x - 6 = (x+3)(x-2)(x^2 + x + 1) = 0. \tag{A.16}$$

Theorem 2: Let m be the range of values for the following sets of inequalities:

$$\begin{cases} p = \dfrac{m^2}{m + \dfrac{a_n}{a_{n-1}}} > 0, \\ p = \dfrac{-a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}} > 0, \end{cases}$$

$$\begin{cases} p = \dfrac{m^2}{m + \dfrac{a_n}{a_{n-1}}} < 0, \\ p = \dfrac{-a_{n-1}}{m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2}} < 0, \end{cases} \tag{A.17}$$

where $m + \dfrac{a_n}{a_{n-1}} \neq 0$; $m^{n-2} + a_1 m^{n-3} + a_2 m^{n-4} + \ldots + a_{n-2} \neq 0$.

If on $[m_1, m_2]$, belonging to m, expressions (A.18) differ in sign, then there is a real root of equation (A.1) in this region.

$$\Delta p(m_1) = \frac{m_1^{\,2}}{m_1 + \dfrac{a_n}{a_{n-1}}} - \frac{-a_{n-1}}{m_1^{\,n-2} + a_1 m_1^{\,n-3} + a_2 m_1^{\,n-4} + \ldots + a_{n-2}},$$

$$\Delta p(m_2) = \frac{m_2^{\,2}}{m_2 + \dfrac{a_n}{a_{n-1}}} - \frac{-a_{n-1}}{m_2^{\,n-2} + a_1 m_2^{\,n-3} + a_2 m_2^{\,n-4} + \ldots + a_{n-2}}. \tag{A.18}$$

Proof: First of all, it is important to note that expressions (A.2) are fractional rational functions. These functions are continuous in all points m for which the denominator is not zero. Hence, $\Delta p(m)$ is a continuous function as well (if two functions are continuous, then their sum is also continuous). And according to Cauchy's theorem, if $f(x)$ is continuous on $[a, b]$ and $f(a)$ and $f(b)$ differ in sign, then, at some point $x_0 \in [a, b]$, $f(x_0)$ must equal zero.

A.2 Solving Transcendental Equations

Let us consider a transcendental equation in the canonical form:

$$x^n + a_1 x^{n-1} + a_2 x^{n-2} + \ldots + a_{n-d} x^d + a_{n-q} x^q + f(x) + a_n = 0, \tag{A.19}$$

where $f(x)$ is a transcendental function.

According to [79], this equation has the following determinant:

$$p = \frac{m^d}{m^q + \dfrac{a_n + f(m)}{a_{n-q}}}, \quad p = \frac{-a_{n-q}}{m^{n-d} + a_1 m^{n-(d+1)} + a_2 m^{n-(d+2)} + \ldots + a_{n-d}}, \tag{A.20}$$

where $m^q + \dfrac{a_n + f(m)}{a_{n-q}} \neq 0$; $m^{n-d} + a_1 m^{n-(d+1)} + a_2 m^{n-(d+2)} + \ldots + a_{n-d} \neq 0$.

Example 2: Let us define the interval containing real roots of the following equation [79, p.215]:

$$x^3 + 8x^2 + 12x - 2^{x^2} + 8 = 0. \tag{A.21}$$

Observing that $f(x) = -2^{x^2}$ and using (A.20), we obtain:

$$p = \frac{m^2}{m^1 + \dfrac{8 - 2^{m^2}}{12}} = \frac{m^2}{m + \dfrac{8 - 2^{m^2}}{12}}, \quad p = \frac{-12}{m^{3-2} + 8m^{3-(2+1)}} = \frac{-12}{m+8}. \tag{A.22}$$

Based on Theorem 1, we can find the intervals containing real roots of equation (A.21). We have:

$$p > 0, \qquad \begin{cases} m + \dfrac{8 - 2^{m^2}}{12} > 0, \\ m + 8 < 0, \end{cases} \qquad \begin{cases} 12m + 8 > 2^{m^2}, \\ m < -8, \end{cases} \qquad \text{no solutions}, \tag{A.23}$$

and

$$p < 0, \qquad \begin{cases} m + \dfrac{8 - 2^{m^2}}{12} < 0, \\ m + 8 > 0, \end{cases} \qquad \begin{cases} 12m + 8 < 2^{m^2}, \\ m > -8, \end{cases} \qquad \begin{cases} m > 2.44, \\ -8 < m < -0.6. \end{cases} \tag{A.24}$$

Hence, the roots of equation (A.21) lie in the intervals $-8 < m < -0.6$ and $m > 2.44$ (Fig. A.2).

