VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Arūnas STATKUS

PERFORMANCE INVESTIGATION OF DATA PACKETS TRANSPORT CONTROL PROTOCOL IN HETEROGENEOUS NETWORKS

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES,
ELECTRICAL AND ELECTRONIC ENGINEERING (01T)

Vilnius LEIDYKLA TECHNIKA 2017

Doctoral dissertation was prepared at Vilnius Gediminas Technical University in 2010–2017.

The dissertation will be defended at the public meeting of the Dissertation Defense Council of Electrical and Electronic Engineering in the Senate Hall of Vilnius Gediminas Technical University at **10 a. m. on 29 August 2017**.

Address: Saulėtekio al. 11, LT-10223 Vilnius, Lithuania.
Tel.: +370 5 274 4956; fax +370 5 270 0112; e-mail: doktor@vgtu.lt

A notification on the intended defending of the dissertation was send on 18 July 2017.
A copy of the doctoral dissertation is available for review at VGTU repository http://dspace.vgtu.lt, at the Library of Vilnius Gediminas Technical University (Saulėtekio al. 14, LT-10223 Vilnius, Lithuania) and State research institute Center for Physical Sciences and Technology (Savanorių pr. 231, LT-02300 Vilnius, Lithuania).

VGTU leidyklos TECHNIKA 2017-054-M mokslo literatūros knyga

ISBN 978-609-476-046-4

arunas.statkus@vgtu.lt

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Arūnas STATKUS

DUOMENŲ PAKETŲ TRANSPORTO PROTOKOLO HETEROGENINIUOSE TINKLUOSE NAŠUMO TYRIMAS

DAKTARO DISERTACIJA

TECHNOLOGIJOS MOKSLAI,
ELEKTROS IR ELEKTRONIKOS INŽINERIJA (01T)

VGTU LEIDYKLA
Vilnius TECHNIKA 2017

VGTU leidyklos TECHNIKA 2017-054-M mokslo literatūros knyga.

Abstract

Nowadays with growth of the Internet traffic, it is essential to control channel congestion efficiently for successful utilization of network resources and network equipment. Today's main protocol for this objective is Transport Control Protocol (TCP). It is one of the main data interchange protocols on the Internet, based on which most of the other, higher-level protocols are running. This is a reliable data exchange protocol guaranteeing reliable data transfer between two remote network points with data flow control methods.

The aim of this dissertation is to investigate the TCP protocol behaviour in heterogeneous networks in different network conditions and how network efficiency can be increased via TCP optimization. TCP protocol modifications to improve the performance in high speed links with dynamically adapting acknowledgment function which could lead to significant decrease load of network equipment and increase speed of shared networks links are proposed in the dissertation.

The first chapter gives a short overview of TCP protocol, how it works and that are the main TCP algorithms that are used today in modern operating systems. Also a new TCP overhead reduction method was propose for improving network throughput for TCP in IEEE802.11 and Ethernet networks.

In the second chapter of the work an investigation of TCP in Linux OS kernel is made. A short overview of basic functions and algorithm is made to understand how TCP works in Linux OS, also new modification and implementation in Linux kernel code are proposed and investigated to achieve better efficiency in heterogeneous networks.

The third chapter presents the results of testing made on new modified Linux TCP kernel under different data and network conditions.

The dissertation suggests that in some conditions the network load can be reduced or throughput can be increased due to reduced TCP protocol overhead this not only allows to reduce the load to network equipment, but also allows to reduce load to TCP client and server which are generating and receiving data over TCP protocol. This TCP protocol overhead can have a big impact to TCP throughput in high speed networks or reduce the load on network nodes and servers.

The main results of the dissertation were published in 4 scientific publications, all of them were printed in peer-reviewed scientific journals. The results were presented in 4 scientific conferences.

Reziumė

Šiandien vis sparčiau augant Interneto srautams, itin svarbu valdyti duomenų perdavimo kanalo pralaidą siekiant kuo efektyviau išnaudoti duomenų perdavimo kanalus bei ryšio tinklo įrenginius. Šiuo metu pagrindinis duomenų apsikeitimo protokolas yra transporto valdymo protokolas (angl. *Transport Control Protocol – TCP*). Tai vienas iš pagrindinių duomenų apsikeitimo protokolų Internete, kurio pagrindu veikia didžioji dalis kitų, aukštesnio lygio, protokolų. Tai patikimas duomenų apsikeitimo protokolas garantuojantis patikimą duomenų perdavimą tarp dviejų nutolusių tinklo taškų su duomenų srauto kontrolės valdymo metodais.

Disertacijos darbo tikslas yra ištirti TCP protokolo veikimą nevienalyčių duomenų perdavimo tinkluose su skirtinga tinklo įranga ir TCP konfigūracija, norint pasiekti geresnį TCP efektyvumą bei sumažinti perteklinį duomenų srautą ir elektroninės tinklo įrangos apkrovimą.

Pirmame skyriuje pateikta trumpa TCP apžvalga bei istorija, pagrindiniai TCP protokolo algoritmai, naudojami moderniose operacinėse sistemose. Taip pat pasiūlytas naujas TCP perteklumo mažinimo metodas, siekiant pagerinti doumenų pralaidumą IEEE802.11 ir Ethernet tinkluose.

Antrajame skyriuje nagrinėjamas TCP protokolo veikimas ir programinio kodo modifikavimas Linux operacinėje sistemoje (angl. *Operating System – OS*) branduolyje. Trumpai apžvelgiami pagrindiniai algoritmai bei funkcijos, pateikiamos Linux branduolio kodo modifikacijos bei esamų TCP programinių algoritmų patobulinimai, norint pasiekti didesnį TCP efektyvumą nevienalyčiuose duomenų perdavimo tinkluose.

Trečiajame skyriuje pateikti patobulinto Linux OS branduolio TCP veikimo testavimo rezultatai esant skirtingiems duomenų tinklo parametrams ir sąlygoms.

Disertacijoje nustatyta, kad TCP protokolo veikimo efektyvumas, greitaveika bei esamų tinklo bei serverių resursų apkrova gali būti sumažinta naudojant adaptyvų patvirtinimo (angl. *Acknowledgment* – ACK) filtravimo algoritmą. Jo dėka galima pasiekti ryškų duomenų perdavimo greitaveikos prieauglį didelės spartos tinkluose, ar sumažinti tinklo įrenginių bei serverių apkrovą.

Pagrindiniai disertacijos rezultatai paskelbti 4 recenzuojamuose mokslo žurnaluose. Rezultatai viešinti 4 mokslinėse konferencijose.

Notations

Symbols

α_{TCP} – TCP RTT calculation variable;
N_{ACK} – number of ACK messages;
β_{cubic} – a constant multiplication decrease factor applied for $W_{\mathrm{c}}(t)$;
β_{TCP} – a TCP RTT calculation variable;
$F_{\mathrm{FlighSize}}$ – number of bytes sent by TCP sender but not yet acknowledged by the receiver;
G_{TCP} – wireless throughput of TCP data;
IW – initial congestion window in bytes;
$N_{\mathrm{DBP}S}$ – number of data bits per OFDM symbol;
S_{MSS} – TCP advertised maximum segment size;
$T1$ – time needed for sender from Slow Start to reach slow start threshold value;
t_{ACK} – RTT time for ACK message;
$\mathrm{t_{data}}$– RTT time for TCP data message;
$T2$ – time needed to reach from Slow Start to RWIN value;
Tb – random back of period of time for data transmission;
T_{CPU} – time needed for CPU process TCP packet;
T_{DCF} – time needed to make full frame exchange sequence;
$T_{\mathrm{DCF\_ACK}}$ – time needed to send one ACK frame;
$T_{\mathrm{DCF\_DATA}}$ – time needed to send one TCP data segment;
T_{DCF} – time needed to make full frame exchange sequence;

T_{DIFS} – DCF Interframe spacing time;
$T_{\max}$ – time needed to reach maximum link throughput;
T_{OFDM} – transmission time of one OFDM symbol;
T_{RTO} – TCP retransmission time value according RFC;
t_{RTO} – TCP retransmission time value;
T_{RTT} – TCP round tripe time with system processing time;
t_{RTT} – round tripe time of average successful TCP transmission;
T_{slot} – slot time which is physical characteristic of 802.11 standard;
$\mathrm{T}_{\mathrm{SIFS}}$ – shortest interframe spacing time;
t_{RTT} – is the average time of successful TCP message transmission;
ΔCWND – congestion window change value;
R – the first *RTT* value in new TCP session;
SMSS – sender's maximum segment size;
SRTT – smoothed round trip time;
SSTRESH – slow start threshold;
W_{CWND} – TCP congestion size window in bytes;
W_{RWND} – TCP receive window size in bytes;
σ_{RTT} – the root mean square of deviation of t_{RTT}.

Abbreviations

ACK – TCP acknowledgement;
BDP – bandwidth delay product;
BW – bandwidth;
CBS – committed burst size;
CIR – committed information rate;
CPE – customer premises equipment;
CPU – central processing unit;
CRC – cyclic redundancy check;
CSMA/CA – carrier sense multiple access with collision avoidance;
CW – contention window;
CWDN – TCP congestion window;
DCF – distributed coordination function;
DIFS – distribution inter frame space time;
FTP – file transport protocol;
GRO – generic receiver offload;
GSO – generic segmentation offload;
HGW – home gateway;
IAD – integrated access devices;
IP – Internet protocol;

I/O – input output device;
KVM – Linux kernel-based virtual machine;
MSS – maximum segment size;
MIPS – microprocessor without interlocked pipeline stages;
NIC – network interface card;
NDP – network delay product;
OS – operating system;
PCF – point coordination function;
PPS – packer per second;
RAM – random access memory;
RTO – retransmission time-out;
RTT – round trip time;
RWND – TCP receive window;
SIFS – short inter frame space;
TCP – transport control protocol;
TSO – TCP segmentation offload;
UDP – user data protocol;
WIN – TCP window size;
WWW – World Wide Web.

Contents

[1] The annexes are supplied in the enclosed compact disc.

Introduction

Problem Formulation

The global interconnecting network or the modern Internet as it is known now emerged more than 25 years ago with the birth of Transport Control Protocol (TCP) and Internet protocol (IP) suite (TCP/IP). Now it connects more than billions of different electronic devices worldwide and provides their data routing and exchange channels. In the early days of Internet, it was used for basic communication (WWW, email and file exchange), but only after one decade later things started to change rapidly, the Internet evolved to multidimensional service field with network expanding to all directions. From mobile phone to TV, game stations and even government management and election moved to global network.(Wellman *et al.* 2008).

From the first days of the TCP protocol the main and most important problems were and still are the TCP efficiency and performance (Nagle 1984: 160; Paxson *et al.* 1999; Garsva *et al.* 2014), to make this protocol link aware and allow utilize the network capacity as much as possible and as fast as possible, independent from changing condition of network or transport layer. To do this TCP or to be more correct the Linux OS kernel uses slow start, congestion avoidance, fast retransmit and other algorithms, which are responsible for data transmission and

Congestion Window Control (CWND). By controlling CWND the TCP stack indirectly controls the data sending rate, by slowly probing the link speed and allowing to increase or decrease sending data rate (Jacobson 1988). As the CWND size is controlled by TCP acknowledgment rate from the TCP data receiver side. In most cases lower network layers are not controlling data sending rate, and the information leaving the system Network Interface Card (NIC) at line rate of the link or maximum speed it is able to do it. In some cases it can lead to network overload and congestions.

In addition TCP also must handle system resources efficiently, as the same protocol implementation must work efficiently and with out of problems on different type of systems and devices (Priescu *et al.* 2012; Eidukas *et al.* 2005; Kajackas *et al.* 2015). As most modern systems like supercomputers or servers equipped with high performance CPU and RAM can handle intense information rates. But for TCP data transmission the same TCP implementation is used like in system with small amount of Random Access Memory (RAM) or low efficiency performance CPU. In such conditions most important is constant and stable data flow due to limited or very low network link through which they communicate.

Despite the fact that modern electronic and Central Processing Unit (CPU) units have high power efficiency, and some basic networking jobs can be done in NIC, the load generated from TCP/IP stack can take up to 4–13% of data processing load (Bencivenni *et al.* 2009). In normal conditions TCP code execution can take about ~200 instructions for processing one TCP packet, depending from CPU architecture and model. Therefore in the following, the problem of TCP implementation efficiency is addressed in the dissertation.

In order to solve this problem such main hypothesis was raised and proven: the reduction of TCP acknowledgment overhead can reduce load on network and electronic systems and increase their TCP data throughput without negative impact to TCP session stability and TCP recovery mechanism.

Relevance of the Thesis

The growing demand of Internet and its services increases the need of better utilizations of existing services and transport links. It is extremely important that current transport control protocol (TCP) should not be the drawback of the evolution of Internet but also offer new possibilities for developing new services and network applications.

Increased efficiency of TCP not only reduces the load to electronic systems and network but allows other applications and system developers to use freed resource for other purpose.

Present research is dedicated to investigate the possibilities and methods to reduce TCP overhead and reduce load not network and its equipment.

The Object of Research

The object of presented research is current implementation of TCP and acknowledgment algorithms that is used in operating systems (OS) of modern electronic systems and is used in heterogeneous data transmission networks.

The Aim of the Thesis

The goal of the dissertation is to increase TCP performance in heterogeneous networks and electronic network equipment, through reduction of TCP acknowledgment overhead in TCP stack and network equipment.

The Tasks of the Thesis

In order to solve the problem and achieve the aim of the dissertation the following tasks must be accomplished:

1. Analyse efficiency and operation of TCP in operating system and its data transmission over heterogeneous networks or asymmetric data links.
2. Analyse the limits and methods of TCP Acknowledgment (ACK) filtering in electronic network equipment and develop methods for ACK limiting in Linux kernel.
3. Experimentally investigate the impact of developed ACK limiting method for TCP performance in heterogeneous networks.

Research Methodology

The following research methods are applied in this work: literature analysis, security risk analysis, artificial computer networks with emulated delay and throughput characteristics, code and algorithm optimization and statistical analysis, statistical results validations, performance analyses and methods of experimental research have been used to validate proposed methods.

Scientific Novelty of the Thesis

In the thesis the following scientific novelty were achieved:

1. Proposed the new dynamical ACK limiting implementation in Linux OS kernel TCP stack, which allows efficient reduction and control of TCP protocol overhead.
2. Proposed the new algorithm for calculation of the initial and maximum values for ACK limiting algorithm in heterogeneous networks.
3. Developed the algorithm for ACK filtering in network and IEEE 802.11 wireless equipment which adapts to various network configurations

Practical Value of the Research Findings

In dissertation two different algorithms were created for TCP overhead reduction. The first algorithm implements ACK filtering on network equipment. It allows to reduce the TCP overhead without any end system knowledge or system modification. The network load from ACK messages can be reduced up to 80% and the performance of network routers can be increased up to 32%.

The second algorithm implemented as Linux kernel TCP stack modification. It allows not only to reduce the TCP overhead and reduce network utilization but also to reduce system load and dynamically improves TCP performance on the TCP client and server side dynamically.

Both proposed TCP overhead limiting algorithms can be used in IP based networks to reduce network load and increase asymmetric link throughput. By using Linux OS based ACK limiting algorithm end system load can be reduced up to 50% and increase end system performance.

The Defended Statements

1. ACK filtering on network equipment can reduces the ACK message overhead up to 80% without impacting existing TCP session stability. This optimization can reduce network load and increase network router performance up to 32% due to reduced load of ACK messages.
2. The ACK filtering on network equipment and ACK limiting in Linux OS kernel does not impact TCP session performance and such TCP sessions can successfully recovers from packet drops if upper limits of ACK filtering and limiting is not reached.

3. The dynamic ACK limiting in Linux kernel can reduces CPU load to network equipment, and can increase TCP throughput up to 50% in embedded devices.

Approval of the Research Findings

The main results of the dissertation were published in 4 scientific papers: 2 paper in foreign journal indexed in Thomson ISI Web of Science 2 papers in local journal indexed in Thomson ISI Web of Science.

The main results and ideas of work were presented in following scientific conferences:

1. Evaluation of TCP Acknowledgment Mechanism Influence on Router Performance. 14 ELECTRONICS'2010 , 2010, Lithuania, Vilnius.
2. Vilniaus miesto bevielių tinklų statistiniai tyrimai. Jaunųjų mokslininkų konferencija „Elektronika ir elektrotechnika 2011“, 2011, Lithuania, Vilnius.
3. Analysis of Home WiFi Internet Access Networks Situation in Vilnius City. 15 tarptautinė konferencija ELECTRONICS 2011, 2011, Lithuania, Vilnius.
4. Analysis of Home WiFi Access Networks Situation in City Area. ICCCISE 2013: International Conference on Computer, Communication and Information Sciences and Engineering, 2013, Spain, Barcelona.

Structure of the Dissertation

A dissertation consists of three main chapters and general conclusions with summary in Lithuanian. The dissertation start from introduction followed by first chapter in which TCP protocol, main functions and working algorithm are explained. Also an investigation of heterogeneous networks is described and how ACK overhead can be reduced in network layers is described. In the second chapter of dissertation Linux kernel and TCP limiting kernel is described and how can it reduce not only ACK overhead to network is described. In the third and final chapter experimental testing of new Linux kernel ACK limiting algorithm is made and results provided.

The volume of dissertation is 93 pages, in which are: 63 figures, 35 formulas, and 2 tables, also in dissertation are 91 references.

1

Transmission Control Protocol Overhead in Heterogeneous Networks

In this chapter a background and short history of TCP protocol is provided including a basic working functions and algorithm which are needed for guaranteed and stable data transmission from one end point to other and how TCP handles packet loss or session distortions. Also the issue of TCP acknowledgment overhead in network nodes will be discussed and what new implementation or modifications could reduce TCP overhead and improve the TCP efficiency and speed in high utilized or overloaded networks.

The research results are published in author publications: Pavilanskas *et al.* (2015); Statkus *et al*. (2013); Statkus *et al.* (2012). The main results are announced in scientific conferences: “Electronics” (Vilnius 2011), “Communication and Information” (Barcelona 2013).

1.1. Evolution of Transmission Control Protocol

The history of Transport Control Protocol or else known as Transport Control Program started in 1969 with building the United States Defence Advanced Research Projects Agency (DARPA or ARPA) research network else called the Arpanet. As the number of hosts in Arpanet network increased the developers realized that trying to use existing protocol in current infrastructure will lead to performance problems due to different technologies and different network properties. So the work on new lower layer protocol started in 1973 by Vin Cerf and Robert E. Kahn, who was hired by DARPA in 1972, where he worked in the packet communication satellite and radio waves. The new developed protocol had to meet the following requirements:

1. Independence from underlying network techniques and architecture.
2. Universal connectivity throughout the network.
3. Standardized application protocols.
4. Acknowledgments function.

The new layer protocol of TCP was first formally specified in December of 1974 in RFC 675 by Vinton Cerf, Yogen Dalal, Carl Sunshine. The development and testing of TCP continued for several years, and in early 1977 a new version of TCP came out – TCP v2. After one year in 1978 a new version of TCP v3 came out, with novel idea of splitting the TCP into TCP at the transport layer and IP at the network layer. The separation of to transport and transmission control protocol leads to creation of TCP/IP architecture and separation of protocol layers and new protocol creation. Finally in 1980 the fourth version TCP/IP came and the Internet was born, the fourth versions are still used up to now. In dissertation only TCP part will be discussed leaving the IP layer out.

1.2. Transmission Control Protocol Acknowledgement

The most important function of TCP is the guaranteed data transmission function. For successful and assured data transmission TCP is using acknowledgement algorithm, which "is at the heart of TCP" (RFC 813). It relies on demand for the receiver to communicate with the sender by sending back an ACK as it receives data. For this two main TCP protocol headers fields: sequence and acknowledgments fields are used, they are situated after source and destination fields of TCP header (Fig. 1.1) (Socolofsky *et al.* 1991).

Both Sequence Number and Acknowledgment Number fields are 32 bit long and mainly are responsible of data acknowledgment and informing about successful data delivery the TCP sender. As defined in TCP RFC documentation all the data bytes must be send in sequential way to TCP receiver. Based on successfully received data the TCP receiver acknowledges the received information by naming (sending back in ACK message) the highest number of successfully received data byte to the TCP data sender. This is called cumulative ACK technique and is described in RFC 813 document. The last ACK message (with highest acknowledgment number) received by the TCP sender indicates that all bytes of data with sequence numbers less than that value have been successfully received or acknowledged by TCP receiver. To be more correct the ACK sequence number identifies the first byte of data which has not been yet received by the client (ACK shows how much data the receiver has received plus one byte). The example of TCP acknowledgment algorithm is show in Fig.1.2.

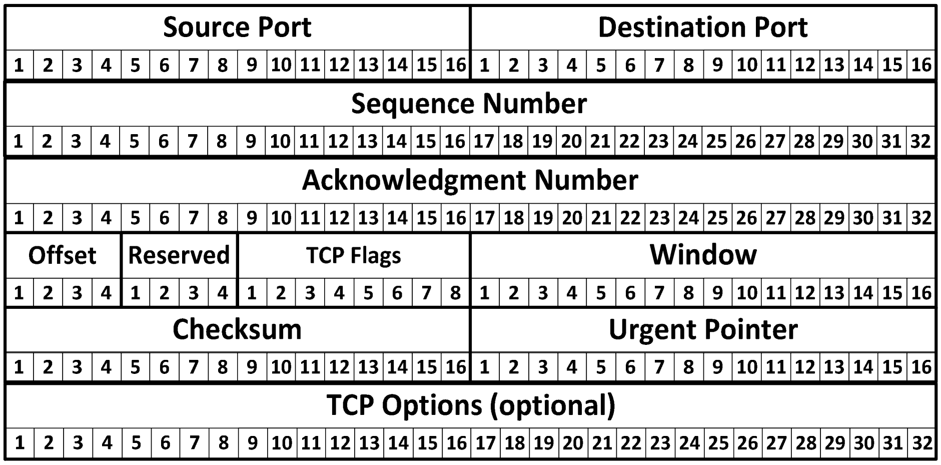


Fig. 1.1. Transmission control protocol header structure

Because every ACK packet not only generate network load but also increases processing time of end node, this is very important to systems with lower CPU power. The frequency of ACK sending must be controlled and meet the following conditions (Bott 2014; Stevens 1997; Berkeley *et al.* 2011):

1. After receiving other full segment size (Linux, UNIX) or two TCP packets (Windows), according RFC 5681, an ACK message should be generated for at least every second full size segment data packet.
2. After receiving a packet with push bit set.
3. On update of window size (retransmission timer must not be expired).

4. On expiration of time trigger.

The TCP contains only a general assertion that data should be acknowledged promptly, but gives no more specific indication as to how quickly and as how frequently an ACK must be sent. Normally TCP does not send ACK instantly but after receiving data but delays ACK messages sending (Bott 2014; Allman *et al.* 1999), hoping to send some data going in the same direction as ACK message. A typical TCP uses delay of 200 ms and sends ACK for every other data message. However, there is not much TCP traffic with bidirectional data flows in Internet (Socolofsky *et al.* 1991; Kajackas *et al.* 2011). Delayed ACK is usable for an interactive applications (Telnet, SSH, and etc.) only (Kim *et al.* 2008; Cáceres *et al.* 1991).

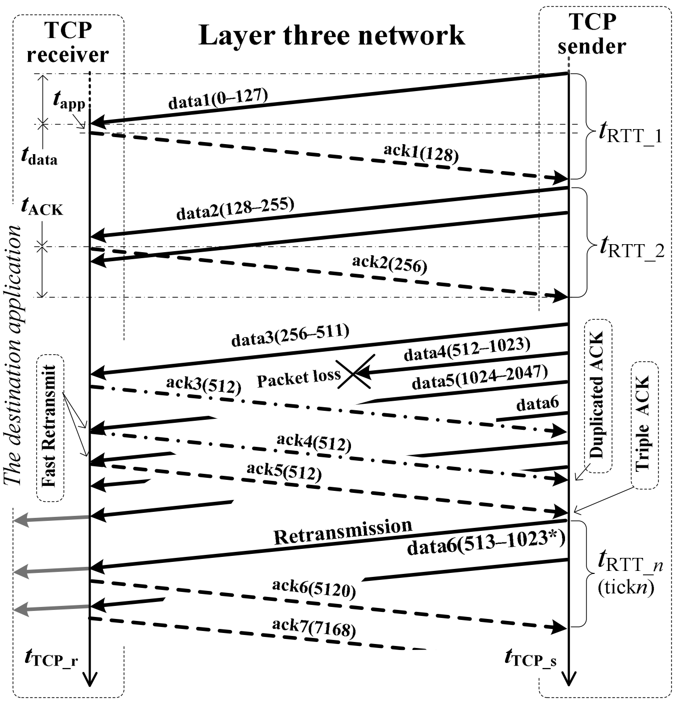


Fig. 1.2. Transmission control protocol flow of acknowledgment process

Beside the acknowledgment the ACK messages also returns the information of the buffer which is currently available at the receiver side, also known as window size (Fig. 1.1). It is responsible for data flow and control of it, also TCP window protects the receiving side from buffer overflow (Clark 1982). The TCP sender uses the offered window to calculate the usable window, this is done by calculating the difference of received window minus the data which is outstanding in the network and still are unacknowledged and (estimate how much data can be outstanding in the network). Upon the size of usable window the TCP sender uses a congestion window, which growth function depends on the feedback sender gets from the network through the received ACK. After initializing the TCP sender is allowed to increase the congestion window (CWND) for each incoming ACK.

1.3. Transmission Control Protocol Flow Control and Congestion Window

To allow TCP protocol dynamical consume network bandwidth a congestion control functions and algorithms are employed. It consists of four critical congestion control algorithm: slow start, congestion avoidance, fast retransmit and fast recovery. Each of the algorithms is applied in different state TCP to control the data flow of TCP data. The current congestion control algorithm implementation is defined in the latest RFC 5681 document (it absolutes RFC 2581 and RFC 2001, RFC 813).

1.3.1. Transmission Windows

Before grasp slow start algorithm, it is necessary to understand how TCP places limits on the amount of data, which can be in transit between two endpoints at a given time. Because of the reliable nature of TCP, a TCP sender can transmit only a limited amount of data before it must receive an acknowledgement from the TCP receiver. This is done to ensure that all send data during time frame was received successfully and that any lost segments can be retransmitted efficiently (Jacobson 1988; Griner *et al.* 2000).

There are two main variables (windows) which affect how much unacknowledged data a sender can send. The first windows or receiver window – *RWND*, it is always advertised by the TCP peer to other end in TCP header at the start of TCP session or updated during data transmission. The second window is called congestion window CWND and is dynamically calculated on TCP data sender side (Handley *et al.* 2000; Allman *et al.* 1999). The sender's congestion window, however, is known only to the sender and does not appear on the wire. According the RFC documentation the CWND window must be lower than *RWND* value, and it is the maximum amount of unacknowledged data the sender can transmit before receiving ACK message.

At the start of TCP session the CWND initial windows or *IW* must be set according the RFC 5681 the initial window (*IW)* size must be set in accordance with the following conditions shown in flow graph (Fig. 1.3).

The *IW* calculates based on sender's maximum segment size or *SMSS* value. It is received in TCP header option field (Fig. 1.1) as maximum segment size (MSS) sub option value, from other side during TCP connection setup in the TCP Synchronize (TCP SYN) packet. In example a MSS value 1460 B according to RFC 5681 algorithm (Fig. 1.3), would give *IW* of 4380 B (3 × 1460 = 4380 B). However, in practice the *IW* or initial CWND size will vary among TCP/IP stacks and OS implementations.

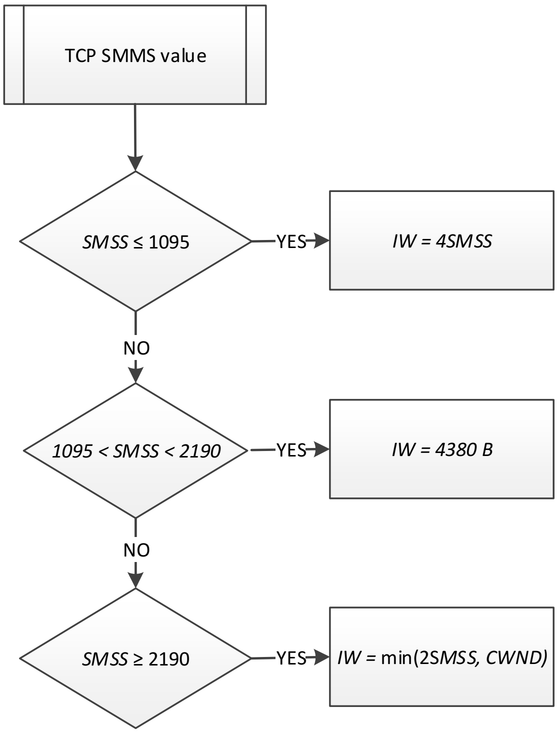


Fig. 1.3. Transmission control protocol initial window selection based on sender's maximum segment size value algorithm

It is important to note that the sender's effective transmission window is always lower than CWND and RWND and by using slow start, congestion avoidance algorithm the TCP sender dynamically adjusts data sending speed during the TCP session time (Jacobson 1988; Allman *et al.* 1999).