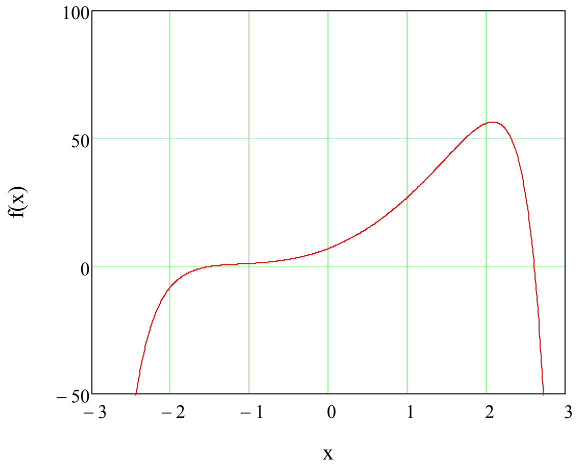


Fig. A. 2 $f(x) = x^3 + 8x^2 + 12x - 2^{x^2} + 8$

Based on Theorem 2, we can further specify the intervals containing the real roots of equation (A.21). As an example, let us consider the interval $m > 2.44$. We compute $\Delta p(m)$ for $m_1 = 2.5$ and $m_2 = 2.7$:

$$\Delta p(m_1) = \Delta p(2.5) = \frac{2.5^2}{2.5 + \frac{8 - 2^{2.5^2}}{12}} - \frac{-12}{2.5 + 8} = \frac{2.5^2}{2.5 + \frac{8 - 2^{2.5^2}}{12}} + \frac{12}{2.5 + 8} \approx -0.825,$$

$$\Delta p(m_2) = \Delta p(2.7) = \frac{2.7^2}{2.7 + \frac{8 - 2^{2.7^2}}{12}} - \frac{-12}{2.5 + 8} = \frac{2.7^2}{2.7 + \frac{8 - 2^{2.7^2}}{12}} + \frac{12}{2.7 + 8} \approx 0.368. \tag{A.25}$$

It is easy to see that the root belongs to $[2.5, 2.7]$, since $\Delta p(2.5)$ and $\Delta p(2.7)$ differ in sign. We then compute $\Delta p(m)$ for $m_1 = 2.6$ and $m_2 = 2.7$:

$$\Delta p(m_1) = \Delta p(2.6) = \frac{2.6^2}{2.6 + \frac{8 - 2^{2.6^2}}{12}} + \frac{12}{2.6 + 8} \approx -0.04,$$

$$\Delta p(m_2) = \Delta p(2.7) = \frac{2.7^2}{2.7 + \frac{8 - 2^{2.7^2}}{12}} + \frac{12}{2.7 + 8} \approx 0.368. \tag{A.26}$$

Again we note that $\Delta p(2.6)$ and $\Delta p(2.7)$ differ in sign, thus the root lies in $[2.6, 2.7]$.

Note that these steps can be repeated as needed to achieve the required accuracy. For instance, let us consider $[2.6076, 2.6077]$ and $[2.6077, 2.6078]$. We get:

$$\Delta p(2.6076) = \frac{2.6076^2}{2.6076 + \frac{8 - 2^{2.6076^2}}{12}} + \frac{12}{2.6076 + 8} \approx -0.0003,$$

$$\Delta p(2.6077) = \frac{2.6077^2}{2.6077 + \frac{8 - 2^{2.6077^2}}{12}} + \frac{12}{2.6077 + 8} \approx 0.0002, \tag{A.27}$$

$$\Delta p(2.6078) = \frac{2.6078^2}{2.6078 + \frac{8 - 2^{2.6078^2}}{12}} + \frac{12}{2.6078 + 8} \approx 0.0007.$$

Since $\Delta p(m)$ reverses its sign only in $[2.6076, 2.6077]$, then the root belongs to this interval.

BIBLIOGRAPHY

[1] V. Cerf, R. Kahn, A protocol for packet network intercommunication, IEEE Transactions on Communication 22 (5) (1974) 637-648.

[2] R. Zakon, Hobbes' Internet Timeline, RFC 2235, IETF, November 1997.

[3] V. Cerf, Y. Dalal, C. Sunshine, Specification of Internet Transmission Control Program, RFC 675, IETF, December 1974.

[4] V. Cerf, Specification of Internet Transmission Control Program TCP (version 2), http://www.cs.utexas.edu/users/chris/DIGITAL_ARCHIVE/TCPIP/IEN5.pdf

[5] J. Postel (Ed.), DoD Standard Transmission Control Protocol, RFC 761, IETF, January 1980.

[6] J. Postel (Ed.), Transmission Control Protocol, RFC 793, IETF, September 1981.

[7] J. Postel (Ed.), Internet Protocol, RFC 791, IETF, September 1981.

[8] ITU-T Recommendation X.200, Information Technology – Open Systems Interconnection – Basic Reference Model: the Basic Model, January 1994.

[9] J. Nagle, Congestion Control in IP/TCP Internetworks, RFC 896, IETF, January 1984.

[10] V. Jacobson, Congestion avoidance and control, ACM SIGCOMM Computer Communication Review 25 (1) (1988) 157-187.

[11] W. Stevens, TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, RFC 2001, IETF, January 1997.

[12] H. Schulze, K. Mochalski, Internet study 2008/2009, ipoque GmbH, 2009, http://www.ipoque.com/study/ipoque-Internet-Study-08-09.pdf

[13] Internet2 NetFlow: Weekly reports, http://netflow.internet2.edu/weekly/

[14] CAIDA's traffic analysis research: Analyzing UDP usage in Internet traffic, http://www.caida.org/research/traffic-analysis/tcpudpratio/

[15] C. Barakat, E. Altman, W. Dabbous, On TCP performance in a heterogeneous network: a survey, IEEE Communications Magazine 38 (1) (2000) 40-46.

[16] H. Inamura, G. Montenegro (Eds.), TCP over Second (2.5G) and Third (3G) Generation Wireless Networks, RFC 3481, IETF, February 2003.

[17] Y. Tian, K. Xu, N. Ansari, TCP in wireless environments: problems and solutions, IEEE Communications Magazine 43 (3) (2005) S27-S32.

[18] G. Appenzeller, I. Keslassy, N. McKeown, Sizing router buffers, ACM SIGCOMM Computer Communication Review 34 (4) (2004) 281-292.

[19] L. Guo, I. Matta, The war between mice and elephants, in: Proceedings of ICNP 2001, Riverside, CA, November 2001, pp. 180-188.

[20] S. Jin, L. Guo, I. Matta, A. Bestavros, A spectrum of TCP-friendly window-based congestion control algorithms, IEEE/ACM Transactions on Networking 11 (3) (2003) 341-355.

[21] S. Floyd, M. Handley, J. Padhye, J. Widmer, TCP Friendly Rate Control (TFRC): Protocol Specification, RFC 5348, IETF, September 2008.

[22] V. Srivastava, M. Motani, Cross-layer design: a survey and the road ahead, IEEE Communications Magazine 43 (12) (2005) 112-119.

[23] D. Moltchanov, Cross-layer performance evaluation and control of wireless channels in next generation all-IP networks, PhD thesis, Tampere University of Technology, Finland, 2006.

[24] Q. He, M. Ammar, G. Riley, R. Fujimoto, Exploiting the predictability of TCP steady-state to speed up network simulation, Performance Evaluation 58 (2-3) (2004) 163-187.

[25] I. Khalifa, L. Trajković, An overview and comparison of analytical TCP models, in: Proceedings of IEEE ISCAS 2004, Vancouver, Canada, May 2004, pp. 469-472.