1.3.2. Slow Start Algorithm

Every new TCP session, not depending from the last status (if it's new/reconnected) starts from slow start algorithm. This is done due to unsureness of data transmission link and network conditions, because a big burst of data injected into network could cause packet loss in network due equipment buffer congestion or network overload (Cáceres *et al.* 1991; Floyd 2003; Podolsky *et al.*).

Due to network link diversity and fragmentation most of the networks nodes must queue received data packets for further processing. In most cases the network nodes must accumulate several network links which eventually lead to network overload and congestion. To avoid this and not allow initial network queues overrun TCP is using slow start algorithm. This algorithm operates on TCP sender side and is based on received ACK message rate or "self-clocking", as described by V. Jacobson in 1988. This algorithm allows slowly (in the start) increase TCP sending rate, without instantly overrunning the network links or routers queues with TCP data messages (eventually, after some time the network congestion mostly will happen).

The TCP slow start algorithm is used in three states of TCP session:

1. After starting a new TCP session (after TCP three way hand shake).
2. After restarting the transmission after long idle period.
3. After retransmission timeout.

Every time a slow start algorithm is activated two variables must be set: CWND, and slow start threshold (*SSTHREH*) values (Seth *et al.* 2008; Al-Khatib *et al.* 2006). The first variable defines the amount of data in bytes which can be sent without receiving ACK from the receiver. It must not be bigger the advertised window size or *RWND*. The second one defines the threshold of TCP slow start, used to define the upper limit of slow start algorithm and must be used until CWND > *SSTHRESH*) After exceeding this value the congestion avoidance algorithm is used to increase CWND and control TCP data transmission (RFC 5681). The initial size of *SSTHRESH* should not be higher than advertised RWIN, but it's not fully specified in the RFC what is the maximum and minimum size of it. According the old RFC 2001 the initial value of *SSTHRESH* for new session should be 65,535 B. In the initial state of new TCP session slow start *IW* of CWND must be set, according the RFC 5681seen in Fig. 1.3.

The *IW* selection is only use after three way hand shake and must be used if no packet loss where in *three-way-hand* shake phase (Bott 2014). If packet loss during *three-way-hand* occurs or SYN, SYN/ACK messages do not arrive upon a TCP timeout. The CWND must set to one *SMSS* bytes at most (RFC 5681) (Loss Window). By using bigger initial window for CWND the transmission time can be noticeably decreased, this is especially noticed in short time TCP sessions with less than 4 KB (Allman *et al.* 2002; Fox 1989), but this could also can lead to network overload and congestions with small bandwidth capacity or with multisession applications.

The slow start algorithm is explained in Fig 1.4, the *IW* in our example is set to one *MMS* = 1000 B for simplicity.

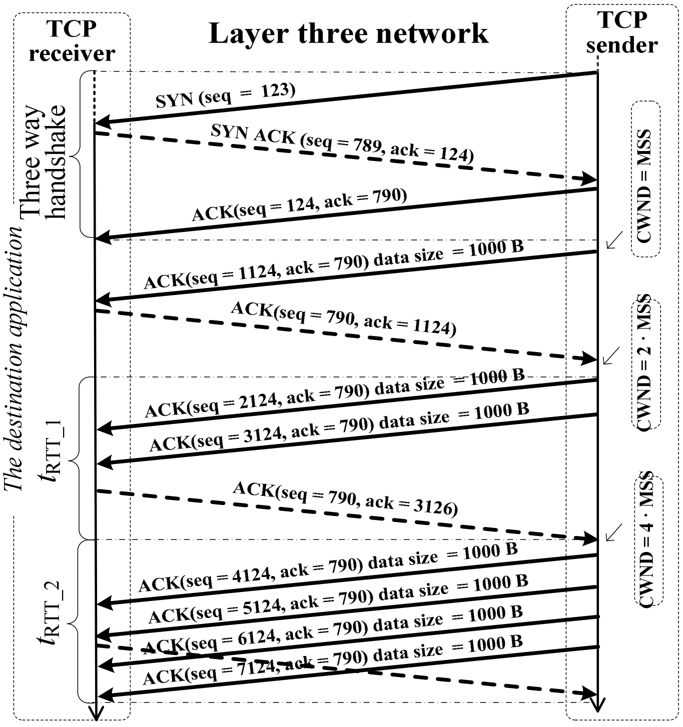

TCP receiver
Layer three network
TCP sender
Three way handshake
The destination application
t_{RTT_1}
t_{RTT_2}
SYN (seq = 123)
SYN ACK (seq = 789, ack = 124)
ACK(seq = 124, ack = 790)
ACK(seq = 1124, ack = 790) data size = 1000 B
ACK(seq = 790, ack = 1124)
ACK(seq = 2124, ack = 790) data size = 1000 B
ACK(seq = 3124, ack = 790) data size = 1000 B
ACK(seq = 790, ack = 3126)
ACK(seq = 4124, ack = 790) data size = 1000 B
ACK(seq = 5124, ack = 790) data size = 1000 B
ACK(seq = 6124, ack = 790) data size = 1000 B
ACK(seq = 7124, ack = 790) data size = 1000 B
CWND = MSS
CWND = 2 · MSS
CWND = 4 · MSS

Fig. 1.4. Transmission control protocol slow star*t* algorithm at session start

The slow start algorithm in Fig. 1.4 takes place only the *three-way-handshake* and initial window is set (in over case the *IW* is set to one data segment or MSS to 1000 B). After it the TCP data sender is allowed transmit one data segment (MSS = 1000 B) and must wait for ACK message which acknowledges received data for further data transmission. After receiving ACK message the congestion window (CWIN) can be increased two times to maximum segment size and two TCP data segments are allowed to transmit. If both data segments received one acknowledgment for both of them is generated, it acknowledges both data segments, (RFC 5861) and is send to the TCP data sender. After receiving it the TCP CWIN is increased up to 4 MSS and so on until the maximum allow window size (*SSTRESH*) is reached or packet loss occurs (Stevens 1997b). The congestion value is increased by maximum one *SMSS* every ACK (if ACM messages is generated for every data segment) or by CWND within Round Trip Time (*RTT)* time if ACK is generated for all data segment.

In practice the ACK is generated for every second full size TCP segment and this increase is no more aggressive than allowed in RFC documentation of TCP. Despite the fact that in practice most operating systems generate ACK for every second full size segment (RFC 5681) in some cases it could have a security drawback. This is known as ACK division attack, when ACK is generated more frequently. It leads to TCP CWND inflation and can reduce or reset parallel TCP sessions. This issue was discussed in detail and a solution was proposed in RFC 3465 document, which later was included in standard track of latest TCP congestion control RFC document (RFC 5681). The offered solution agents ACK division suggested of using ABC algorithm or appropriate byte count for CWND increase. It suggests that TCP window CWND must be increased not be bigger when the number of previously unacknowledged bytes of last received ACK:

$$\Delta W_{\mathrm{CWND}} = \min(N, SMSS)\,. \tag{1.1}$$

Where *N* is the number of previously unacknowledged bytes acknowledged in the incoming ACK, *SMSS* is the size of the largest segment that the sender can transmit. This not only increases the CWND more correctly but also protects agents ACK division attacks described in paper "TCP congestion control with a misbehaving receiver" by S.Savage *et al.* (1999).

The slow start and congestion window growth functions can be defined as exponential (due to variation of *RTT* and delayed ACK is not exact exponential) and the sending rate of TCP is increasing rapidly after three way hand shake is over. The time needed to reach TCP *SSTRESH* defined value mostly depends form *RTT* and link speed. In cases where *SSTRESH* value is set as big as *RWIN* the time needed to reach maximum link throughput or $T_{\max}$ time can by:

$$T_{\max} = T_{\mathrm{RTT}} \cdot \log_2 \frac{W_{\mathrm{WIN}}}{MSS^2}\,. \tag{1.2}$$

Where W_{WIN} is the TCP window size value needed to reach maximum link throughput, and can be equate to *network delay product (BDP)* of transmission link. Considering that the system processing time of TCP is much smaller than *RTT*. The *BDP* can be calculated by equations (Alrshah *et al.* 2014):

$$W_{\mathrm{WIN}} = BDP = T_{\mathrm{RTT}} \cdot BW\,; \tag{1.3a}$$

$$T_{\max} \cong T_{\mathrm{RTT}} \cdot \log_2 \frac{T_{\mathrm{RTT}} \cdot BW}{MSS^2}\,. \tag{1.3b}$$

Based on (1.3b) the TCP throughput in slow start state is mostly impacted from *RTT* time and can dramatically reduce the TCP throughput increase on high speed links with big delay or short TCP sessions then receiver employs delayed ACK (Allman *et al.* 2002). The results of how *RTT* impacts TCP throughput are presented in Table 1.1.

Table 1.1. Transmission control protocol start time dependence from round trip time value

Bandwidth	Time with RTT = 5 ms	Time with RTT = 10 ms	Time with RTT = 20 ms
1 Mbit/s	61.4 ms	132.9 ms	285.8 ms
10 Mbit/s	78 ms	166 ms	352.2 ms
100 Mbit/s	94.7 ms	199.3 ms	418.6 ms
1000 Mbit/s	111.3 ms	232.5 ms	485 ms
1 Gbit/s	128.9 ms	265.8 ms	551.5 ms

A noticeable impact is seen in short time (small data size transfer) TCP sessions, when all data exchange takes places in slow start. Basically most of websites are impacted by this issue and can dramatically increase the page loading time with big *RTT* even having high bandwidth throughput links (Shepard *et al.*; Pranevičius *et al.* 2006). To reduce the impact of TCP slow speed in Slow Start phase, several proposal were suggested (Allman *et al.* 2002) to increasing the initial upper bound from one segment RFC 2001 to up to 4 KB, which was left for further testing and discussion in RFC 2581 documentation.

Later more discussion by M. Allman was made, who suggested increasing the initial slow start congestion window (CWIN) from two/four segments to 10 or more (Berkeley *et al.* 2011). This would help to overcome short TCP session problem, the time $T_{\max}$ needed to reach maximum link throughput would be:

$$T_{\max} = T_{\mathrm{RTT}} \cdot \log_2\left(\frac{BW \cdot T_{\mathrm{RTT}}}{MSS}\right) - T_{\mathrm{RTT}} \cdot \log_2(10 \cdot MSS) = \\ = T_{\mathrm{RTT}} \cdot \log_2\left(\frac{BW \cdot T_{\mathrm{RTT}}}{10 \cdot MSS}\right). \qquad (1.4)$$

Also additional solution was proposed, which focused on reducing the total time of all session. So basically the three way handshake algorithm was modified to allow to send or request need data in SYN packet and replay needed data in SYN ACK packet. This allows noticeably reduce session time in small size TCP exchange sessions and speed up sequential communication between TCP client and server.

In addition to increased *IW* value, the latest RFC also recommends of using appropriate byte count (Allman 2003), which should increase the security of TCP and now allow ACK division attacks.

1.4. Congestion Avoidance Algorithm

One of most important TCP function is dynamical transmission rate control and adaptation to changing network conditions. It is controlled by *congestion control* algorithm which is responsible for CWND window growth after the slow start is over or then CWND reaches the SSTRESH value. Like in TCP the slow start the congestion avoidance algorithm tries to increase the CWND but this is doing it more slowly and keep the network in optional state of operation. This is done by increasing the CWND by a small amount of data for each received non duplicated ACK packet (RFC 5681).

The increase of CWND should not be bigger than one *SMSS* per RTT, and usually is describe as following equation (Bott 2014):

$$W_{\mathrm{CWND}} = W_{\mathrm{CWND}} + \frac{SMSS \cdot SMSS}{W_{\mathrm{CWND}}}. \tag{1.6}$$

This functions is executed every time an ACK message is received which acknowledges new data received from TCP sender. In case the ACK message is generated for every TCP data segment the increase of the CWND would be one *SMSS* per RTT, if ACK is generated for every second TCP data segment as defined in RFC 1122 the CWND increase will be no bigger than one *SMSS* per RTT. The increase fraction per RTT is define as following equation, in condition then ACK is generated for every second full size TCP data segment:

$$\Delta W_{\mathrm{CWND}} = \frac{SMSS^2}{W_{\mathrm{CWND}}} \cdot N_{\mathrm{ACK}}. \tag{1.7}$$

Here N_{ACK} is the number of ACK messages received per RTT time, and can be found as follow (Allman *et al.* 1999):

$$N_{\mathrm{ACK}} = \frac{W_{\mathrm{CWND}}}{2 \cdot SMMS}; \tag{1.8}$$

$$\Delta W_{\mathrm{CWND}} = \frac{SMSS}{2}. \tag{1.9}$$

The main problem of this function is that CNWD growth function is directly depending from ACK generation rate. With more frequent ACK generation rate the CWND increase can be bigger and unfairness can arise in concurrent TCP sessions, like it was said in paper of Stefan Savage *et al.* "TCP congestion Control with a Misbehaving Receiver" (1999). This not only has a negative impact to TCP session's fairness but can also be used as a security fault, also known as ACK division. It can lead to denial of service for other TCP clients or network link overrun.

The second issue with (1.6) is due the arithmetic counting, according, when CWND is bigger than *SMSS* × *SMSS,* the delta or increase of CWND will yield 0. To overcome this problem the returned value must be rounded to one.

The solution of these issues was defined and explained in RFC 3465, which suggested of using appropriate byte counting or ABC for CWND increase, the CWND should be increased not based on number of received ACK from the receiver but based on the number of acknowledged bytes. In this case the rate of received ACK has no impact to CWND increase and only the amount of bytes acknowledged by ACK is used as a factor for increase of CWND. This not only allows overcoming the security issued of ACK division but also can help in cases of lost ACKs and delayed ACK is used.

Despite the fact that ABC algorithm allows to solve ACK division issue and increase the CWND more equal in time it also requires new TCP state variables and checking with current CWND. In adds additional load to system and consumes CPU and RAM resources. Also the RFC 5681does not forbid of using old implementation of CWND increase in congestion avoidance phase, it should give as acceptable growth of congestion window, with limitation that the CWND increase must not be bigger than 1 *SMSS* per RTT.

If during the CWND increase a segment loss accurse, due to expiration of retransmission timer the *SSTHRESH* (slow start threshold) must reduce as follow (Berkeley *et al.* 2011):

$$SSTHRESH = \max\left(\frac{F_{FlighSize}}{2}, 2\cdot SMSS\right). \tag{1.10}$$

Here $F_{\text{FlighSize}}$ variable defines bytes that are sent but not yet acknowledged by the receiver, the bigger the delay the bigger $F_{\text{FlighSize}}$ will be.

Without mentioned and described *congestion control* algorithm in practice where more OS are based congestion control algorithm that includes various aspects of an additive increase/multiplicative decrease. In dissertation TCP Cubic congestion avoidance algorithm was used. TCP Cubic congestion window is determined by function(Ha *et al.* 2008):

$$K = \sqrt[3]{\frac{W_{\max}(t) \cdot \beta_{\text{cubic}}}{C}};$$
$$W_{\text{c}}(t) = C\left(t - K\right)^3 + W_{\max}(t), \tag{1.11}$$

here K is time period that takes to increase current congestion windows W_{c} to $W_{\max}$, C is a Cubic parameter (scaling factor), t [s] – the elapsed time from the last window reduction, $W_{\max}$ [bytes] – the maximal window size before the last reduction; β_{cubic} – a constant multiplication decrease factor applied for W_{c} reduction to minimal at the time of loss event (Ha *et al.* 2008).

In TCP data loss invent the new congestion window of Cubic would be:

$$W(t)_{\min} = W(t)_{\max} - \beta_{\text{cubic}} \cdot W(t)_{\max}. \tag{1.12}$$

Here $W_{\min}(t)$ is the new minimal congestion window, $W_{\max}(t)$ – current congestion window before loss.

1.5. Fast Retransmit and Fast Recovery Algorithm

The TCP data message retransmission can occur because of two main reasons (Keceli *et al.* 2007): after expiration of retransmission timer (RFC 813) and after receiving three or more duplicated ACK (Fig. 1.5: *ack3, ack4, ack5*). In first case the retransmission timer expires when no new data is acknowledged for a set of threshold time (RFC 2581).

The retransmission timeout (t_{RTO}) is taken as a loss indication, and it triggers retransmission of the unacknowledged segments. The threshold time during which the confirming ACK message must be received, t_{RTO} can be determined by the equation (Allman *et al.* 1999)

$$t_{\text{RTO}} = \overline{t_{\text{RTT}}} + 4 \cdot \sigma_{\text{RTT}}. \tag{1.13}$$

Here $\overline{t_{\text{RTT}}}$ is the average time of successful TCP message transmission, σ_{RTT} – the root mean square of deviation of $\overline{t_{\text{RTT}}}$.

If during t_{RTO} the ACK is not received, the data segments loss will be detected and the TCP sender will set $W_{\text{c}}(t)$ to one segment (RFC 2988); since t_{RTO} indicate that channel utilization has changed dramatically (Keceli *et al.* 2007).

Second reason occurs after receiving three or more duplicated ACK (Allman 2003; Sarolahti *et al.* 2003; Kojo *et al.* 2009). As shown in Fig. 1.4 the message *data4* was lost. The TCP receiver accepts messages *data3, data5, data6*, and so on but not *data4*. The *ack4, ack5* with same ACK number 512 were sent for received messages. After duplicated ACK is detected, the transmitter waits for t_{RTO} to expire. TCP does not yet know whether a duplicate ACK is caused by a loss or just reordering of segments. TCP sender waits the timeout value and assumes that if there is just a reordering it will not get ACK with number 512 anymore. However, after a while, a third duplicated ACK is received in a row (*ack5*). It is a strong indication that segments have been lost. TCP performs retransmission of segments 513–1023 with *data6*, without waiting for a t_{RTO} to expire. After this the sender maintains the number of outstanding segments by sending a new segment for each incoming ACK. It should be noted that only retransmission of identified as lost (timed out) TCP segments, are implemented in the conservative o passive TCP versions (RFC 3517). Meanwhile, in aggressive TCP implementation after the loss, all unacknowledged messages are retransmitted (Seth *et al.* 2008).

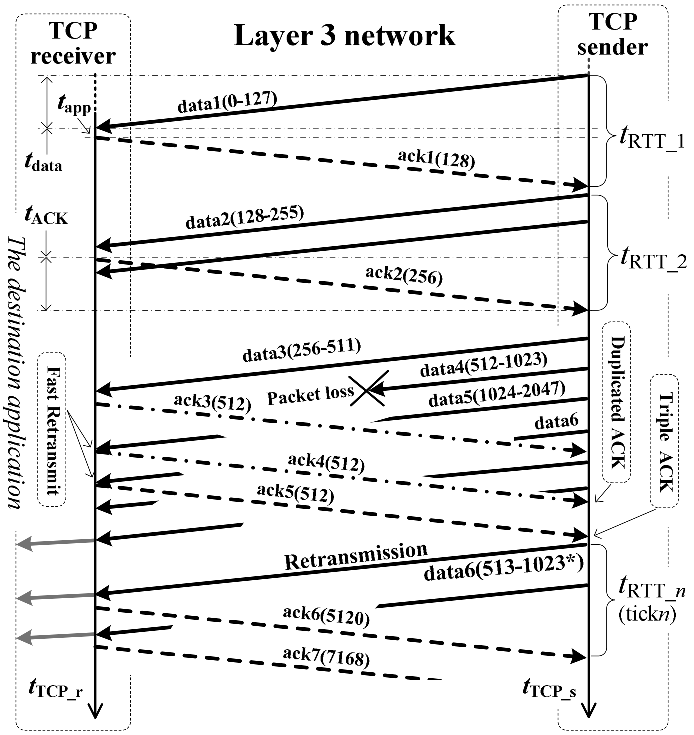


Fig. 1.5. Transmission control protocol fast retransmit and fast recovery after packet drop

The TCP contains only a general assertion that data should be acknowledged promptly, but gives no more specific indication as to how quickly and as how frequently an ACK must be sent. In RFC’s clearly is indicated, that current algorithm must maintain two very important functions: to prevent data retransmission, and as soon as possible to make ACK to permitting further data to be sent (Bott

2014). In addition to this argument, the fact that ACK message are very important in the initial phase (1) of data transmission and the fact that rate of ACK relay on data rate (segments loss due to receiving 3 or more duplicated ACK) and cannot be less than $1/t_{\mathrm{RTO}}$ must be evaluated.

1.6. Nagle's and Delayed Acknowledgment Algorithm

In the early days of TCP the small packet problem appeared when TCP had to deal with small data chunks send to TCP stack. This generates a big protocol overhead and could lead to congestion in heavily loaded networks. To solve this problem John Nagle introduced simple but effective algorithm to solve small packet problem (Nagle 1984). It's based on delaying of sending the data to remove node until one following requirements are met:

1. The buffered data reached MSS.
2. An ACK from remove node is received – the system state changes.

The Nagle's algorithm can protect the network from small packet problem by forbidding of sending small chunk of data before buffering.
In same time a different type algorithm on receiving side is used – *delayed ACK*. It implies that the receiver must not send an acknowledgment to remote side until following requirements are met (see Fig. 1.6) (Nagle 1984):

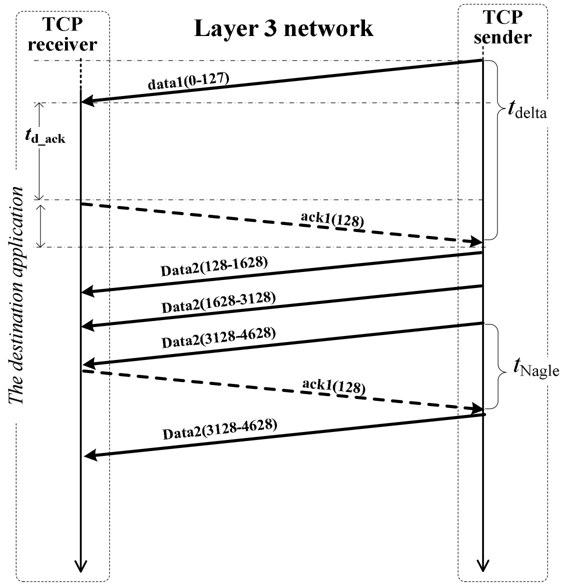


Fig. 1.6. Transmission control protocol Nagle's and delayed acknowledgment algorithm

1. The ACK messages should not be delayed more than for 500 ms (from 2.6 Linux kernel implementation the value is reduced to 200 ms).
2. A second full size segment is received (or two TCP packets in Windows OS).

Two algorithms working at same time can create a problem when TCP sender is not allowed to send more data until it has more data or an ACK is received from remote node. But in the remote node the *delayed ACK* algorithm postpone of sending ACK messages due to that only one TCP data segment is received and the *delayed ACK* algorithm timer is not expired and could lead of performance degradations where additional timer is needed to finish data exchange between two nodes (see Fig. 1.6) (Karn *et al.* 1987).

1.7. Retransmission Timer Calculation

As TCP segments can be lost or corrupted due to network errors, communication problems or OS issues, the TCP sender and receiver must track the information they have send or received. It is done via TCP sequence and acknowledgment numbers (see Fig. 1.1). This is information is send back to TCP sender or to receiver in TCP header. After receiving acknowledgment for the sent segment, a new portion of data can be transmitted. In cases a last ACK messages is lost, the TCP sender will wait this message and will not transmit any new TCP data messages. To solve this issue retransmission timeout (RTO) was introduced in RFC 793. As the network conditions can change over the time the retransmission timeout values must be dynamically adjusted for changing conditions (Karn *et al.* 1987; Ramakrishnan *et al.* 2001). As defined in the latest standard track RFC 6298 (in some implementation it can differ or use older RFC recommendations) the RTO value is computed from two state variables *RTTVAR* – round trip time variation and *SRTT* – smoothed round trip time (Berkeley *et al.* 2011). For this the RTT timer must be calculated first, it is done by measuring the time it took the sender to send and receive the ACK packet containing acknowledgment for the sent sequence number. Then RTT is calculated the *SRTT* value can be computed. Depending from the state of the session two different calculations of this variable can be made. If the TCP is in initialization state and no RTT is calculated, the RTO value is set to one second, in older RFC documentations or OS the suggested value is set to three seconds. In new TCP session, when no RTO value is compute yet. the RTO is calculated accordingly RFC (Berkeley *et al.* 2011):

$$SRTT = R; \tag{1.14}$$

$$RTTVAR = \frac{R}{2}; \tag{1.15}$$

$$T_{\mathrm{RTO}} = SRTT + \max\left(G, K \cdot RTTVAR\right). \tag{1.16}$$

Here T_{RTO} is new retransmission time out value, $K = 4$, G is the clock granularity in second and the R is set to round trip time value when the first measurement of RTT is made.

After subsequent RTT measurement are made and new R values are obtained, the *RTTVAR* and *SRTT* values must be updated as follows (Berkeley *et al.* 2011):

$$RTTVAR = (1 - \beta_{\mathrm{TCP}}) \cdot RTTVAR + \beta_{\mathrm{TCP}} \cdot | SRTT - R |. \tag{1.17}$$

$$SRTT = \left(1 - \alpha_{\mathrm{TCP}}\right) \cdot SRTT + \alpha_{\mathrm{TCP}} \cdot R. \tag{1.18}$$

The *RTTVAR* and *SRTT* values, according the RFC 6298, should be computed using $\alpha_{\mathrm{TCP}} = 0.125$ and $\beta_{\mathrm{TCP}} = 0.25$, but can changed on condition. The new value of T_{RTO} is calculated based on (1.16). If calculated RTO value is less than one second the result should be rounded to 1 s.

Despite the fact that the T_{RTO} value must be large enough to not allow retransmission of the segment to early, the big T_{RTO} value has significant impact to TCP performance (McKenzie 1989) after packet drops or idle periods especially in wireless link with big delay variations and smaller RTO would be more acceptable (Chen *et al.* 2011).

1.8. Heterogeneous Networks

From the birth of TCP protocol in early 1974, the first specification of TCP protocol, was made to work in heterogeneous networks and connections, over long distance links with high probability of packets loss and low throughput. But now with network throughput increasing the typical TCP protocol implementations, which is successful at low speed (up to 100 Mbps), is unfit for high speed networks (Berkeley *et al.* 2011; Allman *et al.* 1999). This not only reduce the efficiency of TCP protocol (does not allow for full link utilization in some cases) but also the growth of network bandwidth is directly coherent with ACK message rate on the following TCP session. When the TCP data rate increase, the ACK rate on the channel is increasing too. High speed network have another undesirable feature – growth of technological expenditures (Dalton *et al.* 2004; Maier *et al.* 2009). This protocol overhead not only reduces the network capacity but also increase the load to network devices and end nodes. In addition to TCP overhead other different network technologies and transport protocol add protocol overhead, which uses network capacity and system resource of network nodes. It is especially noticeable in wireless networks (IEEE 802.11a/b/g etc.) where same data channel is used for data transmission and receiving (Keceli *et al.* 2007;

Rindzevičius *et al.* 2015; Al-Khatib *et al.* 2006). In such heterogeneous networks TCP protocol overhead is mostly noticeable and has biggest impact to TCP response time and throughput.

Even nowadays basically most of current existing networks can be called heterogeneous, due to fact that are using direct interconnecting technologies. Only small part of data centers and NAS networks still can be call homogeneous networks, where networks servers and clients are not limited by network throughput or all nodes have same throughput with no multisession communication in place. By saying heterogeneous networks we mean not only different network technologies but also protocols and link utilization and delay. With more data streams send over one link or network node a fairness problems arise. In such case one or several data streams with smaller RTT or loss probability can consume most of the network traffic and leaving nothing to other streams (Floyd *et al.* 2000; Paxson *et al.* 1999). In addition to this the TCP protocol by nature is dynamical, and tries fully utilize network capacity. Basically there are two network streams which compete for network capacity you create heterogeneous network environment. Reduction of the TCP overhead not only reduce the load to network nodes and channels but also decrease electronic system utilization of network nodes (Paredes-Farrera *et al.* 2006). In some cases the reduction of TCP overhead can reduce the load off all network and even eliminate the CPU bottleneck in some network devices or end systems. It also reduces the level of uneven in data transmission networks (Rose *et al.* 1991).

1.9. Network Based Acknowledgment Filtering Technique

The concept of ACK filtering was discussed well in (Karn 2011). The idea of ACK message filtering is fairly simple, when router or other network devices needs to send the ACK message, it scans output queue for any earlier arrived ACK message. If a new ACK has higher *acknowledgment number* value, the older ACK message must be dropped. Since TCP acknowledgments are cumulative and the newest one obsoletes all older ACK messages. So, there's no need for more than one ACK message to be in output queue belonging to the same TCP session. Dropping old ACK when a new one is queued means there would never be more than one ACK on the queue at any time. It is the same as replacing the earlier ACK with the newer one.

The main concept of ACK message filtering is seen in Fig. 1.7. In network devices, which performs ACK message drop. This action does not affect the TCP session performance and stability and also the TCP sender does not even suspect that ACK message drop occurred. But as explained in (Karn 2011) where also main drawbacks of ACK filtering were reviewed.