[26] J. Olsén, Stochastic modeling and simulation of the TCP protocol, PhD thesis, Uppsala University, Sweden, 2003.

[27] P. Lassila, M. Mandjes, A multi-level TCP model with heterogeneous RTTs, in: Proceedings of IFIP TC6 Networking 2004, Athens, Greece, May 2004, pp. 52-63.

[28] R. Braden (Ed.), Requirements for Internet Hosts – Communication Layers, RFC 1122, IETF, October 1989.

[29] M. Allman, V. Paxson, W. Stevens, TCP Congestion Control, RFC 2581, IETF, April 1999.

[30] RFC Index Search Engine, http://www.rfc-editor.org/rfcsearch.html

[31] M. Duke, R. Braden, W. Eddy, E. Blanton, A Roadmap for Transmission Control Protocol (TCP) Specification Documents, RFC 4614, IETF, September 2006.

[32] L. Eggert, F. Gont, TCP User Timeout Option, RFC 5482, IETF, March 2009.

[33] P. Sarolahti, M. Kojo, K. Yamamoto, M. Hata, Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP, RFC 5682, IETF, September 2009.

[34] J. Postel, TCP Maximum Segment Size and Related Topics, RFC 879, IETF, November 1983.

[35] D. Clark, Window and Acknowledgment Strategy in TCP, RFC 813, IETF, July 1982.

[36] V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, RFC 1323, IETF, May 1992.

[37] J. Mogul, S. Deering, Path MTU Discovery, RFC 1191, IETF, November 1990.

[38] J. Reynolds, J. Postel, Assigned Numbers, RFC 1700, IETF, October 1994.

[39] R. Braden, D. Borman, C. Partridge, Computing the Internet Checksum, RFC 1071, IETF, September 1988.

[40] SpeedGuide.net: TCP optimizer, http://www.speedguide.net/downloads.php

[41] DrTCP: Windows TCP tuning and tweaking, http://www.dslreports.com/drtcp

[42] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgement Options, RFC 2018, IETF, October 1996.

[43] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, IETF, July 2000.

[44] E. Blanton, M. Allman, K. Fall, L. Wang, A Conservative Selective Acknowledgement (SACK)-based Loss Recovery Algorithm for TCP, RFC 3517, IETF, April 2003.

[45] V. Paxson, M. Allman, Computing TCP's Retransmission Timer, RFC 2988, IETF, November 2000.

[46] P. Karn, C. Partridge, Improving round-trip time estimates in reliable transport protocols, ACM SIGCOMM Computer Communication Review 17 (5) (1987) 2-7.

[47] M. Allman, S. Floyd, C. Partridge, Increasing TCP's Initial Window, RFC 3390, IETF, October 2002.

[48] A. Medina, M. Allman, S. Floyd, Measuring the evolution of transport protocol in the Internet, ACM SIGCOMM Computer Communication Review 35 (2) (2005) 37-52.

[49] S. Floyd, T. Henderson, A. Gurtov, The NewReno Modification to TCP's Fast Recovery Algorithm, RFC 3782, IETF, April 2004.

[50] B. Braden *et al*., Recommendations on Queue Management and Congestion Avoidance in the Internet, RFC 2309, IETF, April 1998.

[51] K. Ramakrishnan, S. Floyd, D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, IETF, September 2001.

[52] L. Grieco, S. Mascolo, Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control, ACM SIGCOMM Computer Communication Review 34 (2) (2004) 25-38.

[53] S. Floyd, HighSpeed TCP for Large Congestion Windows, RFC 3649, IETF, December 2003.

[54] T. Kelly, Scalable TCP: improving performance in highspeed wide area networks, ACM SIGCOMM Computer Communication Review 32 (2) (2003) 83-91.

[55] L. Xu, K. Harfoush, I. Rhee, Binary increase congestion control (BIC) for fast, long-distance networks, in: Proceedings of IEEE INFOCOM 2004, Hong Kong, China, March 2004, pp. 2490-2501.

[56] S. Ha, I. Rhee, L. Xu, CUBIC: a new TCP-friendly high-speed TCP variant, ACM SIGOPS Operating Systems Review 42 (5) (2008) 64-74.

[57] L. Brakmo, S. O'Malley, L. Peterson, TCP Vegas: new techniques for congestion detection and avoidance, ACM SIGCOMM Computer Communication Review 24 (4) (1994) 24-35.

[58] D. Wei, C. Jin, S. Low, S. Hegde, FAST TCP: motivation, architecture, algorithms, performance, IEEE/ACM Transactions on Networking 14 (6) (2006) 1246-1259.

[59] The network simulator ns-2, http://www.isi.edu/nsnam/ns/

[60] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, ACM SIGCOMM Computer Communication Review 26 (3) (1996) 5-21.

[61] J. Padhye, S. Floyd, Identifying the TCP behavior of Web servers, in: ICSI, Technical Report 01-002, February 2001.

[62] W3Schools: OS platform statistics, http://www.w3schools.com/browsers/browsers_os.asp

[63] DistroWatch.com, http://distrowatch.com/stats.php

[64] Wireshark: Go deep, http://www.wireshark.org

[65] M. Mathis, J. Semke, J. Mahdavi, T. Ott, The macroscopic behavior of the TCP congestion avoidance algorithm, ACM SIGCOMM Computer Communication Review 27 (3) (1997) 67-82.

[66] A. Kumar, Comparative performance analysis of versions of TCP in a local network with a lossy link, IEEE/ACM Transactions on Networking 6 (4) (1998) 485-498.

[67] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling TCP Reno performance: a simple model and its empirical validation, IEEE/ACM Transactions on Networking 8 (2) (2000) 133-145.

[68] N. Cardwell, S. Savage, T. Anderson, Modeling TCP latency, in: Proceedings of IEEE INFOCOM 2000, Tel-Aviv, Israel, March 2000, pp. 1742-1751.

[69] B. Sikdar, S. Kalyanaraman, K. Vastola, Analytic models for the latency and steady-state throughput of TCP Tahoe, Reno and SACK, IEEE/ACM Transactions on Networking 11 (6) (2003) 959-971.

[70] V. Paxson, Empirically derived analytic models of wide-area TCP connections, IEEE/ACM Transactions on Networking 2 (4) (1994) 316-336.

[71] K. Thompson, G. Miller, R. Wilder, Wide-area Internet traffic patterns and characteristics, IEEE Network 11 (6) (1997) 10-23.

[72] M. Kim, Y. Won, J. Hong, Characteristic analysis of internet traffic from the perspective of flows, Computer Communications 29 (10) (2006) 1639-1652.

[73] J. Iyengar, A. Caro, P. Amer, Dealing with short TCP flows: a survey of mice in elephant shoes, in: University of Delaware, Technical Report, August 2003.

[74] S. Floyd, V. Paxson, Difficulties in simulating the Internet, IEEE/ACM Transactions on Networking 9 (4) (2001) 392-403.

[75] M. Allman, S. Floyd, C. Partridge, Increasing TCP's Initial Window, RFC 2414, IETF, September 1998.

[76] N. Cardwell, S. Savage, T. Anderson, Modeling the performance of short TCP connections, in: University of Washington, Technical Report, October 1998.

[77] S. Fortin, B. Sericola, A Markovian model for the stationary behavior of TCP, in: IRISA-INRIA, Technical Report RR-4240. September 2001.

[78] D. Zheng, G. Lazarou, R. Hu, A stochastic model for short-lived TCP flows, in: Proceedings of IEEE ICC 2003, Anchorage, AK, May 2003.

[79] M. Eremin, Higher Degree Equations, Arzamas, 2003 (in Russian).

[80] CiteSeer.IST: Scientific Literature Digital Library, http://citeseer.ist.psu.edu/

[81] C. Brandauer, G. Iannaccone, C. Diot, T. Ziegler, S. Fdida, M. May, Comparison of Tail Drop and active queue management performance for bulk-data and Web-like Internet traffic, in: Proceedings of ISCC 2001, Hammamet, Tunisia, July 2001, pp. 122-131.