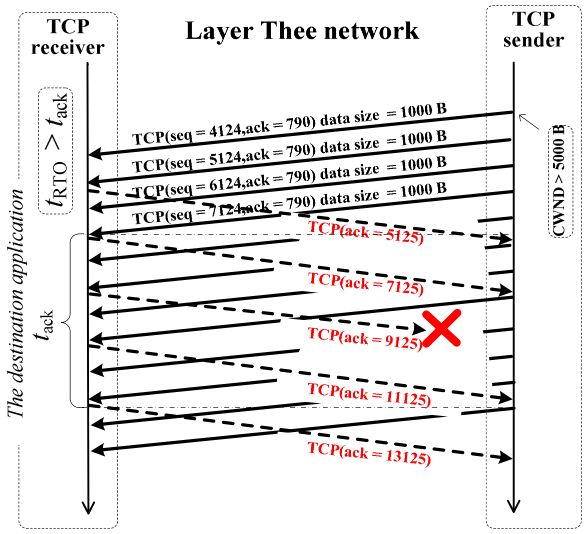

TCP receiver
Layer Thee network
TCP sender
TCP(seq = 4124,ack = 790) data size = 1000 B
TCP(seq = 5124,ack = 790) data size = 1000 B
TCP(seq = 6124,ack = 790) data size = 1000 B
TCP(seq = 7124,ack = 790) data size = 1000 B
TCP(ack = 5125)
TCP(ack = 7125)
TCP(ack = 9125)
TCP(ack = 11125)
TCP(ack = 13125)
CWND > 5000 B
tRTO > tack
tack
The destination application

Fig. 1.7. Acknowledgment filtering technic in network

The system must comply that TCP session is fully opened and the CWND and retransmission time out values T_{RTO} of TCP are high enough to allow one or several ACK drops. It is noted that ACK clocking scheme, which for both drop and congestion control is used, with the ACK filtering can be destroyed. The current propositions are compelling. However, the discussed apprehensions can be challenged with arguments of (Haitao Wu *et al.* 1999), where the TCP performance in the asymmetric links was analysed. It is shown that asymmetry affects the TCP performance, because it relies on feedback of cumulative ACK from the receiver. In addition, typical TCP is ACK clocked, so the arrivals of ACK on the reverse channel have significant effects on the forward channel throughput. In the networks with bandwidth asymmetry the ACK filtering can work well. This improves the forward TCP throughput and the fairness of competing connections greatly.

In the work of F. Keceli (2007) presents a quite similar study. Analysis of unfairness problem between TCP upstream data and ACK downstream on the unevenly shared wireless channel is provided. It is shown that ACK message filtering increases IEEE 802.11 wireless channel utilization without any dependence to t_{RTO} (Ha *et al.* 2008; Soediono 1989; Bellovin 1996).

The extensive simulations with ACK filtering in (Chen *et al.* 2009) were proposed. It is demonstrated that lowering ACK number can improved TCP performance significantly: achieving up to 25% gain in chain networks and 35% in a complex grid network, compare with typical TCP. In work the ACK filtering motivation follows from fact that short ACK messages consume channel capacity comparable to data packets when the transmission is high rate.

1.10. Acknowledgment Filtering Influence on Router Performance

In order to find out, what real influence the ACK filtering makes to the TCP functionality and how it affects the performance of the network channel devices (routers) an experiments were performed. For this reason Ethernet network with IP routing, and TCP session between two independent nodes (PC0 and PC1/PC2, Linux OS, 2.6.32 kernel, and TCP Cubic version enabled (Ha *et al.* 2008; Bao *et al.* 2010; Jain *et al.* 2011) were created. The structure of network is presented in Fig. 1.8.

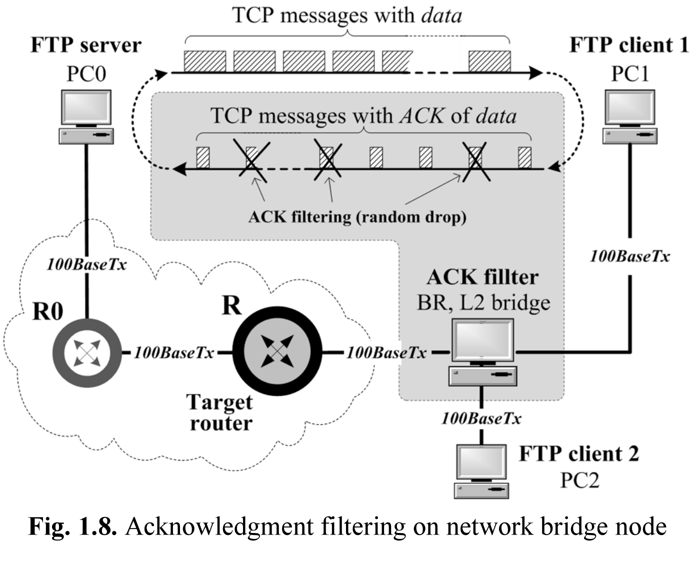


Fig. 1.8. Acknowledgment filtering on network bridge node

In current network the following equipment has been used: PC0 as FTP server – the transmitter of data TCP messages, and PC1/PC2 as FTP client – receiver of TCP data messages (ACK message sender), the transparent Ethernet bridge – BR, target router – R, and additional router – R0 (Cisco 881) as well. For experiments two routers of different generations Cisco 881 and Cisco 1841 were used. The main difference between them is CPU power. All devices were connected with 100 Mbps Ethernet links (100BaseTx). The ACK filtering was implemented on BR device. It has been created on Linux based (2.6.32 kernel) PC with Ethernet bridging <*brctl*> application. This tool was taken because Linux bridging is faster, work as simple switch, and don't make significant impact to flow parameters comparing with routing.

The ACK filtering has been pursued only in one direction from PC1/PC2 to PC0, whereas in opposite direction the traffic passed through BR without any alterations. Filtering was based on exact frame rate control with Committed Information Rate (CIR) and Committed Burst Size (*CBS* = 5 KB). The last one was used to avoid degradation of congestion window on initial growth phase. Filtering was made using <*tc*> application, which drop/policed ACK messages if it exceeds specified rate. The scripting code of ACK filtering is shown in Fig. 1.9.

```
tc qdisc add dev eth1 ingress
tc filter add dev eth1 parent ffff:0
       protocol all prio 1 u32 match u32
       0xaff0001 0xffffffff at 16 classid
       ffff:0 police index 2 rate 12500bps
       burst 102400 mpu 0 action drop/pass
tc filter add dev eth1 parent ffff:0
       protocol all prio 1 u32 match u32
       0x0 0x0 at 0 classid ffff:0 police
       index 3 rate 1bps burst 1 action
       drop/drop
```

Fig. 1.9. Scripting source of <*tc*> policing

The second router (R0) has been used on purpose to keep more realistic IP based network with all routing and switching functionalities. The target router parameters during the experiments with SNMP protocol were collected throw independent router interface. For data transmissions the FTP application has been used. In all experimental iterations file of 400 MB size was transferred. The data speed was controlled on PC0 (FTP server) with <*tc*> script, which shaped to desirable speed without packet loss (delay of traffic only). For this Token Bucket Filtering – TBF was used (Tschofenig *et al.* 2009).

1.11. Performance Evaluation Experiments

The target of first experiment scenario was to find how the ACK message filtering can influence the TCP performance and data transfer integrity. For this, a file of 400 MB size from PC0 to PC1 was repeatedly transferred (Fig. 1.10). On the each iteration the ACK message filtering rate (0%, 20%, 40%, 60%, and 80%; values of the maximal ACK rate without filtering) was changed.

As shown in Fig. 1.10 TCP message rate for entire period was stable in all iterations. The growth of message rate is high and equal at all ACK drop values (events up to ~3 s). This occurred because of two reasons: $W_c(t)$ of used TCP version slightly depends t_{RTO} and ACK filtering is activated only after CBS is exceeded – when the $W_c(t) = W_{max}(t)$. At the end of transfer decline of throughput is seen – the finish of data transfer.

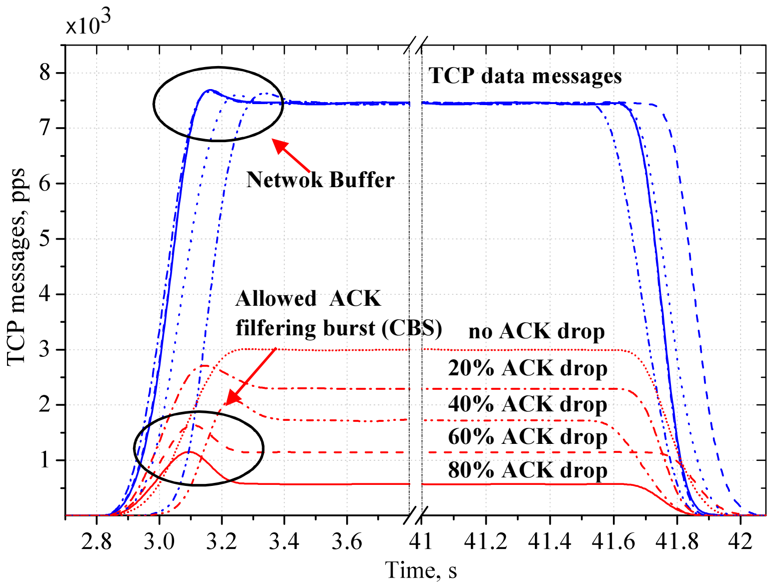


Fig. 1.10. Transmission control protocol messages rate during file transmission with various acknowledgment drop values

The result shows that ACK message filtering does not affect data transfers of single TCP session. Observations shows that transfer remains stable for up to 80% of ACK drops. However, if losses are above 85% the ACK rate becomes less than 1/ t_{RTO}, and data rate degrades fatally. In cases of bigger RTO time the ACK rate limiting could be bigger, but also bigger TCP window size (on TCP sender and receiver) would be required to achieve the desired throughput. Also after packet loss a longer time would be needed to recover from packet drops due to increased RTO timer value and more unacknowledged would be lost.

The comparison of cumulative TCP segments (count of segments on TCP layer) and FTP bytes (count of data on FTP layer) rates during period of file transfer is presented in Fig. 1.11. It is shown that on FTP and TCP layers (lines are coincident) the same amount of bytes was received at any given time period. Consequently, it is possible to do the suggestion, that count of duplicated ACK and TCP data retransmissions are not increased respectively ($t_{\mathrm{RTT}} = t_{\mathrm{data}} + t_{\mathrm{ACK}}$ is less than t_{RTO}).

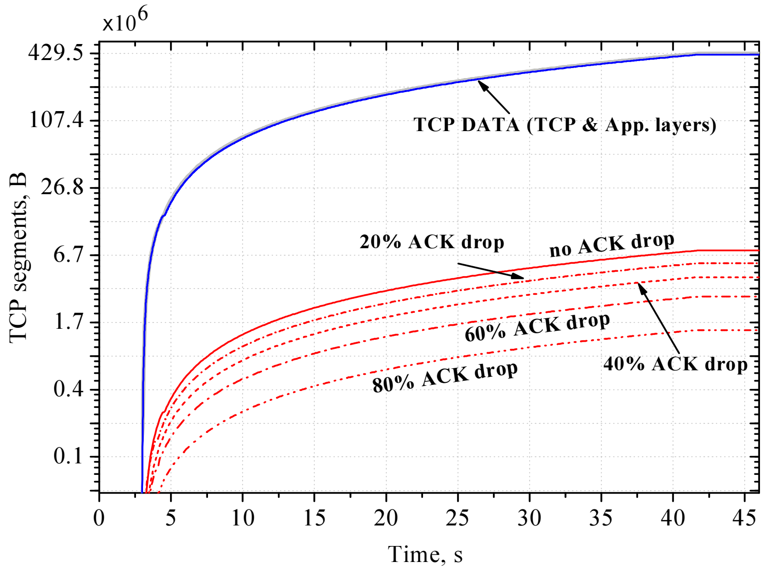


Fig. 1.11. Cumulative growth of transmission control protocol segments count during file transmission for various acknowledgment drop values

The second experiment goal was to find what kind of influence cumulative ACK algorithm has on network equipment performance. For this purpose a file transfer of 400 MB, was performed and CPU load of router (Cisco 881, Cisco 1841) was measured (see Fig. 1.12.). The experiments were performed at various data rates (1, 4, 8, 16, 32, 65, and 90 Mbps) in scenarios with and without filtering and in scenario when 80% of ACK is filtered.

As shown in Fig. 1.12 the CPU load is decreasing when the ACK filtering is used. The same situation is observable for both routers. It is clear that results do not depend on router type and amount of data in TCP message. It depends on amount of processed packets by the router CPU. This is confirming the results presented in (Paredes-Farrera *et al.*; Balakrishnan *et al.* 2002). Moreover, in Fig. 1.12 is observable that CPU load utilization is almost linear, and depends on TCP message rate: if the TCP data rate increases, the ACK rate is increasing too. The functions of the CPU load curves are quite similar for both routers. The difference is only in designed CPU's power.

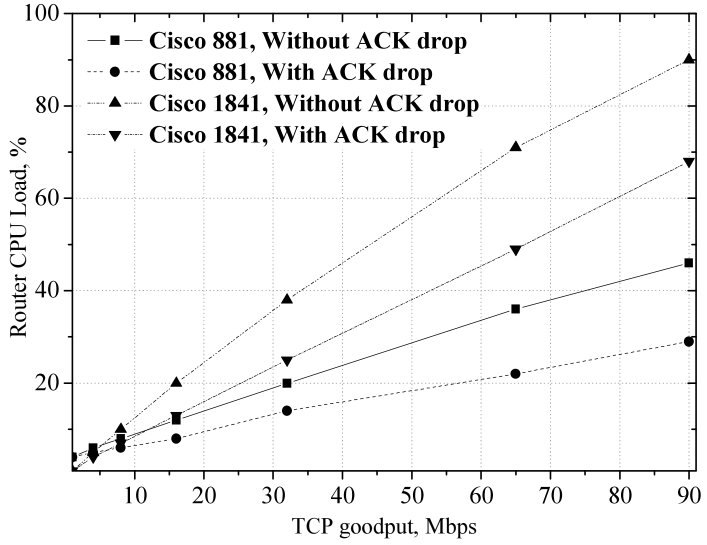


Fig. 1.12. Target routers system load for various traffic rate (TCP goodput) when 80% acknowledgment is dropped

The relation between target router performance and CPU load is shown in Fig. 1.13. Performance increase should be understood as relative CPU load reduction caused by employing of ACK filtering. With ACK message drop of 80% the performance can be increased by 30%, comparing with case when CPU load is 25%, and ACK filtering is not used. Meanwhile, if CPU load is more than 60% (more than routers overload threshold) with 80% of ACK drop the increase of routers performance is ~32%. The reduction of ACK messages could lead to better performance of the network in generally, not only routers but also firewall, proxy or load balancers, who are directly affected by high TCP rates and could have significant gain in performance due reduction of it.

The third experiment issue was to find how the ACK filtering can influence two concurrent and independent TCP sessions. During investigation the PC0 was used as FTP server and PC1/PC2 – as FTP clients. With each TCP session the 400 MB files were transferred. The ACK message filtering was performed on both sessions simultaneously.

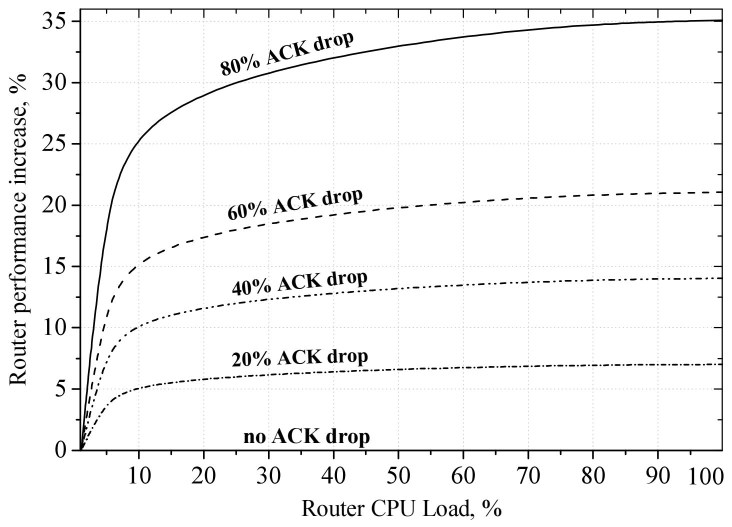


Fig. 1.13. Performance increase on target router system different drop values

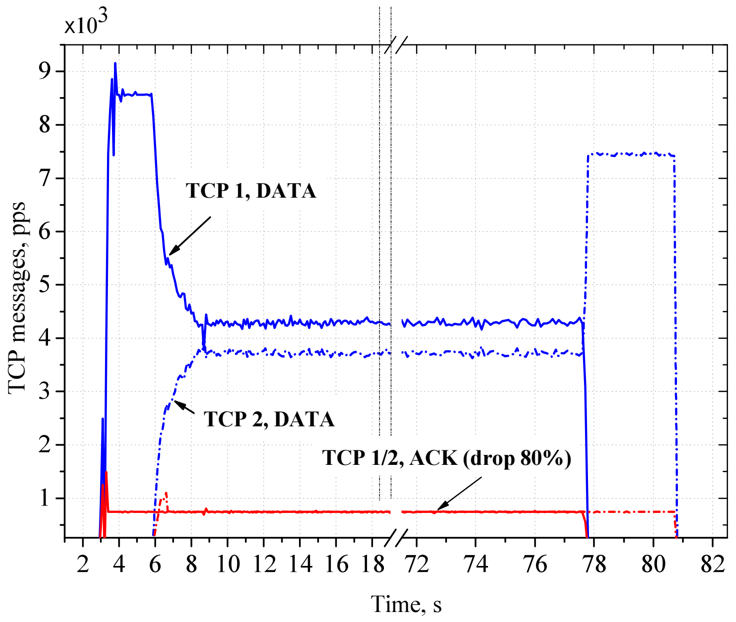


Fig. 1.14. Transmission control protocol message rate during two files transmission: both sessions with 80% of acknowledgment drop

The dependence of two TCP sessions message rate during file transmissions is presented in Fig. 1.14. Both independent sessions remained concurrent and divide channel almost equally (equal channel sharing is not possible due constant TCP channel probing nature (Bott 2014) At 3 s, as in situation with one session (Fig. 1.14), the TCP1 begins to increase the message rate according to $W_c(t)$. While at 6 s TCP2 session is starting too, and about 9 s the message rate of both sessions becomes approx. TCP1 throughput ≈44 Mbps, TCP2 throughput ≈49 Mbps and aggregated ≈93 Mbps.

After 78 s the TCP1 rate is decreasing since finishing of file transfer, while the TCP2 conversely starts to grow-up rate. At 81 s TCP2 is finishing transfer, too. Current fine competition between two independent session's show, that both of them from TCP point of view are working well, and the ACK filtering does not make significant influence on TCP functionality in current conditions.

The dependence of cumulative data during two file transmissions is shown in Fig. 1.15. It can be seen that both sessions collect messages well in TCP layer and in application layer (Fig.1.15.). It means that TCP goodput (transmitted data to upper layer) of both sessions are close to maximal, while the count of duplicated ACK is minimal.

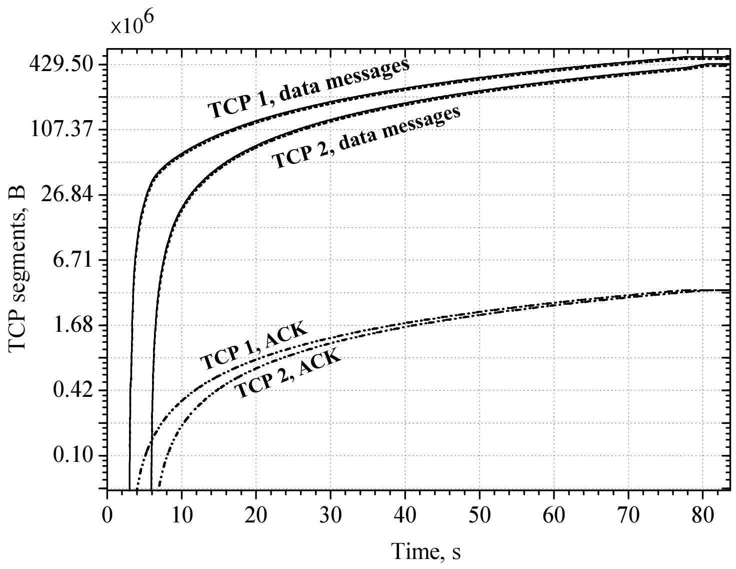


Fig. 1.15. Cumulative growth of transmission control protocol segments count during two file transmission: both sessions with 80% of acknowledgment drops

TCP interoperability defines whether a protocol is fair to other TCP sessions. Therefore, it's important to find how ACK filtering increases unfairness of TCP. For this purpose fourth experiment was performed. The scenario was the same as in previous, only the filtering for one session and the PC0 without shaping was used (Fig. 1.16).

The dependence of message rate of two TCP sessions on file transmissions time is presented in Fig. 1.16. The graph shows that both sessions divide channel almost equally as in previous scenario. The insignificant unfairness between two TCP sessions was observed (Jacobson *et al.* 1992). But this is typical case for real network situation were one TCP session has a bigger throughput gain. To better understand unfairness better, the extensive analysis must be done with various TCP versions and modification also different OS systems. And this is the main issue of future investigations. In addition, during the tests the ACK message rate filtering also adds delay to network, this have negative impact to TCP fairness. Because the TCP session with ACK limiting enabled is filtered on Linux box, and additional delay of packet processing is added to the total RTT time of TCP.

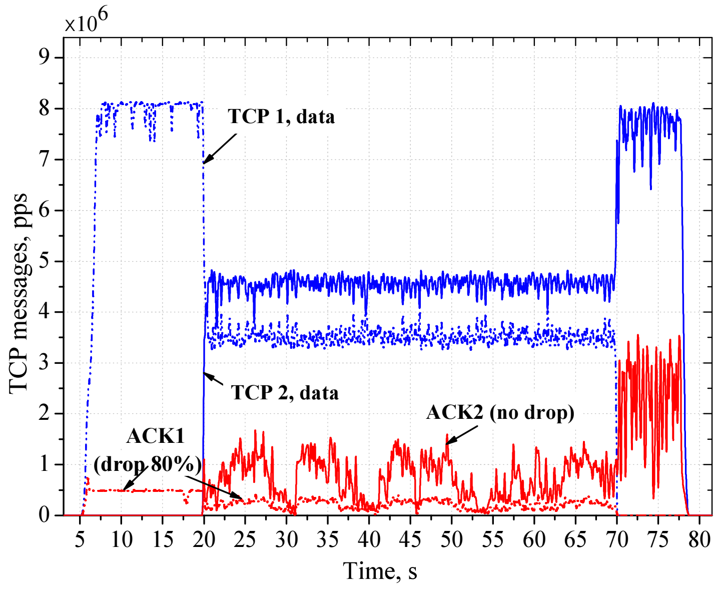


Fig. 1.16. Transmission control protocol message rate during two files transmission: one session without acknowledgment drop, and other with 80% of drops

1.12. Acknowledgment Filtering in Heterogeneous IEEE 802.11 Networks

In order to find out how described ACK filtering technic performance in IEEE 802.11 wireless networks an experiments were performed. Due to shared medium of wireless link the ACK filtering has big impact not only to network equipment but also to shared wireless medium. Reducing ACK message rate can increase wireless link capacity and ensure better performance for applications or reduced wireless saturation.

To better understand how ACK filtering is working and is the impact of it to heterogeneous networks and investigation of ACK messages rate and its impact to IEEE 802.11 was made. In wireless networks TCP or other session orientated protocols is competing to get network access or capacity needed for data transmission. IEEE 802.11 MAC protocol provides a fair access to the shared wireless medium through two different access functions: polling based protocol, called the Point Coordination Function (PCF) and contention based access protocol, called the Distributed Coordination Function (DCF) (Vindašius *et al.* 2015; Potorac *et al.* 2015).

The DCF is more frequently implemented compared with PCF and is used for automatic medium sharing between compatible stations (Choi 2012; Zhao *et al.* 2009). This is done by using CSMA/CA. Before a station can send data, it must sense the wireless medium to determine if it is free. If it is so, the transmission may proceed, other ways the station will wait until the end of the in progress transmission (Potorac *et al.* 2015; Balachandran *et al.* 2002). The CSMA/CA requires a minimum specified space gap between frame transmissions to ensure that the medium has been idle for the specified time period.

To ensure that the DCF uses to basic rules: first the medium must be free more than for DIFS (Distributed Inter Frame Space time) interval, and the second rule says, to reduce the probability of sending two stations at the same time, the stations must wait random back of period – T_b (Zhao *et al.* 2009; Miguel *et al.* 2011):

$$T_b = \text{rand}\left(CW\right) \cdot T_{\text{slot}}. \qquad (1.19)$$

Where rand(CW) is a pseudo random integer value selected from interval from zero to CW and the T_{slot} value correspond to *slot time* which is physical characteristic of IEEE 802.11 standard.

1.13. Acknowledgment in IEEE 802.11 Networks

Like in TCP the IEEE 802.11 MAC uses acknowledgment (ACK) frames upon successful reception of a data frame. Only after receiving an ACK frame correctly, the transmitter assumes successful delivery of the corresponding data frame. To ensure that 802.11 ACK frame is send before other data frame, the wireless client wait small time gap SIFS (Short Inter Frame Space) before transmitting acknowledgment frame. This frame exchange sequence, prevents other stations of sending data – which are required to wait for the medium to be idle for a longer time when SIFS for attempting to use the medium (Jain *et al.* 2005; Xiao *et al.* 2002; Vindašius *et al.* 2015). This allows to successfully completing the frame exchange sequence. The successful frame transmissions is shown in Fig. 1.17.

If no wireless ACK frame is received within a SIFS interval due possibly to an error in reception of the preceding data frame (see Fig. 1.17) the transmitter will contend again for the medium to retransmit the frame after an ACK timeout (Kaur *et al.* 2010; Misra, Sudip, Isaac Woungang 2009).

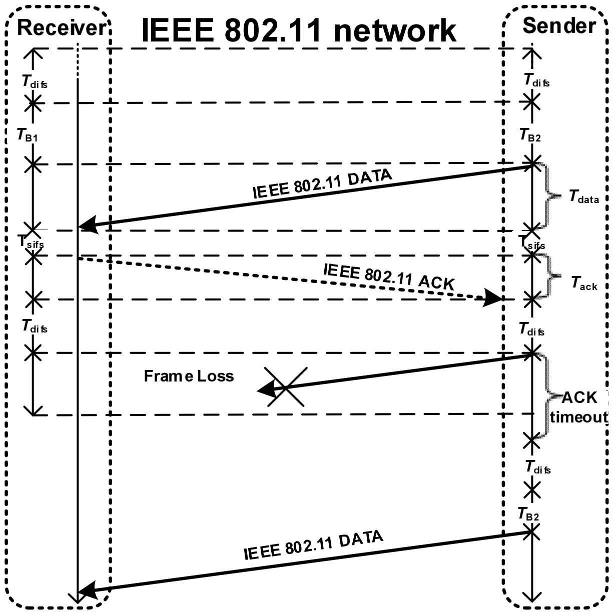


Fig.1.17. IEEE 802.11 wireless frame exchange process

In basic DCF mode of IEEE 802.11 MAC standard the good throughput – G_W of wireless media can be calculated based on following function (Gast 2005; Bianchi 2000):

$$G_{\mathrm{W}} = \left(\frac{8 \cdot L_{\mathrm{DATA}}}{T_{\mathrm{DCF}}} \right). \tag{1.20}$$

Where L_{DATA} is the length of payload size of 802.11 data frame, and T_{DCF} the total time needed to make successful full frame exchange sequence. The T_{DCF} time consists of sum of times needed to make frame exchange in wireless media can be determined by the equation (Xiao 2005):

$$T_{\mathrm{DCF}} = T_{\mathrm{DIFS}} + T_{\mathrm{B}} + T_{\mathrm{DATA}} + T_{\mathrm{SIDFD}} + T_{\mathrm{ACK}}\,. \tag{1.21}$$

The T_{PRE}, T_{PHY}, T_{DIFS}, T_{SIFS} variables are mostly constants and are only differ based on different IEEE 802.11 standards. The T_{DATA} and T_{ACK} time frames are calculated based on payload size and sending data rate of physical wireless media or from T_{OFDM} – transmission time of one OFDM symbol and N_{DBPS} – number of data bits per OFDM symbol:

$$T_{\mathrm{DATA}} = T_{\mathrm{PRE}} + T_{\mathrm{PHY}} + T_{\mathrm{OFDM}} \cdot \left\lceil \frac{22 + 8 \cdot \left(L_{\mathrm{mac}} + L_{\mathrm{DATA}} \right)}{N_{\mathrm{DBPS}}} \right\rceil; \tag{1.22}$$

$$T_{\mathrm{ACK}} = T_{\mathrm{PRE}} + T_{\mathrm{PHY}} + T_{\mathrm{OFDM}} \cdot \left\lceil \frac{22 + 8 \cdot L_{\mathrm{ACK}}}{N_{\mathrm{DBPS}}} \right\rceil. \tag{1.23}$$

From (1.21) and (1.22) the time needed to transmit the information over wireless IEEE 802.11g/a link is closely related to its size of data packet the wireless access point receives in input (in our cases from Ethernet). As the data packet get smaller on Ethernet the total time needed in wireless links increases due to T_{PRE}, T_{PHY}, $\mathrm{T}_{\mathrm{DIFS}}$ $\mathrm{T}_{\mathrm{SIFS}}$ values. The smaller the data packet is send over IEEE 802.11a/g network, the bigger is the protocol overhead for this packet (Gast 2005; Xiao 2005).