[82] D. Loguinov, Adaptive scalable Internet streaming, PhD thesis, The City University of New York, USA, 2002.

[83] M. Goyal, R. Guerin, R. Rajan, Predicting TCP throughput from non-invasive network sampling, in: Proceedings of IEEE INFOCOM 2002, New York, NY, June 2002, pp. 180-189.

[84] K. Park, G. Kim, M. Crovella, On the relationship between file sizes, transport protocols and self-similar network traffic, in: Boston University, Technical Report 1996-016, August 1996.

[85] W. Willinger, M. Taqqu, R. Sherman, D. Wilson, Self-similarity through high variability: statistical analysis of Ethernet LAN traffic at the source level, IEEE/ACM Transactions on Networking 5 (1) (1997) 71-86.

[86] N. Parvez, A. Mahanti, C. Williamson, TCP NewReno: Slow-but-Steady or Impatient?, in: Proceedings of IEEE ICC 2006, Istanbul, Turkey, June 2006, pp. 716-722.

[87] C. Joo, S. Bahk, Start-up transition behaviour of TCP NewReno, IEE Electronic Letters 35 (21) (1999) 1818-1820.

[88] S. Kim, S. Choi, C. Kim, Instantaneous variant of TCP NewReno, IEE Electronics Letters 36 (19) (2000) 1669-1670.

[89] Microsoft Windows Server 2003 TCP/IP Implementation Details, Microsoft Corporation, December 2007, http://download.microsoft.com/download/f/0/f/f0f28365-bd9a-4ff8-a5d4-fc0f94ae7371/TCPIP_2003.doc

[90] P. Sarolahti, A. Kuznetsov, Congestion control in Linux TCP, in: Proceedings of USENIX/FREENIX 2002, Monterey, CA, June 2002, pp. 49-62.

[91] G. Fairhurst, L. Wood, Advice to Link Designers on Link Automatic Repeat reQuest (ARQ), RFC 3366, IETF, August 2002.

[92] M. Schiff, Introduction to Communication Systems Simulation. Artech House, Inc., 2006.

[93] Y. Guo, Advances in Mobile Radio Access Networks. Artech House, Inc., 2004.

[94] S. Fu, M. Atiquzzaman, Modelling TCP Reno with spurious timeouts in wireless mobile environments, in: Proceedings of ICCCN 2003, Dallas, TX, October 2003, pp. 391-396.

[95] A. Abouzeid, S. Roy, M. Azizoglou, Comprehensive performance analysis of a TCP session over a wireless fading link with queueing, IEEE Transactions on Wireless Communications 2 (2) (2003) 344-356.

[96] M. Zorzi, R. Rao, The effect of correlated errors on the performance of TCP, IEEE Communication Letters 1 (5) (1997) 127-129.

[97] D. Barman, I. Matta, E. Altman, R. Azouzi, TCP optimization through FEC, ARQ and transmission power tradeoffs, in: Proceedings of WWIC 2004, Frankfurt, Germany, February 2004, pp. 87-98.

[98] C. Barakat, A. Fawal, Analysis of link-level hybrid FEC/ARQ-SR for wireless links and long-lived TCP traffic, Performance Evaluation Journal 57 (4) (2004) 453-476.

[99] Y. Wu, Z. Niu, J. Zheng, Cross-layer analysis of wireless TCP/ARQ systems over correlated channels, Journal of Communication and Networks 7 (1) (2005) 45-53.

[100] F. Vacirca, A. Vendictis, A. Baiocchi, Optimal design of hybrid FEC/ARQ schemes for TCP over wireless links with Rayleigh fading, IEEE Transactions on Mobile Computing 5 (4) (2006) 289-302.

[101] S. Iyer, S. Bhattacharyya, N. Taft, C. Diot, An approach to alleviate link overload as observed on an IP backbone, in: Proceedings of IEEE INFOCOM 2003, San Francisco, CA, April 2003, pp. 406-416.

[102] M. Zorzi, R. Rao, L. Milstein, ARQ error control for fading mobile radio channels, IEEE Transactions on Vehicular Technology 46 (2) (1997) 445-455.