1.14. Performance Evaluation of Acknowledgment Filtering in IEEE802.11 Networks

In order to find out what is ACK message filtering in IEEE 802.11a networks and how it affects the TCP performance an experiments were performed. For this reason IEEE 80211a wireless network with IP routing, and TCP session between two independent nodes (PC1 and PC2, Linux OS, 2.6.32 kernel, and TCP Cubic version enabled) were created. The structure of network is presented in Fig. 1.18.

In testing network the following equipment has been used: PC1 as TCP client – the transmitter of data TCP messages to PC2 – receiver of TCP data messages. The Wi-Fi AP and PC2 were connected with 100 Mbps Ethernet links (100BaseTx) were created. The structure of network is presented in Fig. 1.18.

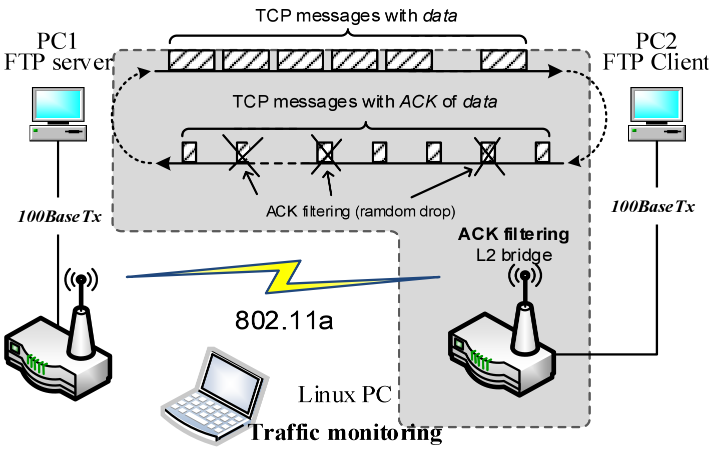


Fig. 1.18. Acknowledgment filtering on Linux wireless access point node

The ACK filtering has been pursued only in one direction from PC2 to PC1 via wireless AP, whereas in opposite direction the traffic passed through wireless AP without any alterations. The ACK filtering was implemented on AP2 node based on exact frame rate control with Committed Information Rate (CIR) and Committed Burst Size (*CBS* = 5 KB). It has been created on Linux based (kernel 2.6.32) Ethernet bridging <*brctl*> application like in tests before. ACK filtering was made using <*tc*> application, which drop/policed ACK messages if it exceeds specified rate.

The theoretical throughput of TCP based on ACK message filtering value in IEEE 802.11a network can be calculate based on (1.18) and is provided as follow:

$$G_{\mathrm{TCP}} = \left(\frac{N \cdot 8 \cdot L_{\mathrm{DATA}}}{N \cdot T_{\mathrm{DCF\_DATA}} + T_{\mathrm{DCF\_ACK}}} \right). \qquad (1.24)$$

Here $T_{\mathrm{DCF\_DATA}}$ is time the time needed to transmit one TCP data segment, $T_{\mathrm{DCF\_DATA}}$ – time needed to send one ACK message, N – number of TCP segments acknowledged by one ACK.

By considering the fact that the TCP segment size is constant payload size (1416 B) the max throughput in IEEE 802.11a links with different values of ACK filtering is showed in Fig. 1.19.

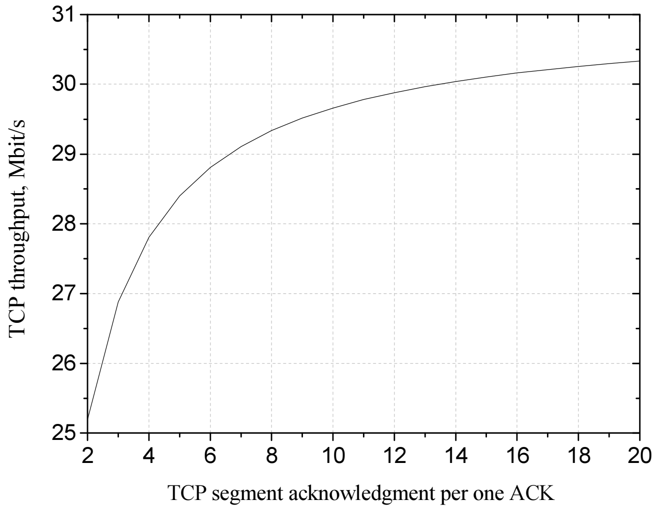


Fig. 1.19. Transmission control protocol throughput dependency from acknowledgment messages rate in IEEE 802.11a network

In practice TCP throughput can and in most cases is much lower due to different TCP windows size value and changing t_{RTO} values or packet drops (Potorac *et al.* 2015; Li *et al.* 2009). In Fig. 1.19 the ideal, theoretical situation is showed, without any negative conditions.

To compare the theoretical results from (1.19) and to see how TCP work over wireless links, a real world test was conducted with ACK messages filtering and without it. In first test a TCP data stream from PC1 to PC2 was created, the results are showed in Fig. 1.20. More than half all wireless frames are the IEEE 802.11a ACK messages send for TCP data and ACK messages transmitted from Wi-Fi AP to Wi-Fi client and back. Though the messages are short in time but it occupies link resources and generates unneeded overhead especially in links with small packet drops. In total the 802.11 layer 2 frame rate exceeds 600 fps from which only the half are TCP data frames and contains information.

During all test the average TCP throughput did not exceed ~ 25 Mbit/s that is considered the maximum practical speed for IEEE 802.11a for TCP protocol (see Fig. 1.19). In case the ACK messages is generated for every second TCP data segment.

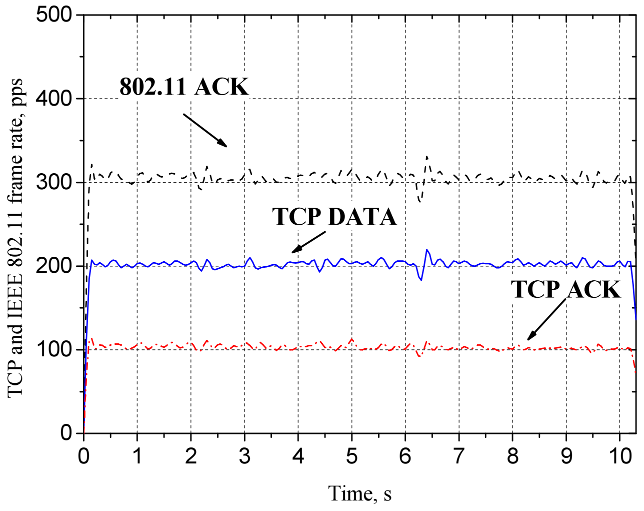


Fig. 1.20. IEEE 802.11 frame rate comparison to Transmission Control Protocol message rate without Acknowledgment filtering enabled

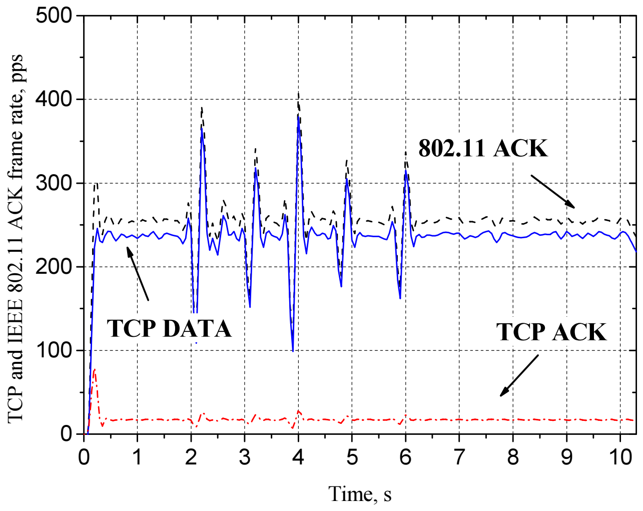


Fig. 1.21. IEEE 802.11 frame rate comparison to Transmission Control Protocol message rate with Acknowledgment filtering enabled

In order to increase the practically useful TCP throughput over Wi-Fi two different methods were used, which reduce the unneeded technical overhead. The first and most simple method is to reduce ACK message rate that in most cases, especially in high speed TCP session, is not needed and does not have any impact to throughput and stability of existing TCP session (Fig. 1.21).

The ACK filtering has been pursued only in one direction from PC2 to PC1, whereas in opposite direction the traffic passed through bridge without any alterations. During the testing the maximum stables ACK message rate of ~ 100 Kbit/s was observed, with which the TCP session gained the best TCP frame rate and sustained stabled during for all testing time (see Fig. 1.21).

The first method not only reduce the IEEE 802.11 acknowledgment rate due to reduced ACK rate, but not eliminating IEEE 802.11 acknowledgment frames. It allows to increase the TCP throughput up to 30 Mbit/s.

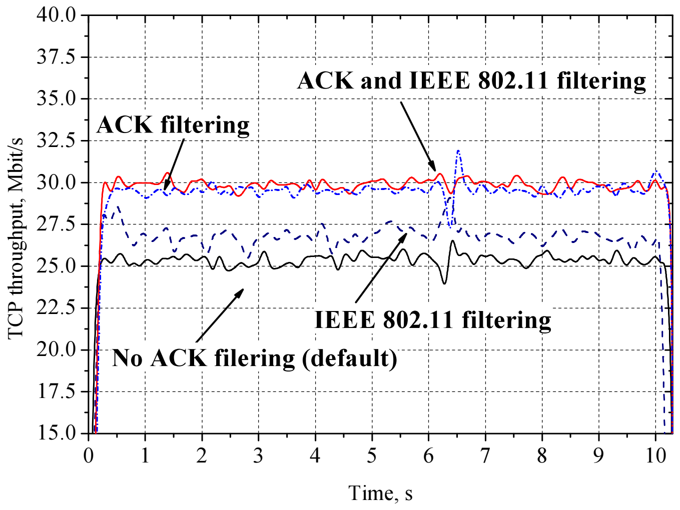


Fig. 1.22. Transmission Control Protocol throughput with Acknowledgment and 802.11 filtering methods

The second method is based on additionally filtering IEEE 802.11 ACK frames for TCP acknowledgment messages. This was accomplished by <*ebtables*> application which change the destination MAC address of Ethernet frame from unicast to multicast of ACK messages, before leaving the PC2 NIC interface (Wu 2012). This allows to disable the IEEE 802.11 acknowledgments mechanism for PC2 in wireless transmission link for ACK messages.

The main difference from TCP filtering from IEEE 802.11 is that the TCP acknowledgment messages are not filtered from TCP stream, but change the wireless mechanism by disabling the IEEE 802.11 acknowledgments for ACK messages.

Of course the ACK and wireless filtering could be implied to both stream (IEEE80.11 and ACK) to reduce the unneeded technical overhead, but in total the difference between the ACK filtering and TCP plus 802.11 filtering is very small and has not impact to TCP throughput (Fig. 1.22).

From results of the first method (Fig. 1.21) the reduction of ACK messages rate not only reduces the IEEE 802.11a acknowledgment rate up to 50% but also increased the TCP throughput up to ~30 Mbit/s. That allows us to gain 20% in throughput or 5 Mbit/s of traffic increase. This allows reduce the IEEE 802.11a acknowledgment frame rate and increase performance of wireless nodes due to reduced saturation of wireless links. From test seen in Fig. 1.20 and 1.21 indicates that ACK rate limiting has no or little impact to TCP data transmission over IEEE802.11a links and can be used in real world wireless networks.

1.15. Conclusions of 1 Chapter and Formulation of the Thesis Objectives

1. ACK limiting in network equipment does not significantly affect the TCP data transfer in normal network conditions. The TCP data transfer remains stable for up to 80% of ACK drops. However, if losses are above 85% the ACK rate becomes less than $1/t_{\mathrm{RTO}}$ and session is terminating immediately.
2. The performance of router CPU depends on ACK count and can be increased with ACK limiting is enabled. The CPU load utilization is linear, and depends on data rate: if the TCP data rate increases, the ACK rate is increasing too.
3. On CPU load of 25% with 80% of ACK drop, it is possible to increase the router performance by 30%. Meanwhile, on CPU load of 60%, on the same ACK limiting conditions, the performance can be increased by 32%.
4. The ACK limiting on network equipment can be used not only with single TCP session but also with concurrent sessions. Results show that two sessions work well, without any evident signs of the instabilities; although a slight unfairness among TCP sessions were observed.
5. ACK filtering in IEEE802.11b/a wireless networks can give performance gain of 20% (in good wireless conditions). By using the same ACK limiting algorithm for IEEE 802.11 acknowledgment frames, a reduction of wireless saturation can be achieved and even better improve throughput of other wireless nodes can be succeeded.

The following tasks must be solved:

1. To investigate TCP packet processing and main function in Linux OS kernel for TCP overhead reduction by limiting ACK message rate.
2. Create dynamic ACK limiting method for Linux OS kernel for efficient TCP overhead reduction based on TCP flow characteristics.
3. Experimentally verify the efficiency ACK limiting method in Linux OS kernel and the impact of it to TCP data transmissions and system stability.
4. Find the upper and lower values for efficient ACK limiting in Linux OS kernel.

2

Linux Kernel Acknowledgment Limiting

During the past decade the Internet was increasing in all directions, from PC web browsing to embedded and mobile devices which are interconnected in global Internet with wired or wireless technologies (Jacobson *et al.* 1992: 1–37; Kaur *et al.* 2010; Kotz *et al.* 2002). The new wave of Linux based devices is growing in numbers in global network every day, it generates new challenges not only for network engineers but also to system developers. The main problem of network layer is to provide reliable and guaranteed network service for customer as fast and as cheap as possible. From system developers point they have to reduce the power consumption and increase the performance or speed of the product.

The TCP optimization problem or performance improvements issue is substantial for low power embedded Linux based devices. The lack of CPU performance is the main bottleneck for these devices for achieving better performance and response time. It can have huge impact to the whole system performance if the embedded CPU is consumed by network stack processing.

TCP optimization is also important to high-end server or clusters. By additionally optimizing or improved TCP protocol it can also improve data center performance not only in service delivery layer but also reduced power consumption, due to reduce need for computing power of CPU (Priescu *et al.* 2012).

In second chapter of dissertation a closer look to packet flow in Linux OS kernel network stack will be made. Finally some modification of current Linux kernel will be made to see the impact of the ACK limiting it makes to TCP performance and stability.

For TCP kernel modifications a real field test were performed. Due to complexity of Linux kernel and OS, it is imported to make the testing as real as possible. Because network simulators and emulators cannot deal with such level of complexity, which persist in real systems, the real field testing's were performed (Schimmel *et al.* 1994).

The decision to use Linux Kernel also came from the excellent support and community, also it has good (not excellent) documentation, what most of OS lack. Moreover Linux kernel is used in big variety of modern devices, starting from CPE to and android and supercomputers, and it is considered as main operating system for Internet and communication.

The research results are published in author publications: Statkus *et al.* (2013), Statkus *et al.* (2012). The main results are announced in scientific conference: "Electronics" (Vilnius 2010).

2.1. Data Receiving in Linux Kernel

Before explaining Linux kernel network stack, it must be stated that current Linux kernel is not final and like real world organism it's constantly evolving and growing, by including new features or code modifications. These changes are made daily, starting from kernel developers to first time students and learners. Some of these changes persist for long time and are included in next kernel release, some code changes stay in patch state or testing kernel possibility. The following code and procedure explanation of kernel network stack are not permanent, and will change in time, but in general the main concept of kernel network stack should stay the same, at least for the third release of the kernel (3.10 release) releases. In this chapter a closer look to Linux kernel network stack procedures and functions will be made to better understand the kernel work flows and TCP algorithms which are used for data processing.

Linux kernel is event driven system, which start processing data after an event or interrupt is received (Seth *et al.* 2008). If the system receives a packet to NIC interface, an interrupt is generated to CPU (for simplicity a system with one CPU is considered). Every system interrupt come with unique interrupt number, by this number the OS select needed driver to handle with arises interrupt (the procedure responsible for selecting driver is called interrupt handler). In Ethernet network example if a packet is received the system call NIC driver to handle the received interrupt (Bhuiyan *et al.* 2009). The NIC driver first puts the received packet in NIC memory, for Ethernet CRC and integrity validation checking (Parham *et al.*). If received packet is good it is moved to system memory, which was allocated by

NIC for data receiving. In case the packet receiving rate is too fast for system to handle or allocated system memory is too small the NIC must drop received data messages, it also protects the system from heavy message flood (Stevens *et al.* 2014).

After the data is put to system memory the kernel handler takes place to process received data, it calls *napi_schedule()* function for processing received packets. The *napi_shecule* function call software interrupt handler. This is done to queue network hardware interrupts and allow the kernel to process the received data, not allowing hardware interrupt to overrun the CPU.

After a packet is received the *softirq* call the *net_rx_action()* function, which pools received data from the NIC. The received data is wrapped it to *sk_buff* structure and is passed to *netif_receive_skb()* function for further processing (Seth *et al.* 2008; Jacobson 1990). Then the received data, based on packet type, is send to upper layer for further processing. To process the received data packet the kernel must know what type protocol is inside the received Ethernet data frame. This is done by looking to 2 B *Ethertype* field (see Fig. 2.1) that indicates the upper layer protocol.

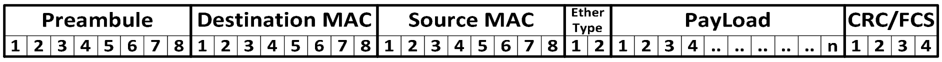

Preambule
Destination MAC
Source MAC
Ether Type
PayLoad
CRC/FCS
1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2 1 2 3 4 .. .. n 1 2 3 4

Fig. 2.1. Ethernet header structure

This information is copied to *sk_buff(sk–>protocol)* structure field, which is used for further packet processing in kernel. If it is an IP packet type *ip_packet_type*, kernel calls *ip_rc()* function for data processing.

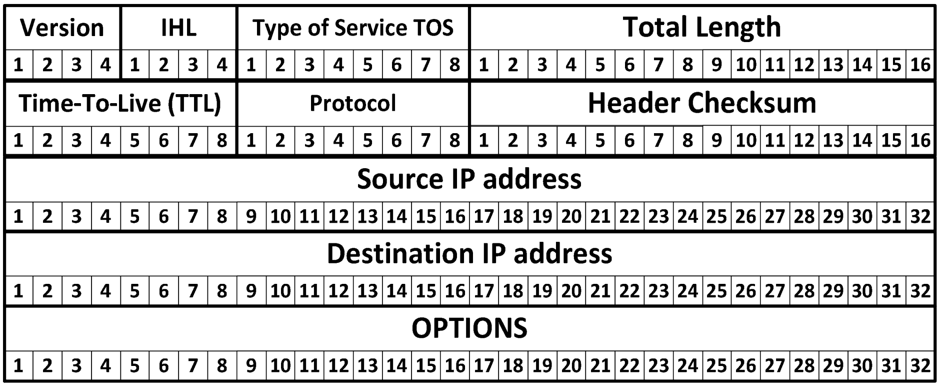


Fig. 2.2. Internet protocol header structure

The *ip_rc()* function examines the structure and checks the header checksum validation. If the packet is valid it is passed through the netfilter code, for firewall processing, if needed the modification for packet are made depending of firewall settings (SNAT, DNAT, etc.). If the packet destination is local system the *ip_local_deliver* function is called, which make final packet assembly and passes the packet to *ip_local_deliver_finish()* function which removes IP header and finds the upper layer protocol by looking to IP header *protocol* field (see Fig. 2.2).

Every upper layer protocol has its own packet procedure for packet processing, if the value is 6 in decimal, it means that inner protocol is TCP (see Fig. 2.3). For TCP packet processing kernel call *tcp_v4_rcv()* function, which is called from *tcp_ipv4.c* file (see Fig. 2.6). As obvious the *tcp_v4_rcv()* function defines only IPv4 standard, and for processing IPv6 TCP header must call *tcp_v6_rcv()* receive function from *tcp_ipv6.c* file, it is done because for TCP checksum is computed from logical TCP IP header also known as "pseudo header", which includes IP address and protocol fields from IPv4 header (Seth *et al.* 2008; Bach 1990; Lahey 2000). The full "pseudo header" is shown in Fig. 2.4.

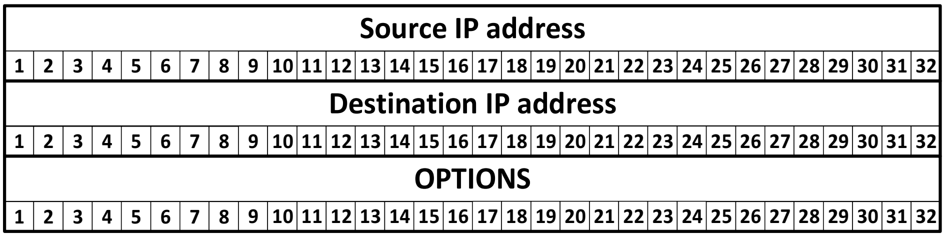


Fig. 2.3. Internet protocol logical header structure

The reserved fields in most cases it is set to 0 and not used, and only four fields: source, destination address, protocol and TCP segment length (header and data length) fields are needed to make TCP checksum (Fig. 2.4).

During the TCP checksum calculation the CRC fields is left empty (field with zero), after checksum calculation the field is filled with value, then the packet is created (Parham *et al.*). After the packet is validated it goes for further processing depending from the state of TCP stats and packet type (in example if RST or FIN fields are marked). Then the kernel tried to find the socket it belongs by calling *tcp_v4_lookup/inet_lookup_skb* function, it looks to hash tablet packet new (no active session).

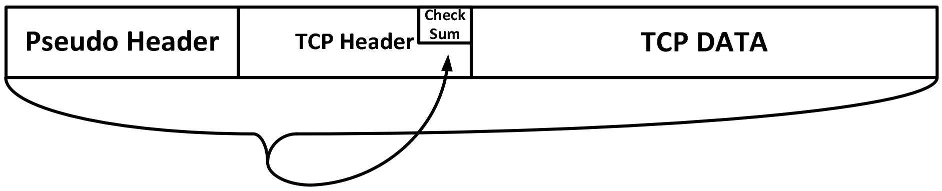


Fig. 2.4. Transmission control protocol cyclic redundancy check calculation

After connection state is check *inet_lookup_skb* the kernel executes the *tcp_v4_rcv* function for further packet processing (see Fig. 2.5). If connection is established and no conditional state occurs (timeout, out of order packet) the connection goes to "Fast Path", by calling *tcp_rcv_estableshed* function. The function goes through the TCP header, by checking the sequence number and putting the received data to the socket buffer for application processing and sending the acknowledgment packet to the data sender (Unzner *et al.* 2014; Beekmans 2010).

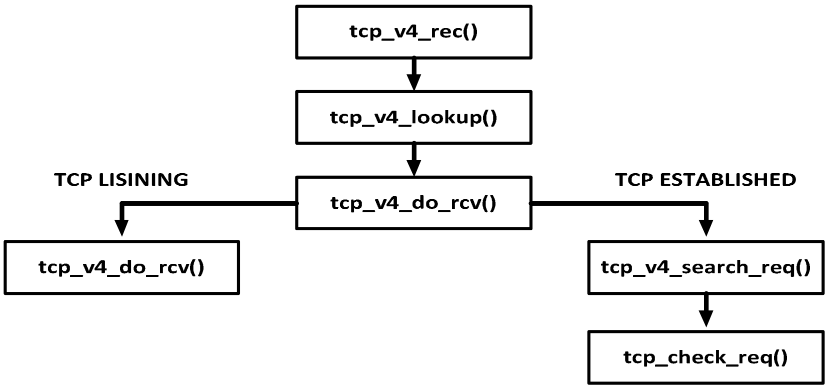


Fig. 2.5. Linux kernel transmission control protocol packet processing algorithm based on session status

2.2. Linux Kernel Transmission Control Protocol Acknowledgment Generation

After the TCP data is processed by *tcp_rcv_established* function and is validate the kernel calls *tcp_ack_snd_check* function (see Fig.2.6) for checking if acknowledgment sending is need (Seth *et al.* 2008).

net/ipv4/tcp_input.c

```
4796 static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
4797 {
4798         struct tcp_sock *tp = tcp_sk(sk);
4799
4800             /* More than one full frame received... */
4801                if  (((tp->rcv_nxt  -  tp->rcv_wup)  >  inet_csk(sk)-
>icsk_ack.rcv_mss &&
4802              /* ... and right edge of window advances far enough.
4803               * (tcp_recvmsg() will send ACK otherwise). Or...
4804               */
4805              __tcp_select_window(sk) >= tp->rcv_wnd) ||
4806             /* We ACK each frame or... */
4807             tcp_in_quickack_mode(sk) ||
4808             /* We have out of order data. */
4809             (ofo_possible && skb_peek(&tp->out_of_order_queue))) {
4810                 /* Then ack it now */
4811                 tcp_send_ack(sk);
4812         } else {
4813                 /* Else, send delayed ack. */
4814                 tcp_send_delayed_ack(sk);
4815         }
4816 }
```

Fig. 2.6. Linux kernel input packet processing in *C* code

The *tcp_ack_snd_check* function receives two arguments, the packet *sk* structure of packet and *ofo_possible* variable, the *ofo_possible* is 1 or 0, showing if out of order segments were received (in case of one) (see Fig 2.6).

The *tcp_ack_snd_check* function decides if TCP acknowledgment must be send now, execute *tcp_send_ack()* function or can be delayed *tcp_send_delayed_ack()* based on four if conditions:

1. The received data segment or unacknowledged data must by more than one maximum segment size for defined in session *icsk_ack.rcv_mss* variable. This comes from RFC 1122 or STD003 specification (section 4.2.3.2), saying that "an ACK SHOULD be generated for at least every second full sized segment" (Berkeley *et al.* 2011; Bott 2014) showed in Fig. 2.6 line 4801. From the logical condition seen in first part of code, the unacknowledged data must be more than *inet_csk(sk)– >icsk_ack.rcv_mss* value. Theoretically it is not according the RFC, but in practice this condition is met after receiving the second TCP segment. That is very important to note here is that the RFC define that the ACK should but not must be generated. So basically RFC allows us to generate the ACK more rarely (see Fig. 2.6 line 4805).
2. The TCP receive window or usable buffer space must be bigger than the receive windows or advertised TCP window by the server to the client. This is done to overcome the Silly Window syndrome (SWS) problem,

which was first defined in RFC 813. It occurs due the bad system implementation of TCP flow control or due to the slow system, which consumes data slowly or can not handle the received information. In such conditions the receive windows *rcv_wnd* variable is filled with data much faster than it can handle it (clean up the receive buffer). In such condition the kernel must reduce the advertised window, by sending the update size to the client. This condition would go on until the receive window is set to minimal allowed size, making the data transmission ineffective. By forbidding the server to send ACK messages, the kernel reduces the packet flow rate, the client must wait for ACK for sending more data and reduce the server load. This condition also satisfies the RFC 5681 that "an ACK SHOULD be generated for at least every second full sized segment" (Bott 2014) seen in Fig. 2.6 line 4807.

3. The third condition check if *tcp_in_quickack_mode(sk)* any data are send back to TCP client, in such case the TCP connection in interactive state and an ACK packet must be send immediately (like telnet or remote data access information applications). In the other way the kernel would wait up to 500 ms before sending an ACK message (Seth *et al.* 2008) (see Fig. 2.6 line 4809).
4. The final conditions check if the server receives out of order data, by checking the *ofo_possible* variable and the looking to receive queue *tp– >out_of_order_queue*, to see if any out of order packet are received. This must be done for faster data recovery and improves TCP recovery time after a loss RFC 5681 – "A TCP receiver should send an immediate duplicate ACK when an out of order segment arrives" (Bott 2014). The purpose of this ACK is to inform the sender that a segment was received out of order and which sequence number is expected. This condition usually happens if packet loss or corruption occurs in the link between the client and server.

After checking these conditions (the first and second are in conjunction), kernel can send ACK message immediately if one of three conditions are true, else the sending of ACK message must be delayed, by calling the *tcp_send_delayed_ack* function, which adjust the sending time based on RTT value and system minimum and maximum delay values. In case where is no data send back over the same TCP session and push bit is not set, the TCP acknowledgment generation must be done on two first conditions (if more than one full size segment is received and kernel has enough space in receive buffer) (Nagle 1984; Stevens *et al.* 1990).

2.3. Acknowledgment Sending Function

As it was shown in previous chapter (how ACK limiting impact network device performance), the TCP acknowledgment messages in high rate can directly influence network equipment, connection links and network endpoints. Also in high speed links, the bottleneck can becomes one of the TCP sender or receiver, not able to handle high ACK message rate. By reducing the TCP acknowledgment rate the system can significantly reduce network load and increase TCP performance in embedded or low CPU power device.

In Linux OS kernel ACK generation for not interactive data transfer, depends mainly from two conditions: *icsk_ack.rcv_mss* and if the system has enough free memory in receive buffer (of course if the system does not encounter any network issues like packet drops or reordering) showed in Fig. 2.6.

As the second variable plays important role in system load control on heavy loaded systems with high TCP rate, it is easier to change *icsk_ack.rcv_mss* variable value to control ACK rate. By changing Linux kernel code it is important to note that most of the code and variables are reused in other parts of kernel code and in most cases will have negative impact to system stability or performance.

To change ACK message or *icsk_ack.rcv_mss* variable an additional C variable (*tp–> tcp_ack_rate*) must be included. Which will increases the *icsk_ack.rcv_mss* size by factor and should not impact any other system code. By doing it the system could easily reduce or increase ACK generation rate without impacting other kernel code and check current ACK generation rate. A negative factor of it that kernel will have to store more TCP state system variables and use system resource to calculate or change its value during the TCP session.

2.4. Start of Acknowledgment Rate Limiting

The biggest issue of ACK rate limiting and TCP in general is that the operating system running TCP server or client does not know anything about network condition (link throughput, delay, transport technologies) and how it is changing during the time. Meaning that TCP must adapt and change conditions after disorder arise in network or remote system. In addition the TCP receiver does not know in what state of TCP (slow start, congestion avoidance, fast recovery, etc.) is the data sender. In example if ACK rate limiting is activated to early the system will have negative impact to TCP data transmission speed ant TCP stability due to the fact that the TCP sender is still in Slow Start and CWND is increasing. If it is activated to late, most sessions will have no positive impact to TCP performance or the TCP session will ended without reach the point when ACK limiting should be activated (system would consume CPU cycles but not use the algorithm).

As explained before the TCP client sending speed basically depends from CWND size which is controlled by Slow Start congestion window algorithms and also delay (Jacobson 1988). To get best TCP performance using the ACK rate limiting first operating system must allow the TCP session to fully open the CWND (to reach the maximum allowed size RWND) and TCP session must be in congestion avoidance phase. After it the ACK rate limiting algorithm can be activated, by reducing ACK rate in small fractions over the time. A big reduction of ACK could lead to NDP increased or the new RTT could be bigger than RTO and TCP session would got to slow start phase.

The problem with TCP receiver is that it does not get any information or indications about client side TCP status like CWND size (is it in maximum or not). In addition the TCP receiver does not know when the Slow Start phase stops and congestion avoidance start on TCP sender. The only information the system can rely is the RFC condition that the sending window increase must not be bigger than 1 *SMSS* per RTT in congestion avoidance phase (RFC 5681) and in Slow Start the sending window is increased by most *SMSS* bytes for each ACK received (RFC 5681). As the Slow Start is more exponential function and the Congestion Avoidance is more linearly (Bott 2014), it is clear that more time is spent in growing the same delta CWND size. So this presupposition allows us to calculate the worst case scenario, if the TCP data sender (client) starts from congestion avoidance phase. By calculating the time or packet count needed for the TCP sender to fully open the CWND to max allowed size the kernel can tell when ACK rate limiting must be enabled. The CWND growth function is defined equation below (Bott 2014):

$$W_{\text{CWND}} = W_{\text{CWND0}} + \frac{SMSS^2}{W_{\text{CWND}}}. \tag{2.1}$$

Where W_{CWND} is new TCP congestion window, W_{CWND0} is the TCP congestion windows before and SMSS is the sender maximum segment size in byes.

In Linux kernel TCP stack this growth function is replaced with a loop function which increased the CWND no more than one *SMSS* per RTT showed in Fig. 2.7.

From code example above the *snd_cwnd* (congestion window) is increased in loop after *snd_cwnd_cnt* (the number of data segments have been transmitted in the current congestion window) becomes bigger then *w* variable (current congestion windows value). The function is activated after ACK is received, indicating that the send data has been successfully received.

net/ipv4/tcp_cong.c

```
void tcp_cong_avoid_ai(struct tcp_sock *tp, u32 w, u32 acked)
{
         /* If credits accumulated at a higher w, apply them gently now. */
        if (tp->snd_CWND_cnt >= w) {
                tp->snd_cwnd_cnt = 0;
                tp->snd_cwnd++;
        }

        tp->snd_cwnd_cnt += acked;
        if (tp->snd_cwnd_cnt >= w) {
                u32 delta = tp->snd_cwnd_cnt / w;

                tp->snd_cwnd_cnt -= delta * w;
                tp->snd_cwnd += delta;
        }
        tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_cwnd_clamp);
}
```

Fig. 2.7. Linux kernel congestion function growth algorithm source code

RFC congestion function (2.1) and the Linux kernel stack code is approximation of conditions that the CWND must increase by one full sized segment per RTT (Touch 2007). The difference between two equations is minimal and can be equated and written in simple logical loop showed in Fig. 2.8.

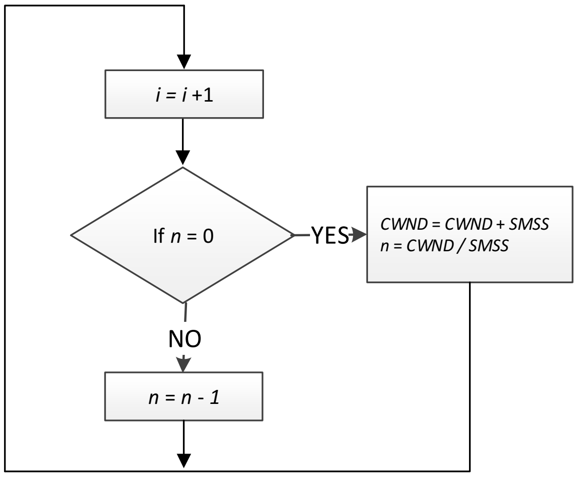


Fig. 2.8. Congestion windows calculation based on received acknowledgments

In Fig. 2.8 *i* is the received ACK packets, *n* is the number of acknowledged packets after last CWND increase.

To solve the problem and to find the number of ACK message count (due to fact that the CWND is increased based on received ACK) first the value of *i* must be found. From logical loop (see Fig. 2.8), which is expressed as number of sequence representing the *i* growth depending from CWND size (for simplicity the SMMS is set to one and CWND is set to zero), the number of ACK needed to get the CWND can be found, showed in table 2.1.

Table 2.1. Acknowledgment sequence numbers

W_{CWND}	1	2	3	4	5	6	7	8	9	10	11	12	13	14
ACK_{N}	2	5	9	14	20	27	35	44	54	65	77	90	104	119
Diff	3	4	5	6	7	8	9	10	11	12	13	14	15	16

So if the CWND is number of the sequence ACK_N, shown as subtraction field Diff. The sequence is quadratic sequence, which can be written as following function:

$$ACK_{\mathrm{N}} = \frac{W_{\mathrm{CWND}} \cdot \left(W_{\mathrm{CWND\_n}} + 1\right)}{2} - 1 . \tag{2.2}$$

Where ACK_N is the number of ACK messages must be send to reach CWND value, W_{CWND} is current TCP congestion window size in bytes, W_{CWND_n} is the congestion window size after ACK_{N} received ACK messages. The W_{CWND_n} valuecan easily found using following equation:

$$W_{\mathrm{CWND\_n}} = \frac{-3 + \sqrt{9 + 8 \cdot ACK_{\mathrm{N}}}}{2} . \tag{2.3}$$

By knowing the TCP receiving or advertised TCP window size value (W_{RWND})), which must be less or equal to W_{CWN}, the number of ACK messages can be found after which ACK rate limiting must start:

$$ACK_{N} = \frac{(W_{\mathrm{CWND}} + 2) \cdot \left(W_{\mathrm{CWND\_n}} + 1\right)}{2} - 1 . \tag{2.4}$$

In (2.4) defined above the *SMSS* 1, so in real world calculation should be replace the *CWND*, by division of *CWND* and *SMSS*:

$$ACK_{\mathrm{N}} = \frac{\left(\dfrac{W_{\mathrm{CWND}}}{SMSS} + 2\right) \cdot \left(\dfrac{W_{\mathrm{CWND}}}{SMSS} + 1\right)}{2} - 1\,. \tag{2.5}$$

To reduce the load to CPU the (2.5) can be optimized of efficiency by eliminating division operation, which is more CPU demanding compared to multiplication or sum and allow the system performance the operation faster:

$$ACK_{\mathrm{N}} = \frac{\left(\dfrac{3W^{3}{}_{\mathrm{CWND}}}{2SMSS}\right)}{2}\,. \tag{2.6}$$

But as the variable calculation is carried out on after TCP window update, in reality it will not have any affect to system performance.

One significant thing is that the equations for calculating the ACK_{N} representing sequence value is based on the logical loop showed in Fig. 2.8. But in RFC documentation the defined equation responsible for congesting window is increased by one if delta increase is less than one:

$$1 < \frac{SMSS^{2}}{W_{\mathrm{CWND}}}\,. \tag{2.7}$$

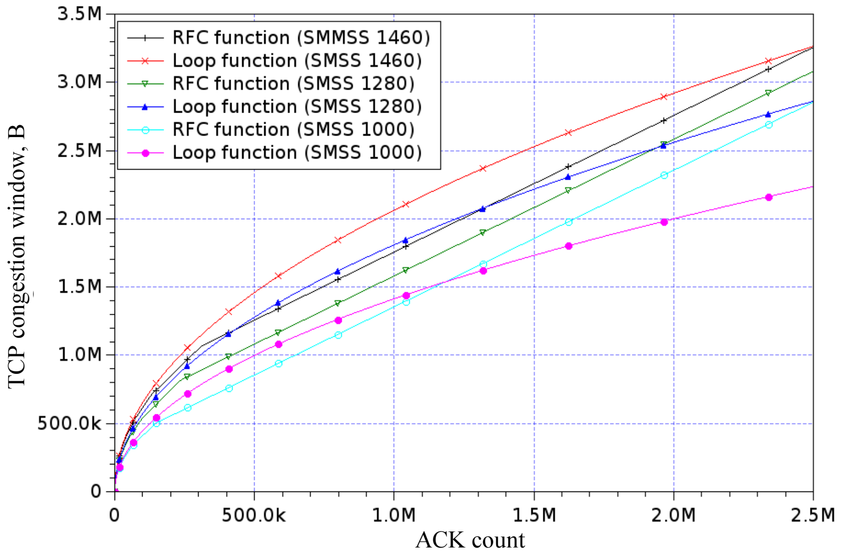


Fig. 2.9. Proposed congestion window growth comparison with different sender maximum segment size values

This allowing faster function growth with higher ACK_N value. The difference between the RFC define and proposed in (2.6) and is showed Fig. 2.6.

From the results of (2.6) and RFC functions showed in Fig. 2.9 the function growth of (2.6) is more aggressive with lower N values, but the N increases the W_{CWND} increase slows downs. This is due to fact that the delta increase of RFC defined equation is replaced with constantan after limit value (2.7) is reached, the RFC function becomes liner function. This allows the RFC function to growth faster, compared to the loop function. Also a big impact to the functions is the *SMSS* value, the less the value, the slower is the growing of the function and the RFC function start to overcome the loop function earlier.

Although the RFC growth function overcomes the loop function in some point but due to high value of ACK_{N} or the sequence number the TCP CWND windows mostly will be fully opened. So if the TCP sending side is using default RFC suggested congestion avoidance growth algorithm for CWND increase the start of ACK rate limiting should not have any impact to TCP session CWND growth. And more, during the proposed calculation, the slow start phase was eliminated.

2.5. Upper Limit of Acknowledgment Rate Limiting

The second problem with ACK rate limiting is that the client receiving less acknowledgments will reduce the sending speed of TCP data segments, if CWND is near or less then *BDP* (bandwidth delay product). It happens due to fact that the sender can send more data only after CWND window is freed (Nagle 1984). Like in rate limiting *_tcp_ack_snd_check* function the receiving data rate can be modified by reducing ACK sending rate *__tcp_slect_widow(sk) >= tp– >rcv_wnd* in condition then system is overloaded with received data and cannot clean up the receive buffer fast enough (Seth *et al.* 2008). In such case by reducing ACK message rate it should reduce the TCP data sending rate to the server. To overcome this problem, the following condition must be met. The RWND variable *tp– >rcv_wnd* (which is advertise to CWND) and maximum system allowed CWND must be large enough to store more than BDP (bandwidth delay product) of data in CNWD. Both TCP windows (CWND, RWND) must be monitored, due to fact that the RWND is advertised to the client and from RWND value the maximum allowed CWND window is set. Also the CWND also depends from system settings and it is allowed maximum CWND value. In some cases this value can be less than advertised RWND window. In addition to performance degradation TCP can face jitter problem, then client starts sending data in more jittering way. This can occur when the CWND size is smaller than the BDP value and ACK rate reduction can cause this behaviour. Also the problem the jitter problem can occur also in the sending system due to heavy system load NIC settings (TSO, GSO, and GRO) or system workload.

In case the *RWND* and CWND is big enough and RTT time is low the ACK sending rate can be reduced significantly without losing performance or seeing receive traffic jittering. But if the ACK limiting value *ack_rate_val* will not stop growing and continue to increase. At some point the same problem of performance and jittering will be encounter, and the increase of RWND and CWND will be needed in the system.

The ACK limiting rate reduction will stop (will grow more slowly) in some point due to fact the main function (logical condition) has additional conditions build in, which will delay sending TCP acknowledgment for some time period (but not discard it). The *tcp_send_delayed_ack(sk)* function based on RTT will calculate the timeout value for sending the TCP acknowledgment. If the time is close to expire, the ACK message must be sent. After the new value of timeout will be recalculated based on change conditions of the network (even if the ACK limiting rate value was reduced). It will surpass/eliminate *icsk_ack.rcv_mss* (the first logical condition) value and send ACK message immediately, by not allowing to reduce the ACK rate to minimum (Seth *et al.* 2008; Bhuiyan *et al.* 2009).

By making the ACK limiting value too high will lead to degradation of performance and increase jittering. The main sign of too high ACK rate limiting value is the reduction of throughput. If TCP data throughput is depredating after ACK limit increase, it must be considered that the ACK limiting algorithm has reached the maximum value. The second condition which can indicate the maximum ACK limiting value is the jitter increase in RTT measurements. But as the jitter is harder to measure and it also depends from network and sender side conditions also, it should not be used to indicate the upper limit of ACK limiting. The jitter variable can be used in conjunction with throughput variable for more accurate detection of upper limit of ACK limiting. In the dissertation only the change of TCP data throughput is considered an indicator for to high ACK limiting value.

Based on *BDP* (2.8), the needed RWND value for the sender to have can be calculated (*tp– >rcv_wnd*), the only difference is that the server (TCP ACK sender) will reduce the ACK rate and additional delta time to RTT must be included. To do this the default equation (2.8) must by modifying with additional variables for *BDP* or RWND calculation of W_{RWND}.

$$BDP = T_{\mathrm{RTT}} \cdot BW \ . \tag{2.8}$$

$$T_{\mathrm{RTT}} = 2T_{\mathrm{d}} + T_{\mathrm{p}} + 2T_{\mathrm{CPU}} \ . \tag{2.9}$$

$$BDP \cong \left(2T_{\mathrm{d}} + T_{\mathrm{p}}\right) \cdot BW = \left(2T_{\mathrm{d}} + \frac{SMSS \cdot N}{BW}\right) \cdot BW = 2T_{\mathrm{d}} \cdot BW + SMSS \cdot N. \tag{2.10}$$

Where T_{RTT} is the collective round trip time, T_{d} – delay of the network, T_{p} – time needed to send the packet to the network. T_{p} – is calculated by dividing the *SMSS* by *BW*. T_{CPU} – time needed to process the packet (see Fig. 2.10).

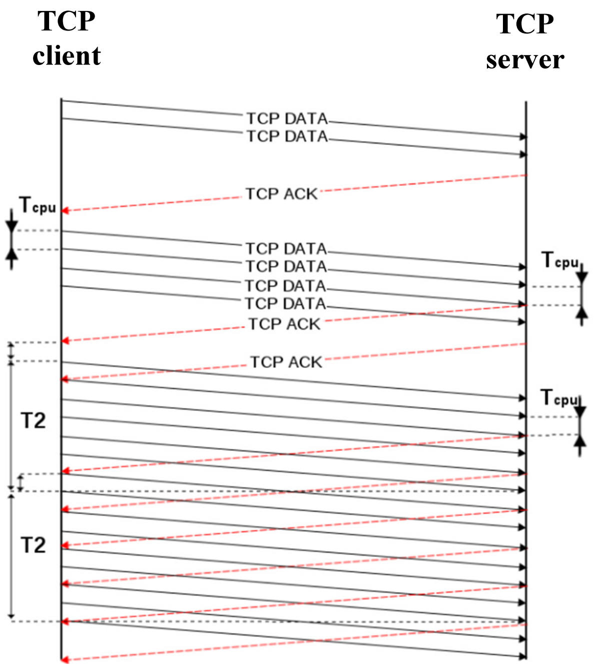


Fig. 2.10. Transmission control protocol messages processing delay in network and remote system

In modern system with high CPU power the T_{CPU} is very small compare to network delay and message sending time and can be eliminated from our equations. So basically the *BDP* is linearly function which is connected to ACK message rate. The issue with real world TCP session is that the real value of T_{RTT} can deviate due to T_{CPU} value or due to changing conditions of the data transmission network. By leaving the RWND variable W_{RWND} undefined, even if the link utilization was full at used W_{RWND} value, the system would not know if the ACK message reduction is the cause TCP session to stop.

So in real world the system can't calculate the needed RWND variable W_{RWND}, but by knowing if the link fully utilized (physical link speed is reached, or the sending system overloaded) the system can reduce the ACK message rate and see if it has affect to TCP performance.

The simplest way to measure data receiving rate to calculate the received TCP data segments in time period. The delta time for measurement must be at least two or more times of advertised *RWND* (or even more to allow RTO and the *tcp_send_delayed_ack* function to recalculate the timeout values).

If the packet rate/count does not change during delta time period conclusion can be made that the ACK message rate reduction did not impact the sending speed of TCP data message rate (by comparing the new value with old one). If it does the system must fall back to old value, and put this value as new minimum for ACK message rate (until new W_{RWND} is advertised or network condition changes).

The testing and results will be made in the third chapter of the dissertations.

2.6. Linux Kernel Transmission Control Protocol Stack Modification for Acknowledgment Limiting

In this section we will show and explain the modification change in Linux kernel and how it will work. The basic concept of ACK message rate limiting was explained before, and the rate of ACK can be changing easily by increasing the *icsk_ack.rcv_mss* value.

Default code of Linux kernel is show in Fig. 2.11. The code is modified by adding additional variable tp–> tcp_ack_rate, seen in Fig. 2.12.

net/ipv4/tcp_input.c

```
static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
{
        struct tcp_sock *tp = tcp_sk(sk);

            /* More than one full frame received... */
               if  (((tp->rcv_nxt  -  tp->rcv_wup)  >  inet_csk(sk)-
>icsk_ack.rcv_mss &&
```

Fig. 2.11. Linux kernel default tcp_input.c source code of *__tcp_ack_snd_check* function

net/ipv4/tcp_input.c

```
static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
{
        struct tcp_sock *tp = tcp_sk(sk);

            /* More than one full frame received... */
               if  (((tp->rcv_nxt  -  tp->rcv_wup)  >  inet_csk(sk)-
>icsk_ack.rcv_mss * tp-> tcp_ack_rate &&
             /* ... and right edge of window advances far enough.
```

Fig. 2.12. Linux kernel modified tcp_input.c source code of *__tcp_ack_snd_check* function

The *tcp_ack_rate* variable defines the needed number of TCP data segments which must be received before allowing to send ACK message. In default or initial state of TCP session the value is equal to 1, so the number of ACK must be at least two. If no TCP data packet drop occur during period of time (or after receiving *WIN/SMSS* TCP data packet) the value can be increased.

The *tcp_ack_rate* value, as it was define before must not only be activated after the client side CWND fully opened, but do impact other TCP functions like fast retransmit or recovery after packet loss. To do so the system must also track if any packet loss did not occurred.

The ACK rate limiting implementation is showed in flow graph (Fig 2.13).

To existing Linux kernel code additional TCP state variables are added for TCP sessions tracking. After packet drop or after receiving out of order TCP data segments the ACK limiting values must be reset and TCP session must be allowed the recover after packet drop, even if one segment is lost the ACK limiting algorithm must be disable and fast retransmit is allowed restore TCP session flow.

First the system must reset the *tp–>init_pkt_cnt* variable which is used in the initial stage of TCP session, it is compared to *tp–>start_ack_lmt* which defines the end of the first stage.

During this phase system also defines the new maximum allowed ACK message rate or *tp–>ack_rate_max* variable. It is used stop the ACK rate limiting after reaching it and preventing of packet drop. It is set to one thirds of *tp–>rcv_wnd* value divided by receive MSS. In addition checking of new *tp–>ack_rate_max* is made to see if it is not more than 64 segments. Finally delta time values are resetted, the system uses them for time calculation needed to learn if the new *tp–>ack_rate_cnt* can be increased or not.

In second stage the algorithm looks if the TCP session is in the initial state or not, if so algorithm proceeds to third stage and ACK limiting algorithm starts, it is define in Fig. 2.14 in the first else statement of C code.

In the third stage the function goes in the loop, increasing the *tp– >ack_pkt_cnt* (increased in end third stage) after every TCP segment is received. If the new *tp–>ack_pkt_cnt* value is bigger the *tp–>start_ack_lmt2* value. The algorithm has to reach the end of the loop and the decision of new ACK limit *tp–>ack_rate_val* must be made. It is done only after knowing if the old value the algorithm used had increased the TCP throughput or did it declaimed. It's done by calculating delta time needed for second stage loop to process *tp–>start_ack_lmt2* TCP data segments. If the value is smaller the algorithm increases the *tp–>ack_rate_val*, else the value must be reduced

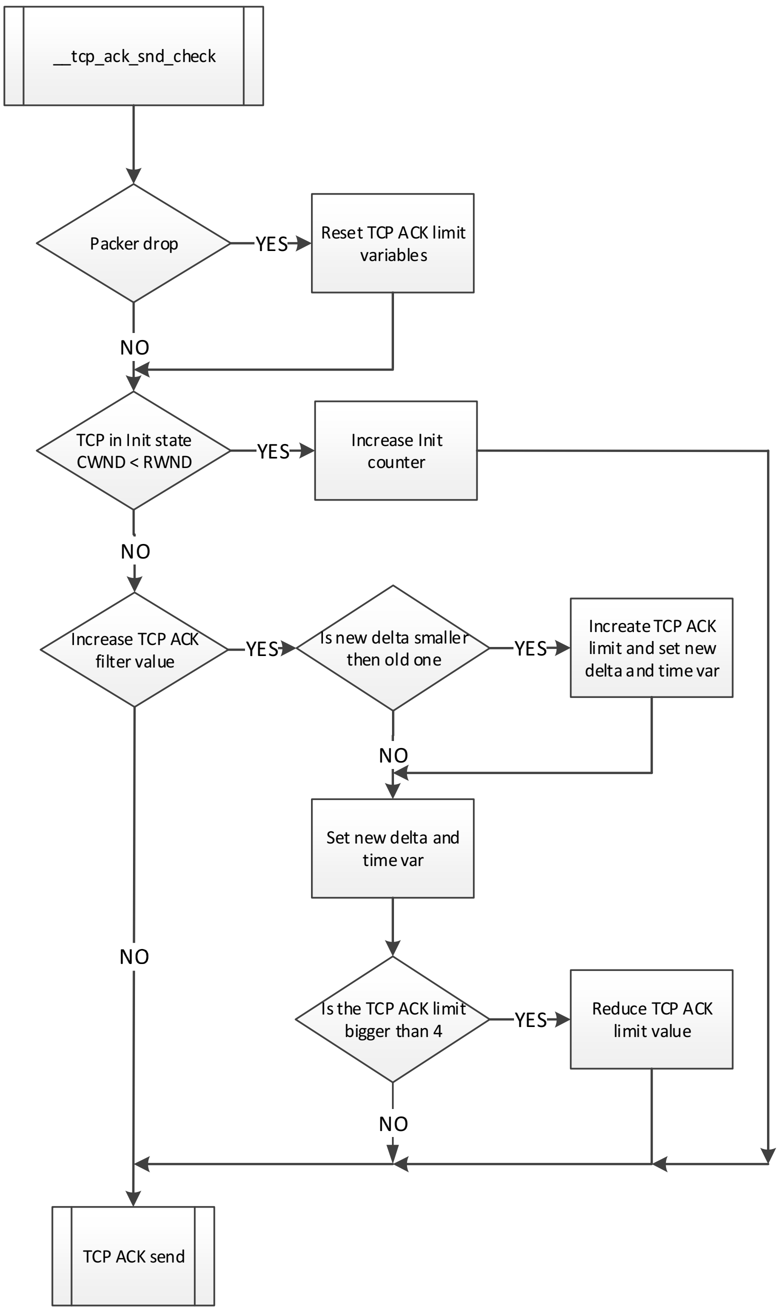

__tcp_ack_snd_check
Packer drop
YES
Reset TCP ACK limit variables
NO
TCP in Init state CWND < RWND
YES
Increase Init counter
NO
Increase TCP ACK filter value
YES
Is new delta smaller then old one
YES
Increate TCP ACK limit and set new delta and time var
NO
Set new delta and time var
NO
Is the TCP ACK limit bigger than 4
YES
Reduce TCP ACK limit value
NO
TCP ACK send

Fig. 2.13. Acknowledgment limit algorithm flow graph for Linux kernel

net/ipv4/tcp_input.c

```
static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
{
struct tcp_sock *tp = tcp_sk(sk);

if ( ofo_possible && skb_peek(&tp->out_of_order_queue) ) {
  tp->init_pkt_cnt = 0;
  tp->ack_pkt_cnt = 0;
  tp->ack_time_t1 = 0;
  tp->ack_rate_val = 0;
tp->ack_rate_max=tp->rcv_wnd/(3*inet_csk(sk)->icsk_ack.rcv_mss);
  if (tp->ack_rate_max > 64 ) {
  tp->ack_rate_max = 64;}
  tp->ack_time_t2 = 0;
  tp->ack_time_delta = 0;
  tp->tcp_delayed_snd = 0;
}
if ( tp->init_pkt_cnt < tp->start_ack_lmt ) {
  tp->init_pkt_cnt += 1 ;
} else {
  if (tp->ack_pkt_cnt > tp->start_ack_lmt2 ) {
    tp->ack_pkt_cnt = 0;
    tp->ack_time_t2 = jiffies;
   if (tp->ack_time_t2 - tp->ack_time_t1 <= tp->ack_time_delta  ||
tp->ack_time_t1 == 0 ) {
    tp->ack_rate_val+=1;
    tp->ack_time_delta = tp->ack_time_t2 - tp->ack_time_t1;
    tp->ack_time_t1 = tp->ack_time_t2 ;
    } else {
      tp->ack_time_delta = tp->ack_time_t2 - tp->ack_time_t1;
      tp->ack_time_t1 = tp->ack_time_t2;
      if ( tp->ack_rate_val > 4) {
        tp->ack_rate_val-=1;
 }}}
tp->ack_pkt_cnt +=1;
}
```

Fig. 2.14. Linux kernel modified tcp_input.c source code of __tcp_ack_snd_check function for checking transmission control protocol state

If new delta time value is smaller than before the algorithm must to reduce the ACK limit value *tp–>ack_rate_val* once more, but additionally the new value must not be smaller than the initial value. If system does not check it, the variable could go end at zero and start generating ACK messages for every received data segment. The upper limit of ACK limit algorithm is defined in variable *tp–>start_ack_lmt* which is calculate via initial *tcp_ack_limit_init* function showed in Fig 2.15.

For *proc* directory kernel variable were created in *sysctl_net_ipv4.c* file (see Fig. 2.16), which can be changed during runtime, by entering new ACK limit value.

include/net/tcp.h

```
static inline int tcp_ack_limit_init(const struct sock *sk)
{
      if ( sysctl_tcp_ack_limit > 1 ) {
              return sysctl_tcp_ack_limit;
      } else {
      return ((rwind/ad_mss + 2)*(rwind/ad_mss + 1))/2 -1;
 }
```

Fig. 2.15. Linux kernel tcp.h source code addition for new *tcp_ack_limit_init* function changes

net/ipv4/sysctl_net_ipv4.c

```
{
 .procname     = "tcp_ack_limit",
 .data         = &sysctl_tcp_ack_limit,
 .maxlen       = sizeof(int),
 .mode         = 0644,
 .proc_handler = proc_dointvec
},
```

Fig. 2.16. Linux kernel tcp.h source code addition for new *proc* file variable *tcp_ack_limit*

The defined source code (see 2.16) allow to change the upper and lower limits of ACK rate limiting via the tp–>start_ack_lmt and tp–>ack_pkt_cnt variables, without going to int_tcp_ack_limit_init function. The function calculates the worst case in which TCP CWND can grow and also the longest period of time will be need to get the first and second stage loops go pass. The new variables can be passed, modified or disabled in Linux OS kernel in real time via Linux pseudo file system proc, without modifying Linux kernel source.