[103] M. Zorzi, R. Rao, Throughput analysis of Go-Back-N ARQ in Markov channels with unreliable feedback, in: Proceedings of IEEE ICC 1995, Seattle, WA, June 1995, pp. 1232-1237.

[104] M. Krunz, J. Kim, Fluid analysis of delay and packet discard performance for QoS support in wireless networks, IEEE Journal on Selected Areas in Communications 19 (2) (2001) 384-395.

[105] A. Fantacci, Queuing analysis of the selective repeat automatic repeat request protocol for wireless packet networks, IEEE Transactions on Vehicular Technology 45 (2) (1996) 258-264.

[106] W. Stallings, Data and Computer Communications, 8th Edition. Prentice Hall, 2007.

[107] J. Aikat, J. Kaur, F. Smith, K. Jeffay, Variability in TCP round-trip times, in: Proceedings of ACM SIGCOMM IMC 2003, Miami Beach, FL, October 2003, pp. 279-284.

[108] S. Shakkottai, R. Srikant, N. Brownlee, A. Broido, k. claffy, The RTT distribution of TCP flows in the Internet and its impact on TCP-based flow control, in: CAIDA, Technical Report TR-2004-02, February 2004.

[109] T. Klein, K. Leung, R. Parkinson, L. Samuel, Avoiding spurious TCP timeouts in wireless networks by delay injection, in: Proceedings of IEEE GLOBECOM 2004, Dallas, TX, December 2004, pp. 2754-2759.

[110] R. Ludwig, A. Gurtov, The Eifel Response Algorithm for TCP, RFC 4015, IETF, February 2005.

[111] J. Blanton, E. Blanton, M. Allman, Using spurious retransmissions to adapt the retransmission timeout, in: ICSI, Technical Report TR-08-005, August 2008.

[112] M. Welzl, Using the ECN nonce to detect spurious loss events in TCP, in: Proceedings of IEEE GLOBECOM 2008, New Orleans, LA, November 2008, pp. 2525-2530.

[113] xjperf: Graphical frontend for IPERF written in Java, http://code.google.com/p/xjperf/

[114] Y. Guan, B. Broeck, J. Potemans, J. Theunis, D. Li, E. Lil, A. Capelle, Simulation study of TCP Eifel algorithms, in: Proceedings of OPNETWORK 2005, Washington, DC, August 2005.

[115] A. Wierman, T. Osogami, J. Olsén, A unified framework for modeling TCP-Vegas, TCP-SACK, and TCP-Reno, in: Proceedings of IEEE/ACM MASCOTS 2003, Orlando, FL, October 2003, pp. 269-278.

PUBLICATIONS

Publication P1

R. Dunaytsev, Y. Koucheryavy, J. Harju, The impact of RTT and delayed ACK timeout ratio on the initial slow start phase, in: Proceedings of IPS-MoMe 2005, Warsaw, Poland, March 2005, pp. 171-176.

Publication P2

R. Dunaytsev, Y. Koucheryavy, J. Harju, The PFTK-model revised, Computer Communications 29 (13-14) (2006) 2671-2679.

Publication P3

R. Dunaytsev, Y. Koucheryavy, J. Harju, TCP NewReno throughput in the presence of correlated losses: the Slow-but-Steady variant, in: Proceedings of IEEE INFOCOM Global Internet Workshop 2006, Barcelona, Spain, April 2006, pp. 115-120.

Publication P4

R. Dunaytsev, K. Avrachenkov, Y. Koucheryavy, J. Harju, An analytical comparison of the Slow-but-Steady and Impatient variants of TCP NewReno, in: Proceedings of WWIC 2007, Coimbra, Portugal, May 2007, pp. 30-42.

Publication P5

R. Dunaytsev, D. Moltchanov, Y. Koucheryavy, J. Harju, Modeling TCP SACK performance over wireless channels with completely reliable ARQ/FEC, *Submitted for publication in International Journal of Communication Systems*.

Publication P6

D. Moltchanov, R. Dunaytsev, Modeling TCP SACK performance over wireless channels with semi-reliable ARQ/FEC, *Accepted for publication in Wireless Networks (in press)*.