2.7. Conclusions of Chapter 2

1. A new proposed method, for calculating the TCP congestion window size on the remote node, can be used to find the ACK message packet count, which is needed to be pass before ACK rate limiting can start. This allow a new TCP session to compete with other TCP data stream and get into congestion avoidance phase.
2. The new method was proposed and explained for calculating the upper ACK rate limiting value, for this the TCP inter packet delay is used. Proposed method allow dynamically adjusting the upper limit of ACK rate limiting based on the receiving speed of data packets on the TCP receiver side.

3. A new proposed algorithm can dynamically monitor and adjust ACK limiting values based on current TCP session status, network or system conditions. It allow dynamical back off and recovery after TCP session packet drops or session restart. This algorithm can be easily implemented and used in current Linux OS kernel without any major impact to the system stability and performance.

3

Experimental Investigation of Acknowledgment Limiting in Heterogeneous Networks

To evaluate the TCP changes made in Linux OS kernel code test were conducted. The results from testing were compared with the default TCP behaver of default Linux OS kernel. In addition to this an investigation of the new code and its impact to system load and system stability were made. For this several test setups were made to see how the Linux OS kernel patch effect whole system and TCP and TCP stack. It is hilly important to note that most of TCP code are related with other kernel code and changes in one part of Linux OS kernel can effect hole system stability.

The research results are published in author publication: Pavilanskas *et al.* 2010

3.1. Experimental Investigation of Acknowledgment Limiting on Virtual Machine

Before the impact of ACK limiting to TCP throughput and system load can be estimated, the testing of modified system must be made.

To reduce the impact of hardware and reduce testing time all the test were done using virtual machines (running on KVM virtualization program) with default configuration set (no major changes to image or its components were not made). For testing an open source OpenWRT Linux distribution was used. The distribution is mainly used in low power and embedded devices (Dutt *et al.* 2012; Maxim *et al.* 2012). Also all OpenWRT image (including Linux kennel and software packages) are made during compilation process allowing use easily export OpenWRT images to other architecture (automatically including needed modules and packages).

For virtualization x86 architecture with one dedicated CPU was selected and 512 MB RAM was allocation for OpenWRT system. The only significant change was NIC device, which was changed from default NIC device virtio, which use packet segmentation in driver. It was changed to Intel e1000 device which was connected to physical server NIC via Network Bridge. Also on KVM server and remote side TCP NIC offloading features were turned off (Zhang *et al.* 2010; Tafa *et al.* 2011; Rathore *et al.* 2013). The full configuration setup with connected device is shown in Fig. 3.1.

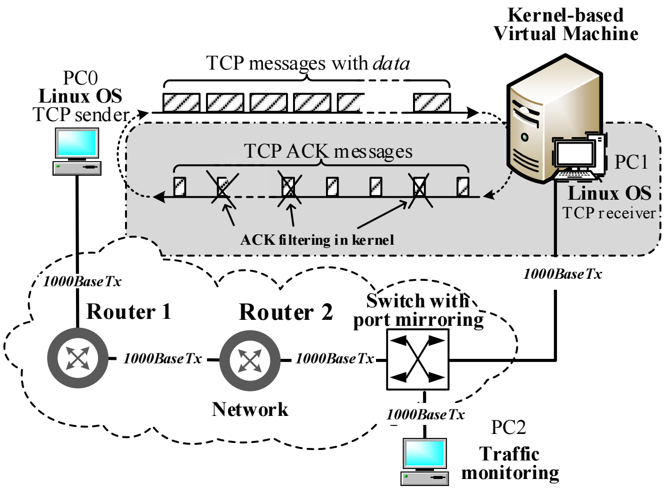


Fig. 3.1. Acknowledgment limiting test setup running Linux in virtual environment

The testing network was made from three PC's connected via copper 1 Gbps Ethernet link, for TCP packet capturing and analysing dedicated PC was used. All hardware NIC TCP offloading features will were turned off for more accurate measurements. This is done due to fact that in most cases this features can and will influencing the TCP behaviour. The only limiting factor or bottleneck in the test setup was the KVM server CPU computing power.

As it was explained before, the new kernel TCP stack modification consists of two main parts. The first part of code is responsible for ACK limiting values, which are changing during the runtime depending from TCP packet flow per delta time. And the second one is deciding if ACK message must be send. It's important to note that ACK sending can be invoked without reaching the defined ACK limiting value after TCP acknowledgment timeout value expiration or etc.

To evaluate the code and the calculated variables the ACK limiting algorithm was outputted to Linux OS kernel system logging facility <*dmsg*>. For this additional Linux kernel printing functions were added. By printing the existing Linux kernel variables of maximum allowed ACK limit value allowed for better TCP sessions tracking in time or after packet drops occurs.

As shown in Fig. 3.2 ACK rate value starts to increase at ~10 s, values changes over the time, from zero on the session start up to 20–25 s. At ~30 s. A first reduction off *ack_rate_val* is seen, indicating that the ACK limiting code is working. The same reduction of *ack_rate_val* value is seen at ~ 50 s and ~ 80 s, indicating that the TCP sending speed in decreased in all cases. The results were collected via <*dmesg*> application. Also TCP and ACK messages rates were monitored with <*wireshark*> to see the impact of ACK filtering algorithm.

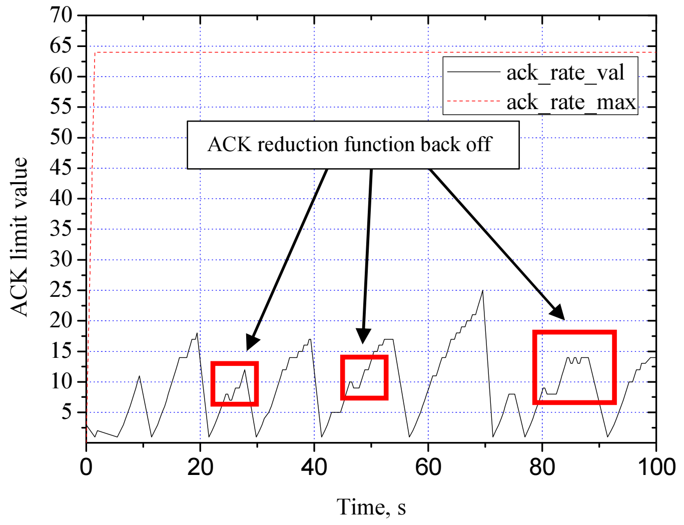


Fig. 3.2. Linux kernel *ack_rate_max* and *ack_rate_val* during transmission control protocol data transmission

During the testing smaller *ack_pkt_cnt* value was used, which correspond to the number of TCP segment must be passed to allow the increase of ACK limiting value. This makes the ACK limiting algorithm to behave more aggressively, by increasing the value more frequently to see the impact of it to TCP stability in short TCP sessions.

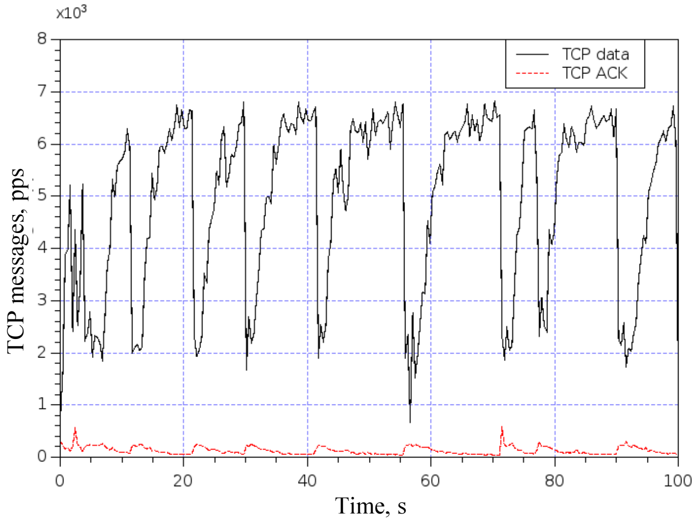


Fig. 3.3. Transmission control protocol throughput running default kernel

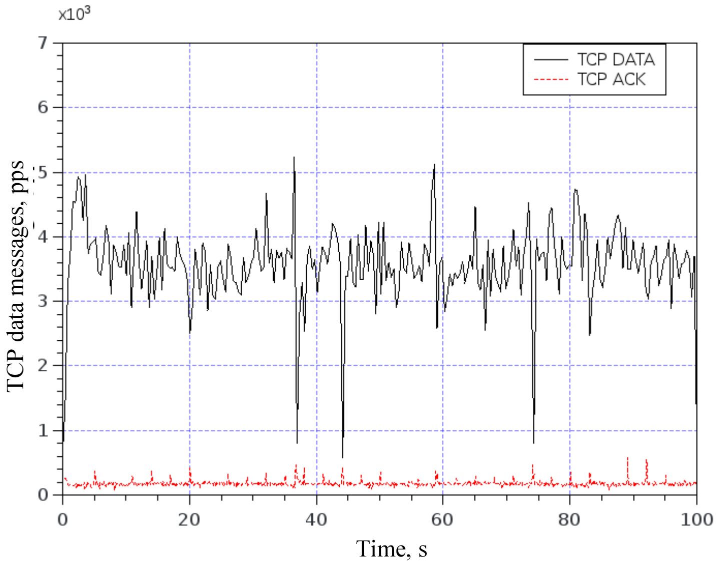


Fig. 3.4. Transmission control protocol performance with modified Linux kernel

To compare the ACK limiting algorithm with default TCP behaver an identical OpenWRT image was compiled without the Linux kernel modification. All the test were performed on the same system only by changing the virtual machines images. The network traffic results we collected and analysed using <*wireshark*> application on dedicated network monitoring PC (Fig. 3.1).

To test the TCP throughput <*iperf*> application was used. TCP traffic was generated at maximum speed for 100 s. The same tests were conducted on both system in identical network configuration.

The results of both systems were compared and analysed in <*wireshark*> application to see the impact of ACK filtering. The TCP packet per second rate or throughput are showed in Fig. 3.3 and 3.4, and the TCP RTT measurement of modified and default Linux kernel in Fig. 3.5 and 3.6.

In TCP throughput results seen in Fig. 3.3 and Fig. 3.4, the modified TCP stack is performing much better compared to default system in virtual environment. In average the TCP data throughput is up to 30% higher with ACK filtering enabled. Also the modified systems has much smaller RTT value compared to default system. The TCP RTT value is reduce almost twice compared to default system (Fig. 3.5 and 3.6).

By looking to Fig. 3.4 a big slump of throughput every 10–20 s can be seen, it correlates with TCP RTT results in Fig. 3.6 and is caused by high ACK rate limiting value. This can be controlled be reducing the growth function of ACK rate limiting algorithm. For experimental setup a faster *ack_rate_val* growing functions was used. It increased the ACK limiting value to obtain needed network conditions and get needed results in more hostile network environments.

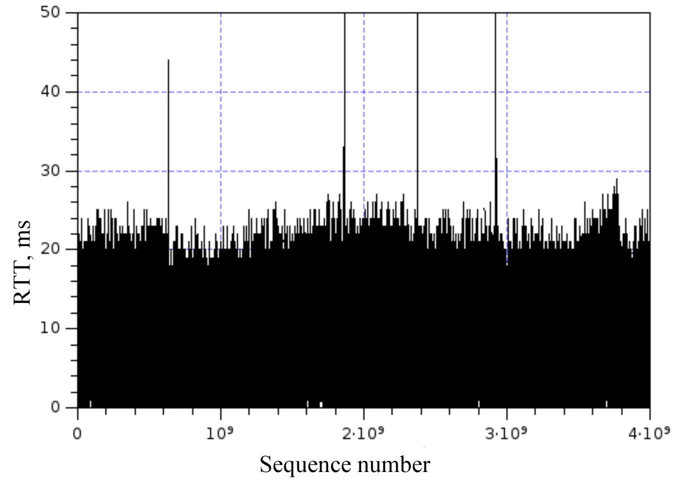


Fig. 3.5. Round trip time deviation during Transmission Control Protocol data transmission with default Linux kernel

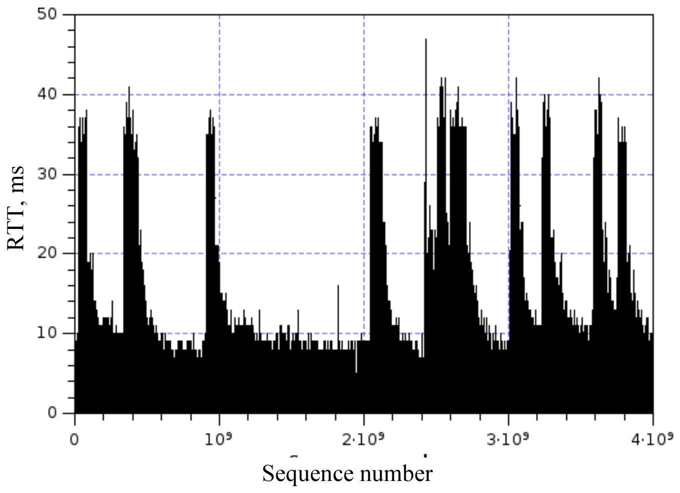


Fig. 3.6. Round trip time (RTT) deviation during Transmission Control Protocol data transmission with modified Linux kernel with small *ack_pkt_cnt* value

From results of TCP RTT seen in Fig. 3.5 and 3.6, a round trip time histograms was generated (Fig. 3.7 and 3.8). It showed a big shift of RTT values to the left (lower values). In default system configuration (system running default Linux OS kernel) a much higher RTT values are seen, and average values are distributed between 10–20 ms.

In modified Linux OS kernel, the RTT histogram results are distribution between the 1–10 ms, which is much smaller and can have substantial effect to multimedia and real time application. Also the periodic RTT increase (peaks) seen in seen in the Fig. 3.6 (modified Linux OS kernel) happened due to high CPU utilization and it correlates with the peaks of TCP throughput Fig. 3.4. By making the ACK limiting more conservative the system could reduce the TCP RTT even more and sustain the RTT distribution in lower values for longer time could be seen.

It is important to notice that even in default Linux OS the ACK message rate (Fig. 3.3) is much smaller than it should be according the RFC 1122 and RFC 5681 ("*should be generated for every second full size segment*"). The ACK message rate is almost equal to system with ACK rate limiting enabled. It happened due to the fact that the system was under the heavy CPU load (due to high TCP data message rate). To reduce the load the kernel function reduces the ACK message rate, and this reduces the TCP throughput and TCP data and ACK message rates.

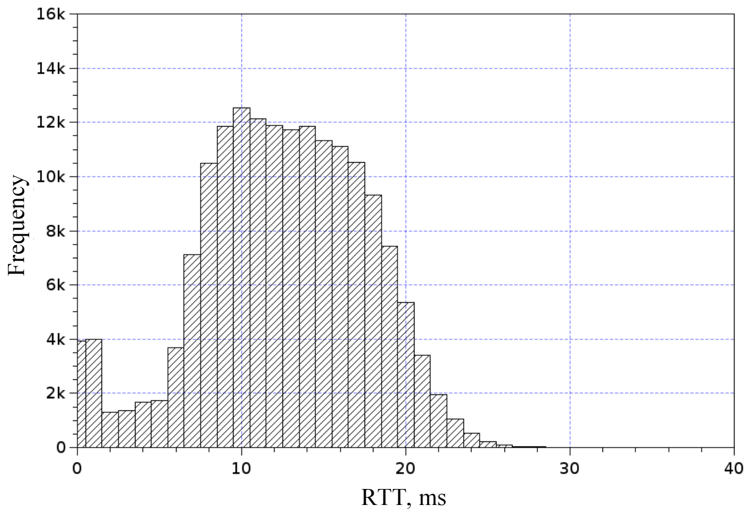


Fig. 3.7. Round trip time histogram running default Linux kernel for 100 s file transfer

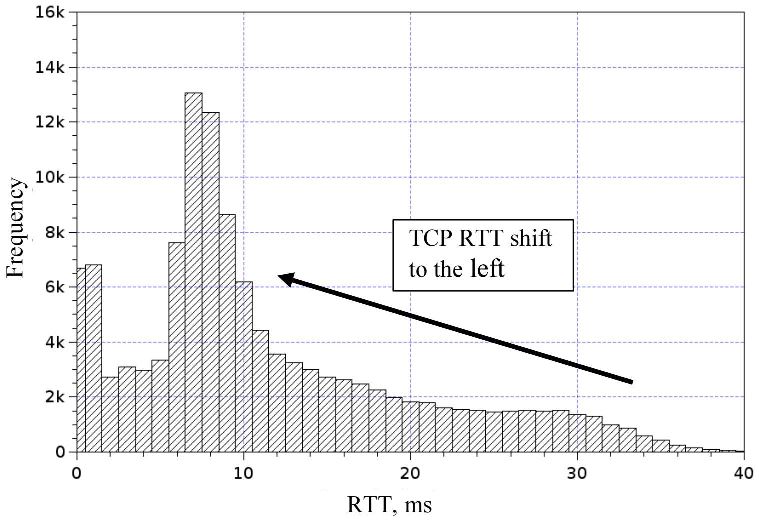


Fig. 3.8. Round trip time histogram running modified Linux kernel for 100 s file transfer

During the testing both system had identical system resources and the only significant difference was ACK message rate limiting function in Linux OS kernel, as seen from the results, the new modified TCP kernel stack performance much better compared to default kernel system. Even in high CPU load the ACK rate limiting function can reduce the ACK rate more than half. By including the ACK limiting code to Linux OS kernel the system can not only reduce the ACK message rate but also skip unneeded kernel function checking and further processing of ACK messages (Fig. 3.9).

net/ipv4/tcp_input.c

```
4796 static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
….
….
4801                 if  (((tp->rcv_nxt  -  tp->rcv_wup)  >  inet_csk(sk)-
>icsk_ack.rcv_mss * tp-> tcp_ack_rate &&
```

Fig. 3.9. Source code of modified Linux kernel *tcp_input.c* file

From the tests made on KVM platform it is clearly seen that the new TCP Linux kernel patch helps to reduce ACK rate so it allows the system under heavy load achieve much better performance in TCP data messages per second throughput and RTT values.

In addition the ACK message rate reduction also reduces the load to the TCP sender, as the end system will receive less ACK messages from the receiver and less CPU power will be needed to process them.

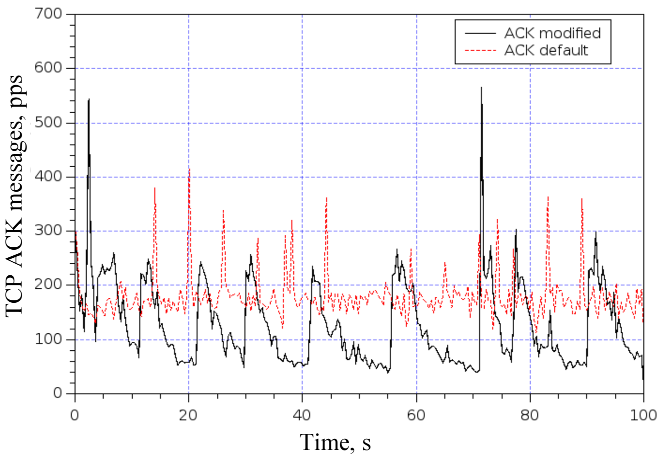


Fig. 3.10. Acknowledgment messages rate comparison between default and modified Linux kernel with small *ack_pkt_cnt* value

Both system TCP throughput and ACK message rates results displayed in Fig. 3.11 and 3.10. The system with ACK limiting enabled has much better TCP throughput, in average ~ 30% increase is seen on modified system compared to default one. The performance spikes can get up to 40% for short period of time.

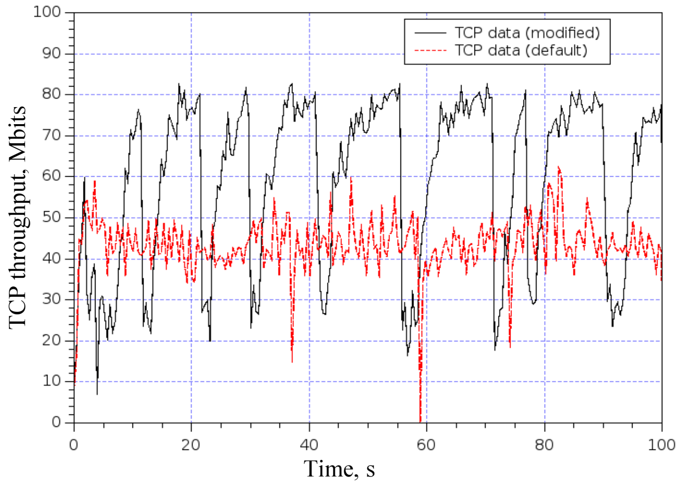


Fig. 3.11. Transmission control protocol throughput in Mbits comparison between default and modified Linux kernel with small *ack_pkt_cnt* value

The changes in TCP throughput and RTT (see Fig. 3.3, 3.6 and 3.11) in modified system happened more dynamically, and more rapidly during the test time. The results of throughput and PPS correlates with ACK limiting value (seen in Fig. 3.10). By analysing the RTT and RTO values it Linux OS kernel, it was found that the drops in TCP throughput flow happed due to t_{RTO} time is expiration. By reducing the RTT time the system reduces the t_{RTO} value, and increase the possibility of TCP segment retransmission if the sender does not receive ACK message in t_{RTO} time.

The last ACK limiting test was made to see how reduction of ACK message rate affects the TCP sending side congestion window growth. By using <*tcp-probe*> in Linux OS kernel, the TCP sender side congestion window (CWND) and slow start threshold *(SSTRESH*) values and how it changes during the time and after packet drop. As in test before, all tests were performed on the same system with identical configuration. By default TCP Cubic congestion function was selected. As the TCP CWND is directly related to ACK message rate, after reduction of ACK messages rate a slowdown of growth of CWND window was expected. The results of CWND and SSTHRSH values are showed in Fig. 3.12 and 3.13.

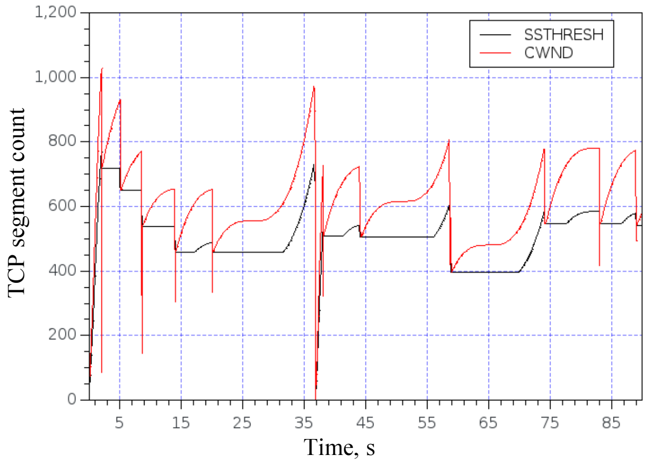


Fig. 3.12. Congestion window and slow start threshold function garth with default Linux kernel during data transmission

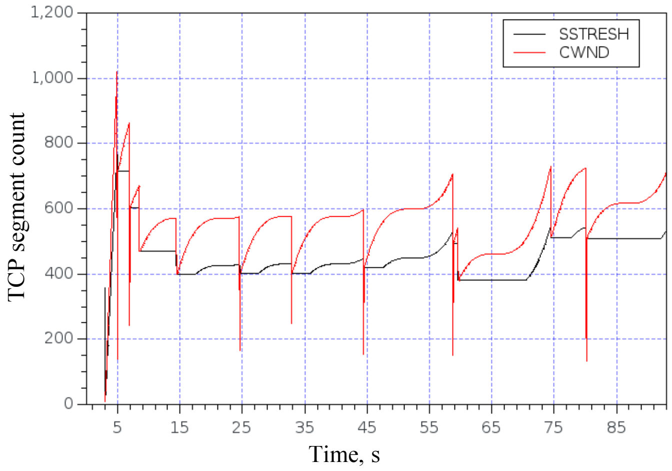


Fig. 3.13. Congestion window and slow start threshold function garth with modified Linux kernel during data transmission

In both cases (see Fig. 3.12 and 3.13) with and without ACK limiting, the TCP congestion window growth and recovery functions perform similarly and no big difference in TCP Cubic function is seen. Most importantly the Cubic function behaves the same like in congestion avoidance phase, which is very important to

TCP recovery process. ACK limiting does change TCP session behaver or growth functions (see Fig. 3.13) and allows TCP session to compete for network resources with other TCP sessions. The only noncable difference was that modified system has slow start threshold (*SSTRESH*) value a little smaller and the CWND Cubic function in most cases does not enter exponential growth phase as in default system. It can be explained due to smaller RTT value which causes high TCP segment rate. So packet drop (t_{RTO} time expiration) happens in smaller *SSTRESH* and CWND values. From results seen in Fig. 3.13 with modified Linux kernel CWND value is more constant and small during the test time. This indicates that TCP sending speed is more constant and the RAM usage is more stable due to more stable system RTT value.

3.2. Acknowledgment Limiting on Physical Machine

As it was described in previous chapter, the TCP performance (RTT, throughput, etc.) is mostly impacted from hardware performance. To limit virtual system performance a CPU limitation was made, not allowing virtual system to consume all physical CPU. In real or physical servers more complicated conditions can arise. The physical server performance can depend from different system components and the bottleneck can be not only the CPU.

To test ACK limiting algorithm in in real environment the same OpenRTW image was used, only the build was made for MIPS based CPU architecture system. In our tests an Ubiquiti RouterStationPro router was used. It was running MIPS based architecture CPU and had limited amount of RAM of 128 MB. The image for testing was identical to one which was used in virtual system test (containing the same prebuild packets and system component) (see Fig. 3.14).

For comparing the impact of ACK limiting on system and TCP throughput two identical routers with OpenWRT images with ACK limiting enabled and without were used. In addition a separate network traffic capture PC for inspecting the communication traffic was used. The routers were connected via 1 Gbps links with NIC offloading disabled (which has big impact to TCP behaver).

The tests were conducted in TCP client to server scenario, when the client is sending the data to server, and comparing it with default TCP stack. The import note here is that bottle neck in such configuration is becoming the sending system or the TCP client, which is under the higher load compared to the server. In such scenario the TCP throughput gain by using ACK limiting is due to reduced load on the TCP sending system.

Like in tests before, the TCP throughput was tested using <*iperf*> application for TCP traffic generation for up to 100 s time period. The same tests were done on both system in identical network configuration.

The results of TCP packet per second and RTT are showed in Figs 3.15 and 3.16.

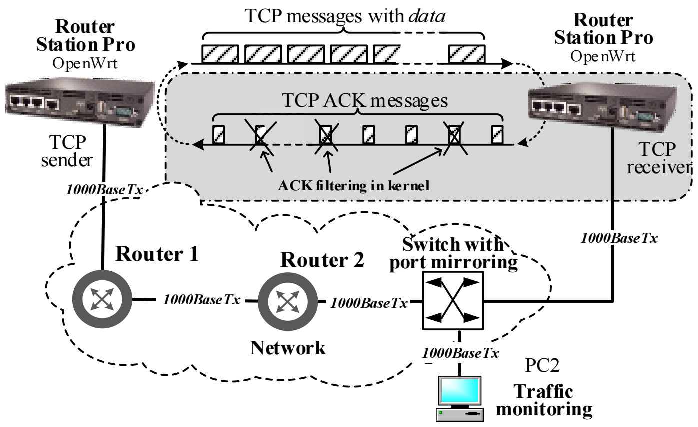


Fig. 3.14. Acknowledgment limiting test setup running Linux machine running on embedded RouterStatio Pro device

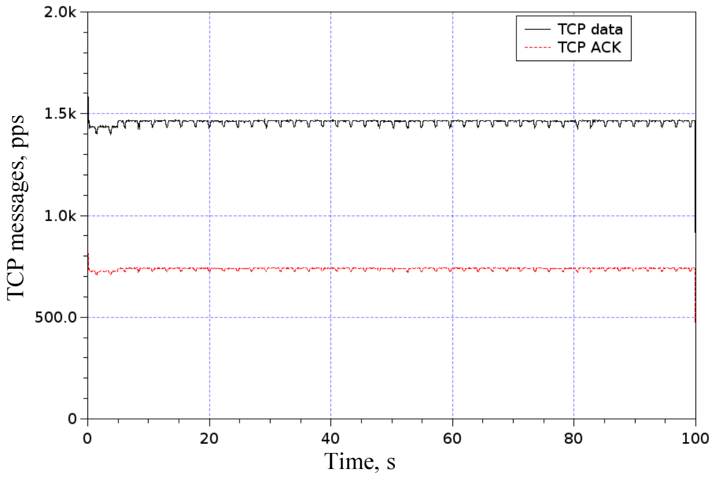


Fig. 3.15. Transmission control protocol data and acknowledgment packet rate in default system

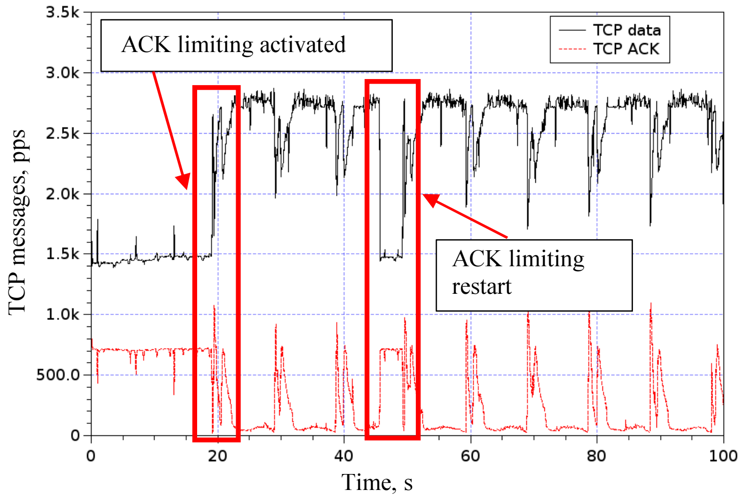


Fig. 3.16. Impact of acknowledgment limiting to transmission control protocol packet with small *ack_pkt_cnt* value

From results of Fig. 3.16 the TCP throughput starts increasing at 20 s from the start of the TCP session. At this point the ACK limiting starts limiting the outgoing ACK messages rate. As the upper limit of ACK limit was set to 2000 (the packet count after which ACK message rate must be reduced by one). The ACK rate goes down quit fast and stop after seeing the increase of RTT in TCP data stream (Fig. 3.16). At ~45 s ACK limiting stops due to multiple TCP data messages drops. The TCP sessions enters slow start phase and ACK limiting re-starts by disabling and resetting the ACK limiting algorithm values. After ~5 s at ~51 s ACK limiting starts again (Fig. 3.16) and grows up to the older value of *ack_rate_val*. The results were collected via <dmesg> application.

In out testing network an increase of TCP throughput can be observed up to 60% in average during 100 s data transfer compared to default TCP behaver. But in different configurations and systems the increase of TCP throughput can deviate due to many factors, like CPU, memory I/O, etc. performance. In most cases this value will deviate especially in system were TCP offloading is enabled and the sending and receiving nodes are in different configuration or hardware specifications.

By joining both TCP throughput graph (see Fig. 3.16 and Fig. 3.16) in one Fig. 3.17, it is clearly seen that TCP throughput of modified TCP stack is identical to default one at the start of TCP sessions and when a TCP packet drop occurred (at 45 s). This shows that the TCP modifications made to Linux kernel code does not impact the TCP performance and works if packet drop occur. The spikes and peeks are seen the Fig. 3.17 and Fig. 3.16 are due to smaller *ack_pkt_cnt* value used in testing environment for ACK limiting algorithm. This was used for faster ACK limiting function grow and to emulated situations with small TCP RWND used by the receiver.

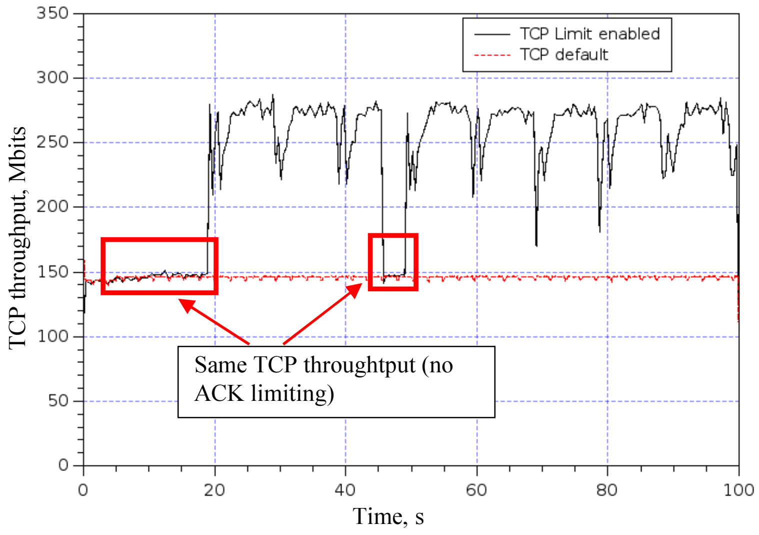


Fig. 3.17. Transmission control protocol data packet throughput comparison of default and modified Linux kernel

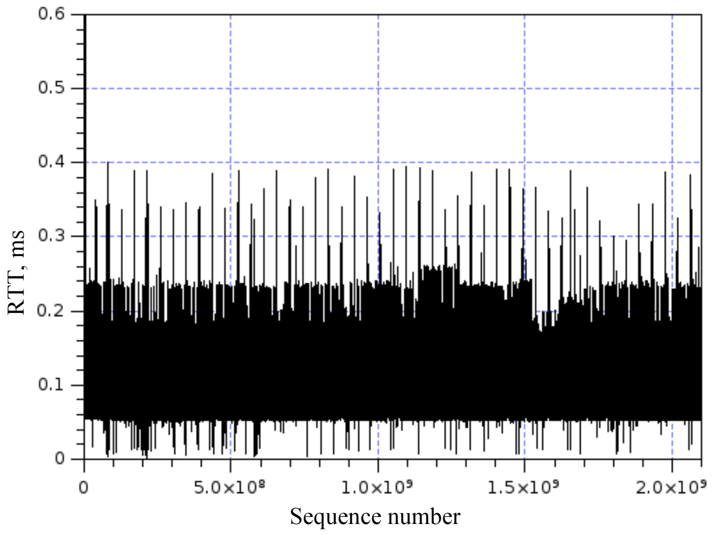


Fig. 3.18. Round trip time deviation during data transmission with default Linux kernel running on RouterStation Pro device

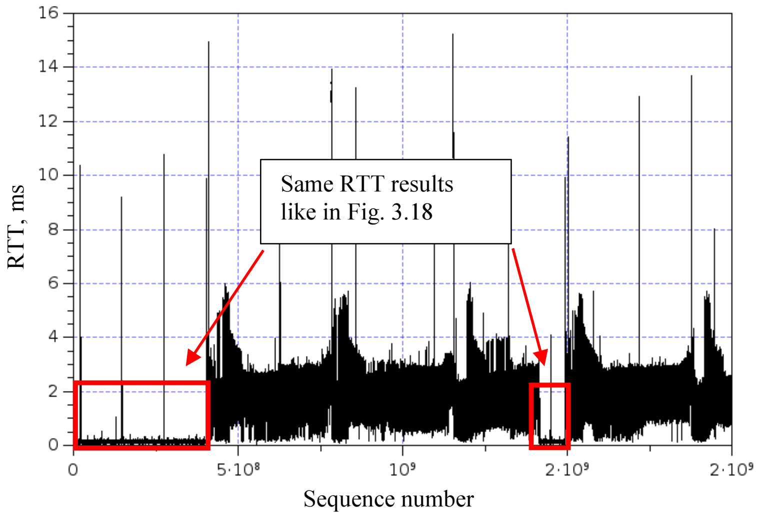


Fig. 3.19. Round trip time deviation during data transmission with modified Linux kernel running on RouterStation Pro device

By comparing TCP RTT results seen in Fig. 3.18 and 3.19, an increase of RTT is seen in modified systems after the ACK limiting algorithms is activated. The increase is seen at the point when ACK limiting start increasing, until a packet drop happens. After it the RTT decreases and is the same as in not modified TCP system (see Fig 3.19). This can be explained via differences that the bottle neck in this case is not the receiving end but the TCP sending end. So by reducing the ACK message rate the system increase the performance not for the TCP receiver but for TCP sender.

To eliminate the impact, of possible slow TCP sender (due to low efficiency CPU node) a modern PC with higher CPU efficacy was used in this testing's. The new TCP sender's performance was much higher compared to RouterStatioPro embedded device. The test setup looked the same like in the test before in Fig. 3.14 only the TCP sending node was a PC having Intel 2 GHz CPU with 3 GB of RAM.

In this testing scenario the TCP sending node was PC, which was able to send or receive TCP data up to 1Git/s to embedded RouterStationPro Pro device, due too much faster CPU. For TCP data generation a Linux testing application <*ip-erf*> application was used.

By comparing testing results of default and modified systems, the TCP throughput (in Fig. 3.21) of default Linux kernel stack with default TCP implementation is performing a little better and TCP data throughput is higher compared with modified TCP stack.

These testing results can be explained by looking to ACK message rates of default and modified TCP Fig. 3.22. In testing the ACK message rate changes in both TCP sessions over the testing time and the ACK message rate is not one half of TCP data PPS rate according the RFC documentations (Berkeley *et al.* 2011; Bott 2014). This happens due the reduced receive window size RWND or to be more correct due to heavy system load, as it was described in previous chapter in TCP acknowledgment generation and SWS part (Clark 1982).

net/ipv4/tcp_input.c

```
4796 static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
…..
….
4805                __tcp_select_window(sk) >= tp->rcv_wnd) ||
4806               /* We ACK each frame or... */
4816 }
```

Fig. 3.20. Linux kernel acknowledgment messages sending source code

In condition then system is overloaded with received data and cannot clean up the receive buffer fast enough. In such case by reducing ACK message rate will reduce the data sending rate to the server. It is controlled by *__tcp_ack_snd_check* function defined in *tcp_input.c* file (see Fig. 3.20 line 4805).

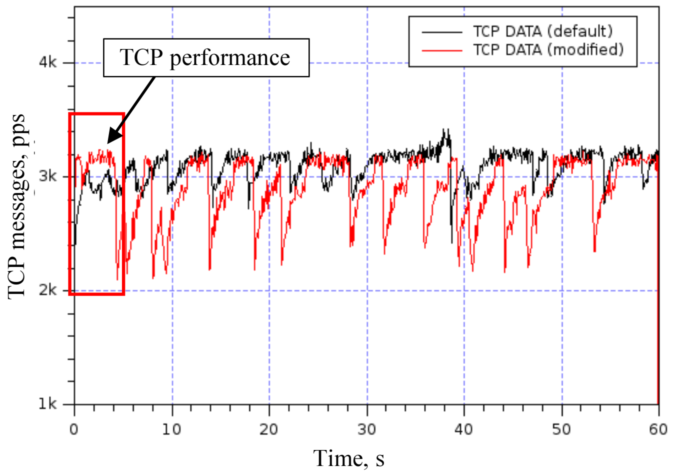


Fig. 3.21. Default and modified Linux kernel data transmission performance of transmission control protocol running with high central processor unit load

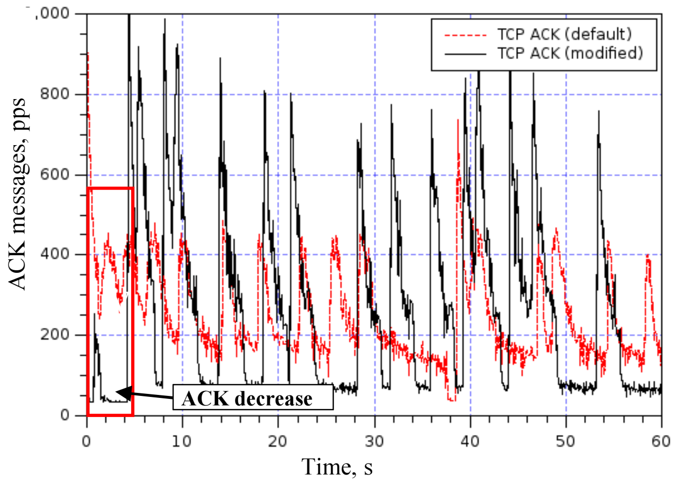


Fig. 3.22. Default and modified Linux kernel acknowledgment message rate with high central processor unit load

In both cases (with and without ACK limiting), a similar ACK message flow graph is seen (Fig. 3.22). The only noticeable difference is that the default system ACK rate is controlled by receive window (*tp–>rcv_wnd*), or in modified version of TCP it's done be received TCP data over RTT time and receive window algorithm. In this situation the performance of TCP decreases due to add additional computing load which is added to modified system and bigger ACK messages rate deviation over the testing time. As the final ACK message sending decision is made by *tp–>rcv_wnd* variable value.

In both testing results of TCP seen in Fig. 3.21 and 3.22 at the beginning of TCP session from 0 – 5 s, a noticeable performance gain on modified system is seen. Showing that modified system is performing better than default system and sending ~ 100 TCP data messages more. Also the ACK messages rate is ~150 less compared to default system. At this point the ACK sending rate is controlled by ACK limiting algorithm, which allows to reduce CPU load and increase the TCP performance like in virtual systems testing scenario.

The same testing results can be observed also in TCP RTT graph, (Fig. 3.23 and 3.24). The default stack in overall is performing better and having the average RTT a bit smaller compared to modified version. It is also noticeable that the RTT spikes are bigger and the higher almost by 1 ms, but in the start of TCP session the same RTT values are observed. This indicates that ACK limiting algorithm has no negative impact to TCP RTT value when running in alone.

The given result were made in directly connected devices with a network switch with minimal network latency. In case of bigger delay (due to networks equipment or due transmission delay) the difference of throughput and RTT can be even smaller.

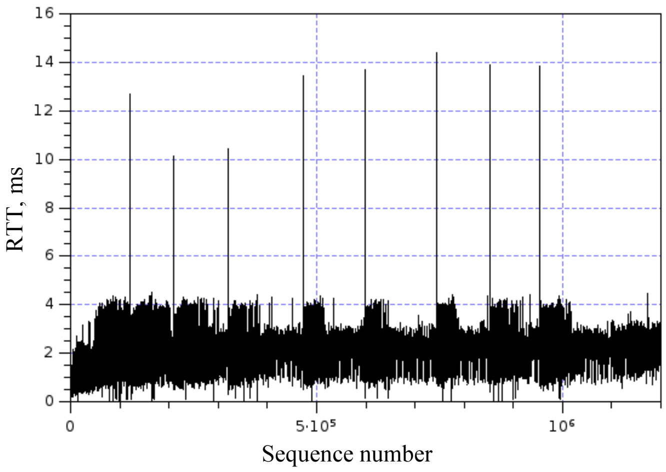


Fig. 3.23. Round trip time deviation with default Linux kernel running on RouterStation with high central processor unit load

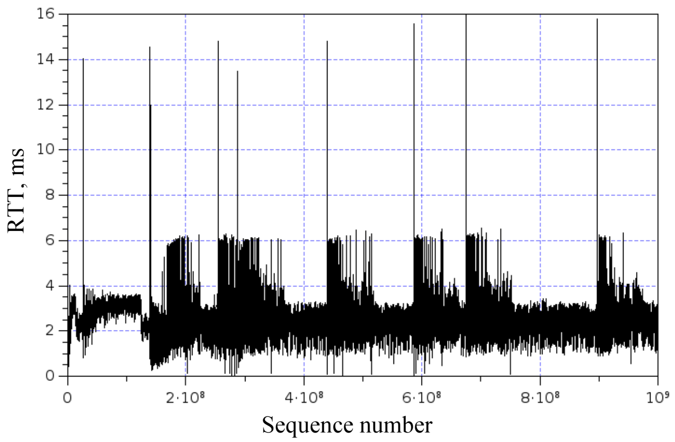


Fig. 3.24. Round trip time deviation with acknowledgment limiting enabled running on RouterStation with high central processor unit load

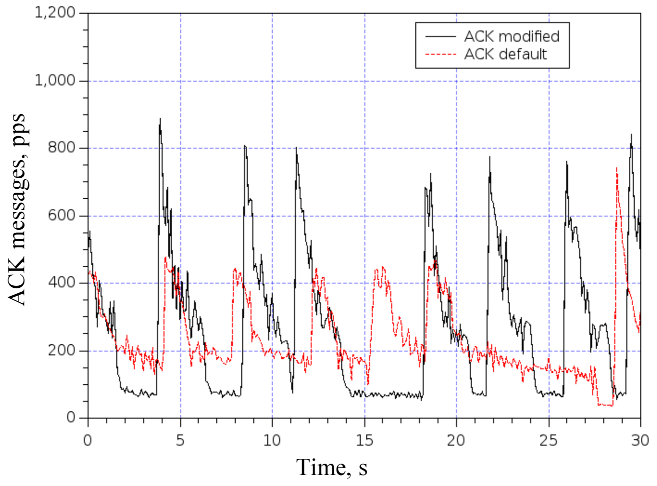


Fig. 3.25. Acknowledgment limiting algorithm comparison with load reduction algorithm running with high central processor unit load

By comparing the result of ACK message rates which were received during the tests with default Linux OS configuration, the observation are seen that the both graph (Fig. 3.25) share similar curves of how ACK rate changes during testing time. In case of default TCP stack the ACK rate changes happen more frequently and are from ~ 400 to ~200. The modified system ACK messages rate reductions starts from ~800 to ~100 and is linear at the end, before the reset happens. By eliminating the upper limit *ack_rate_max* in ACK limiting algorithm the system would end by decreasing the TCP throughput due to small ACK message rate. This can be used to replace the default kernel rate limiting algorithm.

In both cases the system which is running the default or modified TCP stack is reducing the load to the main CPU by limiting the ACK message rate. The only difference that the ACK limiting algorithm start to reduce system load from the beginning of TCP session compared to default method which only start at critical point.

3.3. Conclusions of Chapter 3

The experiments were performed to find how significantly ACK limiting algorithm effects real system and Linux kernel packet flow and processing algorithms. The conclusion are:

1. ACK limiting algorithm does not significantly affect systems or the TCP data transfer in not congested network links. In this condition when the network or operating systems are not under heavy load the ACK limiting has no effect to TCP session stability. However, the packet drop can be caused by expiration of t_{RTO} in same cases. This is easily seen when testing systems are under the heavy load and fluctuation of RTT can easily cause of TCP packet retransmission on system using modified Linux kernel and session is terminating immediately.
2. ACK limiting algorithm can increase RTT deviation several times, depending from CPU load and CWND size, causing the system exceeding the t_{RTO} time value. This is mostly to happen in the systems with heavy CPU load. Even with this disadvantage the results in virtual environment shows much better average performance of TCP throughput compared to default system.
3. By using different ACK limit variable values (using virtual Linux file system to access kernel variables) it is possible dynamically adjust the ACK limit function and change the ACK limiting update period manually, so changing the behaver of the ACK limiting growth function.
4. The ACK message limiting has a big impact (up to 32%) not only to the CPU load network equipment or TCP receiving system but also to the CPU load of the TCP sending system. In case then the TCP data sending node is a lower power embedded device, the observed TCP data throughput gain can by up to 50%, by using ACK limiting algorithm on TCP data receiver.
5. The degradation of TCP data throughput can be observer in case the TCP receiver is the bottle neck (the CPU load of TCP receiver was limiting the receiving speed). In this case the TCP throughput can depredate due to additional system load of ACK limiting algorithm and correlation of two algorithm running same time. In such case ACK limiting and default kernel system algorithm tries to reduce the ACK rate.
6. The experimental results show that ACK limiting work in practical situations allowing to increase the TCP throughput in tested system and reduce unneeded CPU load up to 50%. As seen from tests the ACK limit upper limit should be carefully selected based on CWND value and network conditions. If ACK limiting value is too high, the TCP packet drop can happened more frequently due to RTO time expiration. Also in different systems configurations and network conditions the TCP performance gain can change and even in some cases a little degradation of TCP performance can happen. But in most cases TCP throughput and goodput can be increased by using proposed ACK limiting algorithm.

General Conclusions

In the dissertation the investigation of TCP functions and ACK limiting was presented. The following significant results in dissertation were obtained:

1. ACK filtering in heterogeneous network layer does not significantly affect the TCP data transfer in normal network conditions. The TCP data transfer remains stable for up to 80% of ACK drops.
2. The performance of router CPU depends on ACK message rate and the performance can be increased with ACK filtering. The CPU load utilization is linear, and depends on TCP data and ACK message rate. With 80% of ACK drop, the performance can be increased by 32% with CPU load of 60%.
3. The ACK filtering in Linux OS kernel and network equipment can be used not only with single TCP session but, also with concurrent sessions. Results show that two data TCP sessions can work without any evident signs of the instabilities, although a slight unfairness among TCP sessions were observed.
4. ACK filtering enabled on network equipment does not affect TCP session stability or TCP throughput and can work with multiple TCP sessions in not congested networks environments. But in more dynamic networks with higher RTT deviation and packet drop the initial values must be recalculated for better performance and stability.

5. By using ACK filtering in asymmetric high speed non homogenies networks the TCP protocol ACK overhead several time and reduce the total network of PPS (packet per second) up to 30%. By reducing the network utilization we not only reduce the system load to network devices, but also can increase the network capacity up to 10% in asymmetric networks (IEEE 802.11).
6. Dynamic ACK limiting implementation is Linux OS kernel has not affect systems stability or the TCP data transfer in normal network. In normal condition when the network or operating systems are not under heavy load and TCP RWIN is bigger than BDP the ACK messages limiting has no effect to TCP session stability.
7. Too high ACK limiting value can cause increase and jittering of RTT value, causing the system exceeding the t_{RTO} timer and data resending. This is mostly to happen in the systems with high CPU load.
8. The dissertation results show that ACK limiting, in network devices and Linux OS kernel TCP stack work in practical situations, allowing to increase the TCP throughput up to 50% in tested system and reduce unneeded CPU load up to 32%. The upper ACK limit value should be carefully selected. If it is set to high TCP packet drop (expiration of t_{RTO} timer) could happened more frequently causing TCP session distortions and TCP performance degradation.

References

Al-Khatib W., Gunavathi K. 2006. A New Approach to Improve TCP Performance over Asymmetric Networks, *Electronics and Electrical Engineering* 7(71): 1392–1215.

Allman M. 2003. TCP Congestion Control with Appropriate Byte Counting (ABC), *IETF RFC3465*: 1–10.

Allman M., Floyd S., Partridge C. 2002. Increasing TCP's initial window, *IETF RFC 3390*: 1–15.

Allman M., Glover D., Sanchez L. 1999. Enhancing TCP over satellite channels using standard mechanisms, *IETF RFC 2488*: 1–19.

Allman M., Paxson V., Stevens W. 1999. TCP Congestion Control, *IETF RFC 2581*: 1–14.

Alrshah M. A., Othman M., Ali B., Mohd Hanapi Z. 2014. Comparative study of high-speed Linux TCP variants over high-BDP networks, *Journal of Network and Computer Applications* 43: 66–75.

Bach M. J. 1990. *The design of the UNIX operating system*Prentice-Hall.

Balachandran A., Voelker G.M., Bahl P., Rangan P.V. 2002. Characterizing user behavior and network performance in a public wireless LAN, *ACM SIGMETRICS Performance Evaluation Review* 30(1): 195–205.

Balakrishnan H., Padmanabhan V. 2002. TCP performance implications of network path asymmetry, *IETF RFC3449*: 1–41.

Bao W., Wong V. W. S., Leung V.C.M. 2010. A model for steady state throughput of TCP CUBIC *GLOBECOM - IEEE Global Telecommunications Conference*.

Beekmans G. 2010. Linux From Scratch, *Linux*: 1–318.

Bellovin S. 1996. Defending Against Sequence Number Attacks, *IETF RFC 1948*.

Bencivenni M., Carbone A., Fella A., Galli D., Marconi U., Peco G., Perazzini S., Vagnoni V., Zani S. 2009. High rate packet transmission on 10 Gbit/s Ethernet LAN using commodity hardware *IEEE-NPSS Real Time Conference*. IEEE.

Berkeley I. U. C., Allman M., Chu J., Sargent M. 2011. Computing TCP's Retransmission Timer, *IETF RFC 6298*: 1–12.

Bhuiyan H., Mcginley M., Li T., Veeraraghavan M. 2009. TCP Implementation in Linux : A Brief Tutorial, *Technical Memorandum*.

Bianchi G. 2000. Performance analysis of the IEEE 802.11 distributed coordination function, *IEEE Journal on Selected Areas in Communications* 18(3): 535–547.

Bott R. 2014. TCP Congestion Control, *IETF RFC 5681* (1): 1–5.

Cáceres R., Danzig P. B., Jamin S., Mitzel D. J. 1991. Characteristics of wide-area TCP/IP conversations *Proceedings of the conference on Communications architecture & protocols - SIGCOMM '91.* , New York, New York, USA: ACM Press.

Chen B., Marsic I., Shao H. R., Miller R. 2009. Improved Delayed ACK for TCP over Multi-Hop Wireless Networks *2009 IEEE Wireless Communications and Networking Conference*. IEEE.

Chen Y. C., Kurose J., Towsley D. 2011. A simple queueing network model of mobility in a campus wireless network *Proceedings of the 3rd ACM workshop on Wireless of the students ' by the students ' for the students - S3 '11.* , New York, New York, USA: ACM Press.

Choi W. Y. 2012. Optimal frame aggregation level for IEEE 802.11 PCF protocol, *AEU - International Journal of Electronics and Communications* 66(1): 23–27.

Clark D. D. 1982. Window and acknowledgemnt strategy in TCP, *IETF RFC 813*.

Dalton L. a., Isen C. 2004. A study on high speed TCP protocols, *IEEE Global Telecommunications Conference ' 2004. GLOBECOM '04.* 2: 851–855.

Dutt S., Habibi D., Ahmad I. 2012. A low cost Atheros system-on-chip and OpenWrt based testbed for 802.11 WLAN research *IEEE Region 10 Annual International Conference, Proceedings/TENCON*.

Eidukas D., Valinevi A., Vilutis G., Kilius Š., Vasylius T. 2005. Information Network Loading Evaluation, *Elektronika ir Elektrotechnika* 8(8).

Floyd S. 2003. HighSpeed TCP for large congestion windows, *IETF RFC3649*: 1–34.

Floyd S., Padhye J., Handley M. 2000. TCP Congestion Window Validation, *IETF RFC 2861*.

Fox R. 1989. TCP big window and NAK options, *IETF RFC 1106*: 1–13.

Garsva E., Paulauskas N., Grazulevicius G., Gulbinovic L. 2014. Packet inter-arrival time distribution in academic computer network, *Elektronika ir Elektrotechnika* 20(3): 87–90.

Gast M. 2005. 802.11 Wireless Networks: The Definitive Guide, Second Edition, *OReilly Media*: 1–656.

Griner J., et al. 2000. Ongoing TCP Research Related to Satellites, *IETF RFC 2760*.

Ha S., Rhee I. 2008. CUBIC : A New TCP-Friendly High-Speed TCP Variant, *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel* 42(5): 64–74.

Haitao Wu, Jing Wu, Shiduan Cheng, Jian Ma 1999. ACK filtering on bandwidth asymmetry networks *Fifth Asia-Pacific Conference on ... and Fourth Optoelectronics and Communications Conference on Communications,*. IEEE.

Handley M., Padhye J., Floyd S. 2000. TCP Congestion Window Validation, *IETF RFC 2861* (2861): 1–12.

Information Sciences Institute/University of Southern California 1981. Internet Protocol - DARPA Internet Program Protocol Specification, *IETF RFC 791*.

Jacobson V. 1990. Compressing TCP/IP Headers for Low-Speed Serial Links, *IETF RFC 1144*: 46.

Jacobson V. 1988. Congestion avoidance and control, *ACM SIGCOMM Computer Communication Review* 18(4): 314–329.

Jacobson V., Braden R., Borman D. 1992. TCP extensions for high performance, *IETF RFC1323*: 1–37.

Jain K., Padhye J., Padmanabhan V.N., Qiu L. 2005. Impact of interference on multi-hop wireless network performance *Wireless Networks*.

Jain S., Raina G. 2011. An experimental evaluation of CUBIC TCP in a small buffer regime *2011 National Conference on Communications, NCC 2011*.

Kajackas A., Batkauskas V., Saltis A., Gursnys D. 2011. Autonomous System for Observation of QoS in Telecommunications Networks, *Electronics and Electrical Engineering* 111(5): 15–18.

Kajackas A., Šaltis A., Vindašius A. 2015. User Access Link Impact on Web Browsing Quality, *Electronics and Electrical Engineering* 100(4): 59–64.

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

Karn P. 2011. Index of ftp://ftp.isi.edu/.

Kaur M. A., Vijay S., Gupta S. C. 2010. Performance Analysis and Enhancement of IEEE 802.11 Wireless Local Area Networks, *Simulation* 9(5): 5–8.

Keceli F., Inan I., Ayanoglu E. 2007. TCP ACK Congestion Control and Filtering for Fairness Provision in the Uplink of IEEE 802.11 Infrastructure Basic Service Set *2007 IEEE International Conference on Communications*. IEEE.

Kim H., Claffy K., Fomenkov M., Barman D., Faloutsos M., Lee K. 2008. Internet traffic classification demystified *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08.* , New York, New York, USA: ACM Press.

Kojo M., Hata M., Yamamoto K., Sarolahti P. 2009. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP, *IETF RFC 4138.*

Kotz D., Essien K. 2002. Analysis of a campus-wide wireless network *Proceedings of the 8th annual international conference on Mobile computing and networking - MobiCom '02.* , New York, New York, USA: ACM Press.

Lahey K. 2000. TCP Problems with Path MTU Discovery, *IETF RFC 2923.*

Li T., Ni Q., Malone D., Leith D., Xiao Y., Turletti T. 2009. Aggregation with fragment retransmission for very high-speed WLANs, *IEEE/ACM Transactions on Networking* 17(2): 591–604.

Maier G., Feldmann A., Paxson V., Allman M. 2009. On dominant characteristics of residential broadband internet traffic *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09.*

Maxim V., Zidek K. 2012. Design of high performance multimedia control system for UAV/UGV based on SoC/FPGA Core *Procedia Engineering.*

McKenzie A.M. 1989. Problem with the TCP big window option, *IETF RFC 1110.*

Miguel V., Cabrera J., Jaureguizar F., García N. 2011. Distribution of high-definition video in 802.11 wireless home networks, *IEEE Transactions on Consumer Electronics* 57(1): 53–61.

Misra, S., Isaac W. and S.C.M. 2009. *Guide to Wireless Ad Hoc Networks.*

Nagle J. 1984. Congestion Control in IP/TCP Internetworks, *Ford Aerospace and Communications Corporation* 53(January): 160.

Paredes-Farrera M., Fleury M., Ghanbari M. 2006. Router Response to Traffic at a Bottleneck Link.

Parham B., Touch J. Implementing the Internet Checksum in Hardware.

Paxson V., Allman M., Dawson S., Fenner W. 1999. Known TCP implementation problems (RFC2525), *IETF*: 1–62.

Podolsky M., Mahdavi J., Romanow A., Floyd S., Mathis M. An Extension to the Selective Acknowledgement (SACK) Option for TCP.

Potorac A.D., Onofrei A., Balan D. 2015. An Efficiency Optimization Model for 802.11 Wireless Communication Channels, *Elektronika ir Elektrotechnika* 97(1): 67–72.

Pranevičius H., Kirvėlaitis T., Pranevičienė I. 2006. Numerical Model of Transmission Control Protocol, *Nr* 6(70): 1392–1215.

Priescu V. G., Bos H., Vogt D. 2012. *Efficient Use of Heterogeneous Multicore Architectures in Reliable Multiserver Systems MASTER THESIS Supervisors :* Vrije Universiteit Amsterdam.

Ramakrishnan K., Floyd S., Black D. 2001. The Addition of Explicit Congestion Notification (ECN) to IP, *IETF RFC 3168* 3168(3168): 1–64.

Rathore M., Hidell M., Sjödin P. 2013. KVM vs. LXC: comparing performance and isolation of hardware-assisted virtual routers, *American Journal of Networks and Communications* 2(4): 88–96.

Rindzevičius R., Tervydis P., Narbutaitė L., Pilkauskas V. 2015. Performance Analysis of Data Packet Transmission Network with the Unreliable Channels, *Elektronika ir Elektrotechnika* 84(4): 59–62.

Rose M., McCloghrie K. 1991. Management Information Base for Network Management of TCP/IP-based internets: MIB-II.

Sarolahti P., Sarolahti P., Sarolahti P., Kojo M., Kojo M., Raatikainen K., Raatikainen K. 2003. F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts.

Savage S., Cardwell N., Wetherall D., Anderson T. 1999. TCP congestion control with a misbehaving receiver, *ACM SIGCOMM Computer Communication Review* 29(5): 71.

Schimmel C., Curt 1994. *UNIX systems for modern architectures : symmetric multiprocesssing and caching for kernel programmers*Addison-Wesley.

Seth S., Venkatesulu M.A., Wiley InterScience (Online service) 2008. *TCP/IP architecture, design and implementation in Linux*Wiley.

Shepard T., Partridge C. 1998. When TCP Starts Up With Four Packets Into Only Three Buffers.

Socolofsky T. J., Kale C. J. 1991. TCP/IP tutorial.

Soediono B. 1989. Defending Against Sequence Number Attacks -RFC6528, *Journal of Chemical Information and Modeling* 53: 160.

Stevens W. R. 1997a. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, *IETF RFC 2581*: 5.

Stevens W. R. 1997b. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, *IETF RFC 2001*.

Stevens W. R., Narten T. 1990. *Unix network programming*.

Stevens W. R., Rago S.A. 2014. Advanced Programming in the Unix Environment, *Igarss 2014* (1): 1–5.

Tafa I., Beqiri E., Paci H., Kajo E., Xhuvani A. 2011. The evaluation of Transfer Time, CPU Consumption and Memory Utilization in XEN-PV, XEN-HVM, OpenVZ, KVM-FV and KVM-PV Hypervisors using FTP and HTTP approaches *Proceedings - 3rd IEEE International Conference on Intelligent Networking and Collaborative Systems, INCoS 2011*.

Touch J. 2007. Defending TCP Against Spoofing Attacks, *RFC 4953*: 28.

Tschofenig H., Davies E. 2009. *Quality of Service Parameters for Usage with Diameter* J. Korhonen, sud.

Unzner M., et al. 2014. Diplomarbeit A Split TCP/IP Stack Implementation for GNU/Linux.

Vahalia U. 1995. *UNIX Internals: The New Frontiers*Prentice Hall.

Vindašius, A. 2015. Security State of Wireless Networks, *Elektronika ir Elektrotechnika* 71(7): 19–22.

Wellman B., Haythornthwaite C. 2008. *The Internet in everyday life*.

Wu Q. X. 2012. The Research and Application of Firewall based on Netfilter, *Physics Procedia* 25: 1231–1235.

Xiao Y. 2005. IEEE 802.11 performance enhancement via concatenation and piggyback mechanisms, *IEEE Transactions on Wireless Communications* 4(5): 2182–2192.

Xiao Y., Rosdahl J. 2002. Throughput and Delay Limits of IEEE 802.11, *IEEE COMMUNICATIONS LETTERS* 6(8).

Zhang B., Wang X., Lai R., Yang L., Wang Z., Luo Y., Li X. 2010. Evaluating and optimizing I/O virtualization in kernel-based virtual machine (KVM) *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

Zhao L., Zhang J., Zhang H. 2009. Hub-polling-based IEEE 802.11 PCF with integrated QoS differentiation, *Wireless Communications and Mobile Computing* 9(9): 1220–1230.

List of Scientific Publications by the Author on the Topic of the Dissertation

Papers in the Reviewed Scientific Journals

1. Statkus A., Paulikas S. 2013. Improving TCP Performance in IEEE 802.11 Networks, *Electronics and Electrical Engineering* 19(5): 99–102. DOI: 10.5755/j01.eee.19.5.2571, ISSN 1392-1215.
2. Statkus A. Paulikas S. 2013. Analysis of Wi-Fi Access Networks Situation in the City Area, World Academy of Science, *Engineering and Technology* 7: 145–148 Internet access: http://waset.org/publications/14832/analysis-of-wi-fi-access-networks-situation-in-the-city-area, ISSN 2010-376X.
3. Pavilanskas L., Statkus A. 2010. Evaluation of TCP Acknowledgment Mechanism Influence on Router Performance, *Electronics and Electrical Engineering* 103(7): 95–100. DOI: 10.5755/j01.eee.103.7.9287, ISSN 1392-1215.
4. Statkus A., Paulikas S. 2012. Analysis of Home Wi-Fi Internet Access Networks Situation in Vilnius City, *Electronics and Electrical Engineering* 118(2): 77–80. DOI: 10.5755/j01.eee.118.2.1178, ISSN 1392-1215.

Summary in Lithuanian

Įvadas

Problemos formulavimas

Šiandien vis sparčiau augant duomenų srautams, itin svarbu valdyti duomenų perdavimo kanalo pralaidą, siekiant kuo efektyviau išnaudoti duomenų perdavimo kanalus bei tinklo įrenginius. Šiuo metu pagrindinis duomenų apsikeitimo protokolas tiek Internete, tiek vietiniam tinkle (ang. *Local Area Connection – LAN*) tinkluose yra transporto valdymo protokolas (angl. *Transmission Control Protocol – TCP*) protokolas. Tai vienas iš pagrindinių duomenų apsikeitimo protokolų Internete, kurio pagrindu veikia didžioji dalis kitų, aukštesnio lygio, protokolų. Tai į ryšį orientuotas duomenų perdavimo protokolas, garantuojantis patikimą duomenų perdavimą tarp dviejų nutolusių tinklo taškų ar vidinių kompiuterio programų.

Mažų greitaveikų duomenų tinklai vis dar plačiai paplitę ir sudaro didelę dalį duomenų perdavimo tinklų. Tačiau nuolat besiplečiantys didelės spartos tinklai, per kuriuos perduodama didžioji dalis informacijos, darosi vis populiaresni ir labiau pasiekiami paprastiems vartotojams. Nuolat augant perduodamos informacijos kiekiui bei jos perdavimo greitaveikai, esamas TCP protokolas tampa ne toks efektyvus ir generuoja ne tik didelį duomenų perteklumą, bet ir reikalauja didesnių centrinio procesoriaus (angl. *Central Processing Unit – CPU*) resursų. Šis TCP informacinis perteklumas kuria ir papildomą apkrovą ne tik tinklo elementams bet ir TCP klientui bei mažina serverių našumą (Nagle 1984; Paxson *et al.* 1999; Garsva *et al.* 2014).

Nors šių dienų elektronika nuolat tobulėja ir tampa vis efektyvesne energijos suvartojimo atžvilgiu, tačiau vidutinis TCP paketų apdorojimas vis dar sukuria apie 4–13 % sistemos apkrovos (Seth *et al.* 2008; Vahalia 1995).

Siekiant išspręsti minėtas problemas disertacijoje iškelta ir įrodyta hipotezė: TCP perteklumo mažinimas filtruojant patvirtinimo (angl. *Acknowledgment – ACK*) paketus sumažina tinklo ir jo elektroninių įrenginių apkrovą bei padidina jais perduodamų TCP duomenų pralaidą nesukuriant neigiamos įtakos TCP sesijos stabilumui ir jos atsistatymui po nutrūkimo.

Darbo aktualumas

Pastaruosius metus stebimas pastovus Interneto duomenų srauto didėjimas. Labai svarbu, kad dabar naudojamas transporto valdymo protokolas (TCP) nebūtų Interneto ir duomenų perdavimo tinklų tolesnės evoliucijos trukdžiu, o leistu lengviau ir efektyviau išnaudoti esamas duomenų ryšio linijas ir elektronines sistemas.

Nors pastoviai tobulėjanti elektronika ir efektyvesni energijos taupymo metodai leidžia sumažinti energijos suvartojimą, tačiau vis didėjantys duomenų srautai suvartoja vis daugiau centrinio procesoriaus resursų, kurie gali būti panaudoti kitiems sisteminiams procesams ar programoms.

Tyrimo objektas

Tyrimo objektas – transporto valdymo protokolo (TCP) taikymas šiuolaikinių elektroninių ryšio sistemų programinėje įrangoje.

Darbo tikslas

Disertacijos tikslas – padidinti transporto valdymo protokolo (TCP) našumą heterogeniniuose tinkluose ir elektroninių tinklų įrenginiuose, sumažinant TCP patvirtinimo paketų kiekį operacinėse sistemose ir tinklo įrenginiuose.

Darbo uždaviniai

1. Išanalizuoti transporto valdymo protokolo efektyvumą ir veikimą operacinėje sistemoje, duomenis perduodant nehomogeniniais duomenų tinklais ir asimetrinėmis ryšio linijomis.
2. Išanalizuoti TCP patvirtinimo paketų filtravimo elektroniniuose tinklo įrenginiuose ribas ir taisykles ir sukurti TCP patvirtinimo paketų perteklumo mažinimo metodą Linux operacinei sistemai.
3. Eksperimentiškai ištirti sukurto TCP patvirtinimo paketų perteklumo mažinimo metodo daromą įtaką TCP efektyvumui, kuomet duomenys perduodami nehomogeniniais tinklais.

Tyrimų metodika

Darbe taikomi šie moksliniai metodai: literatūros apžvalga, saugumo ir rizikos analizė, dirbtinių kompiuterinių tinklų su kintamu vėlinimu ir duomenų pralaidumu charakterizavimo metodika, programinio kodo optimizavimo ir veikimo analizė, statistinės analizės, duomenų perdavimo charakterizavimo metodai ir statistinė analizė, patikimumo ir gautų duomenų patvirtinimo metodai.

Mokslinis naujumas

Darbo metu pasiekti šie mokslui reikšmingi rezultatai:

1. Pasiūlytas naujas dinaminis TCP patvirtinimo paketų ribojimo metodas skirtas Linux operacinės sistemos branduoliui, kuris efektyviai sumažina ir valdo TCP patvirtinimo paketų (ACK) perteklumą.
2. Pasiūlytas naujas patvirtinimo paketų (ACK) filtravimo ribų nustatymo algoritmas nehomogeniniams tinklams.
3. Sukurtas patvirtinimo paketų (ACK) perteklumo mažinimo algoritmas IEEE 802.11 belaidžiame tinkle ir jų įrenginiuose.

Darbo rezultatų praktinė reikšmė

Sukurti du TCP perteklumo mažinimo algoritmai. Pirmasis algoritmas įgyvendina ACK filtravimą tinklo įrenginiuose. Jis leidžia sumažinti TCP perteklumą ir nereikalauja galinės tinklo sistemos įrenginių keitimo. Tinklo apkrova ACK duomenų paketais yra sumažinama iki 80 %, o tinkle maršrutų parinktuvų našumas padidėja iki 32 %.

Antrasis algoritmas, įgyvendintas modifikuojant Linux OS branduolį, leidžia ne tik sumažinti TCP perteklumą ir reikiamus tinklo resursus, tačiau kartu dinamiškai mažina tinklo ir jo elektroninių įrenginių apkrovą, padidindamas TCP našumą tiek kliento, tiek serverio pusėje.

Abu sukurti TCP perteklumo mažinimo algoritmai gali būti naudojami IP tinkluose siekiant sumažinti tinklo įrenginių apkrovą ir padidinti asimetrinių ryšio linijų pralaidą. Algoritmas įgyvendintas Linux OS iki 50 % sumažina galinės ryšio sistemos apkrovą ir padidina jos našumą.

Ginamieji teiginiai

1. ACK ribojimas tinklo įrenginiuose iki 80 % sumažina ACK paketų skaičių bei pačią tinklo apkrovą ir iki 32 % padidina tinklo maršruto parinktuvo našumą, nedarant įtakos užmegztos TCP sesijos stabilumui.
2. ACK ribojimas tinklo įrenginiuose ir ACK ribojimas Linux OS branduolyje neturi įtakos užmegztos TCP sesijos našumui ir jos efektyviam atsistatymui po duomenų paketų praradimo.
3. Dinaminis ACK ribojimas Linux OS branduolyje sumažina tinklo įrenginių CPU apkrovą ir iki 50 % padidina TCP pralaidą įterptinėse sistemose.

Darbo rezultatų aprobavimas

Pagrindiniai disertacijos rezultatai buvo paskelbti 4 mokslo straipsniuose įtrauktuose į *Thomson Reuters Web of Science* duomenų bazę ir turinčiuose citavimo indeksą: 2 straipsniai užsienio žurnaluose ir 2 straipsniuose vietiniame žurnale.
Disertacijoje atlikti tyrimai ir jų rezultatai buvo aprobuoti 4 mokslinėse konferencijose:

1. Evaluation of TCP Acknowledgment Mechanism Influence on Router Performance. 14 ELECTRONICS'2010 , 2010, Lietuva, Vilnius.
2. Vilniaus miesto bevielių tinklų statistiniai tyrimai. Jaunųjų mokslininkų konferencija „Elektronika ir elektrotechnika 2011“, 2011, Lietuva, Vilnius.
3. Analysis of Home WiFi Internet Access Networks Situation in Vilnius City. 15 tarptautinė konferencija ELECTRONICS 2011, 2011, Lietuva, Vilnius.
4. Analysis of Home WiFi Access Networks Situation in City Area. ICCCISE 2013: International Conference on Computer, Communication and Information Sciences and Engineering, 2013, Ispanija, Barselona.

Disertacijos struktūra

Disertacija parengta anglų kalba ir sudaryta iš trijų skyrių, bendrųjų išvadų, literatūros sąrašo ir santraukos lietuvių kalba. Darbo apimtis – 93 puslapių be priedų, tekste panaudota 63 paveikslai, 2 lentelės, 35 formulės. Rašant disertaciją panaudotas 91 literatūros šaltinis.

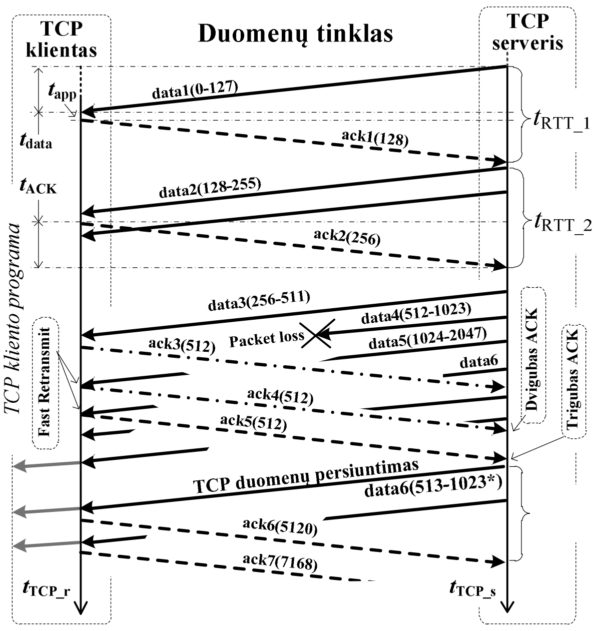
1. Transporto valdymo protokolo perteklumas heterogeniniuose duomenų tinkluose

TCP šiuo metu yra vienas iš pagrindinių duomenų perdavimo protokolų, naudojamų Internete. Šio protokolo pagrindinė paskirtis yra patikimas duomenų perdavimas tarp dviejų nutolusių taškų per duomenų perdavimo tinklą ar kompiuterio viduje. Siekiant patikimo duomenų perdavimo TCP naudoja ACK paketus (duomenų gavimo patvirtinimas), kurių pagalba yra užtikrinamas patikimas ir vientisas duomenų pristatymas iš siuntėjo gavėjui. Tam tikslui TCP pakete yra išskirti du laukai: patvirtinimo (angl. *Acknowledgment*) ir eiliškumo (angl. *Sequence)*. Abu TCP antraštės laukai yra 32 bitų ilgio ir yra naudojami patvirtinti gautos informacijos kiekį baitais ir siunčiamos informacijos vientisumui išlaikyti duomenų gavėjo pusėje (Socolofsky *et al.* 1991; Bott 2014).

Pagal RFC 813 duomenų gavėjas siunčia TCP patvirtinimo paketą ACK su patvirtinimo numeriu. Paketas nurodo sėkmingai gautą paskutinį duomenų skaičių baitais ir yra generuojamas po kiekvieno antro sėkmingai gauto TCP duomenų paketo. Kadangi ACK paketų generavimas reikalauja ne tik TCP siuntėjo sistemos resursų, bet ir sukuria papildomą apkrovą tinklo elementams, ACK paketų siuntimo dažnumas turi atitikti šiuos reikalavimus (Bott 2014; Stevens 1997; Berkeley *et al.* 2011):

1. ACK paketas turėtų būtų siunčiamas po kiekvieno antro sėkmingai gauto TCP duomenų paketo (RFC 5681).
2. ACK paketas turėtų būti siunčiamas gavus TCP paketą su nustatytu PUSH parametru.

3. Pasikeitus TCP duomenų siuntimo lango dydžiui.
4. Viršijus laukimo laikui (kai daugiau nėra gaunama jokių papildomų TCP duomenų paketų iš siuntėjo).

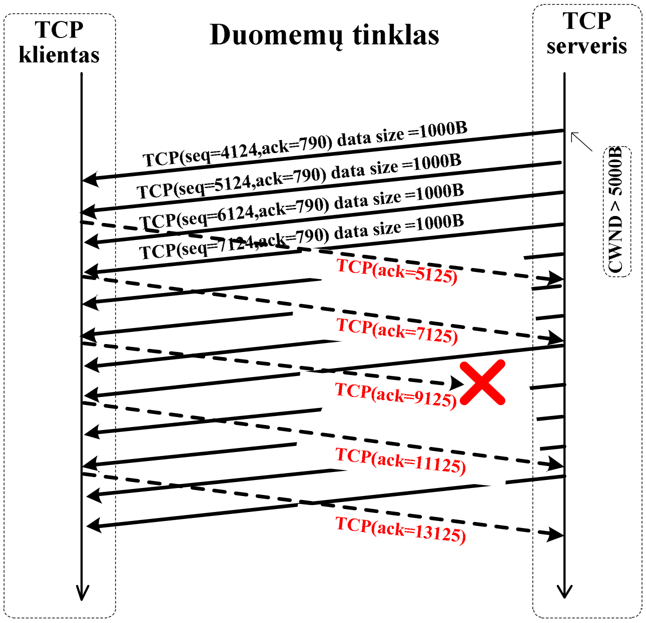


S.1.1 pav. Transporto valdymo protokolo veikimas ir duomenų apsikeitimas procesas

Be duomenų patvirtinimo funkcijos ACK paketai vaidina labai svarbų vaidmenį reguliuojant siunčiamų duomenų greitaveiką (S.1.1 pav.). Norint efektyviai išnaudoti duomenų perdavimo kanalą TCP turi dinamiškai prisitaikyti prie pasikeitusių tinklo parametrų. Tam TCP naudoja keturis pagrindinius srauto kontrolės algoritmus: lėto starto (angl. s*low start*), persipildymo vengimo (angl. *congestion avoidance*), greito persiuntimo (angl. *fast retransmit*) ir greito atsistatymo (angl. *fast revovery*). Išvardinti algoritmai yra atsakingi už TCP sesijos veikimą bei atsistatymą po paketų praradimų ir remiasi ACK patvirtinimo paketais.

Nuolat augant duomenų perdavimo kanalų greitaveikai ir mažėjant klaidų tikimybei juose, ACK paketų perteklumo problema darosi vis aktualesnė.

Disertacijoje yra nagrinėjama ACK paketų perteklumo mažinimo problema, atliekant ACK filtravimą. Metodas yra paremtas tuo, kad TCP duomenų gavėjas siųsdamas ACK paketą informuoja TCP duomenų siuntėją apie sėkmingai priimtą bendrą informacijos kiekį, siunčiant paskutinių sėkmingai gautų duomenų patvirtinimo numerį. Jei duomenų perdavimo kanale įvyksta paketų praradimas ir dingsta vienas ar keli ACK paketai, sekantis siųstas ACK patvirtina visus iki tol gautus TCP duomenų paketus (S.1.2 pav.).


TCP klientas
Duomemų tinklas
TCP serveris
TCP(seq=4124,ack=790) data size =1000B
TCP(seq=5124,ack=790) data size =1000B
TCP(seq=6124,ack=790) data size =1000B
TCP(seq=7124,ack=790) data size =1000B
CWND > 5000B
TCP(ack=5125)
TCP(ack=7125)
TCP(ack=9125)
TCP(ack=11125)
TCP(ack=13125)

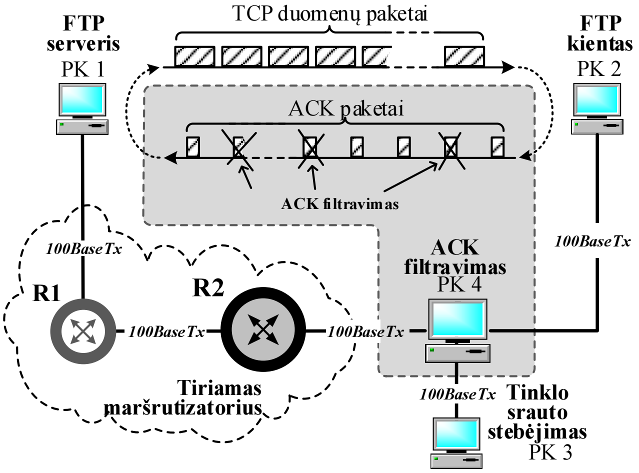
S.1.2 pav. Transporto valdymo protokolo perteklumo mažinimas atsisakant nereikalingų patvirtinimo paketų

Pagal RFC 5681 dokumentą TCP duomenų gavėjas turi siųsti ACK paketus kiekvienam antram sėkmingai gautam TCP duomenų segmentui. Jei dingsta vienas iš ACK paketų TCP sesija gali toliau sėkmingai veikti ir perduoti duomenis, be įtakos TCP sesijos stabilumui ir greitaveikai (1.2S pav.). Tačiau jei TCP sesija yra realaus laiko ar vyksta abipusis, nedidelės apimties duomenų apsikeitimas su mažais siuntimo TCP langais, ACK paketai gali būti siunčiami dažniau. Šiuo atveju bent vienas ACK paketų praradimas gali sukelti TCP duomenų sesijos sustojimą ir padidinti duomenų perdavimo laiką bei sukelti nepageidaujamą vėlinimą.

Pagrindinė ACK filtravimo veikimo sąlyga: TCP duomenų išsiuntimo langas (angl. *congestion window – CWND*) yra didesnis už tinklo duomenų tinklo vėlinimo dydį (angl. *bandwidth delay product – BDP*), o TCP sesija yra persipildymo vengimo fazėje. Per anksti aktyvavus ACK filtravimą ar esant nepakankamam TCP išsiuntimo langui, ACK ribojimas gali ne tik sumažinti tų perdavimų greitaveiką, bet ir sukurti papildomą vėlinimą bei TCP sesijos nutrūkimą. Per vėlai aktyvuotas ACK ribojimas dažniai neturi teigiamo efekto TCP duomenų perdavimo greitaveikiai ir tinklo elementams.

Norint nustatyti kokią įtaką turi ACK paketų ribojimas TCP sesijoms bei tinklo įrenginiams buvo atlikti eksperimentiniai tyrimai realiame tinkle. Tiriamasis tinklas buvo sudarytas iš TCP kliento kompiuterio (PK 2) ir TCP serverio (PK 1), sujungtų per tinklo įrenginius (R1, R2). Viso tinklo įrenginiai ir kompiuteriai buvo sujungti 100 Mbit/s Ethernet duomenų perdavimo prieiga.

Tyrimo metu buvo generuojamas 400 MB dydžio duomenų failas ir perduodamas per tinklą. Duomenų srautas buvo siunčiamas naudojantis <*ftp*> programa, perduodant failą iš TCP serverio TCP klientui (S.1.3 pav.).



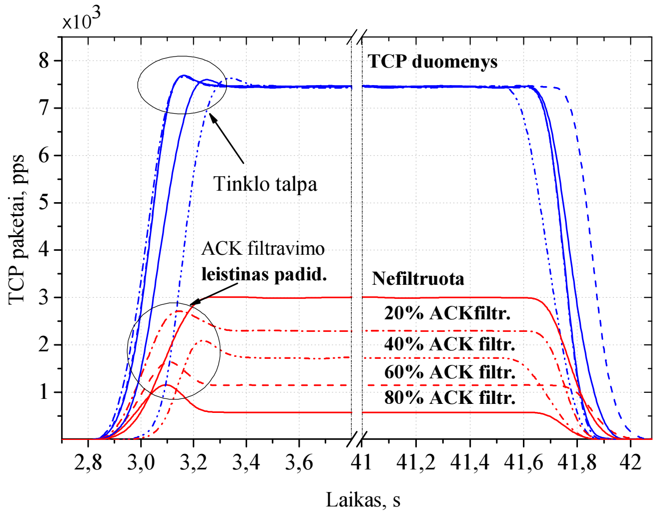
S.1.3 pav. Patvirtinimo paketų filtravimas tinklo įrenginyje

Eksperimento metu ACK filtravimo įrenginys (PK 4) buvo pajungtas iškart po TCP kliento (PK 2), taip ribojant ACK paketų skaičių į kitus tinklo įrenginius. Esamo kompiuterio Linux OS ACK filtravimo ribiniai eksperimento nustatymai pateikti S.1.4 paveiksle.

```
01 tc qdisc add dev eth1 ingress
02 tc filter add dev eth1 parent ffff:0
        protocol all prio 1 u32 match u32
        0xaff0001 0xffffffff at 16 classid
        ffff:0 police index 2 rate 12500bps
        burst 102400 mpu 0 action drop/pass
03 tc filter add dev eth1 parent ffff:0
        protocol all prio 1 u32 match u32
        0x0 0x0 at 0 classid ffff:0 police
        index 3 rate 1bps burst 1 action
        drop/drop
```

S.1.4 pav. Linux OS <*tc*> programos patvirtinimo paketų filtravimo kodas

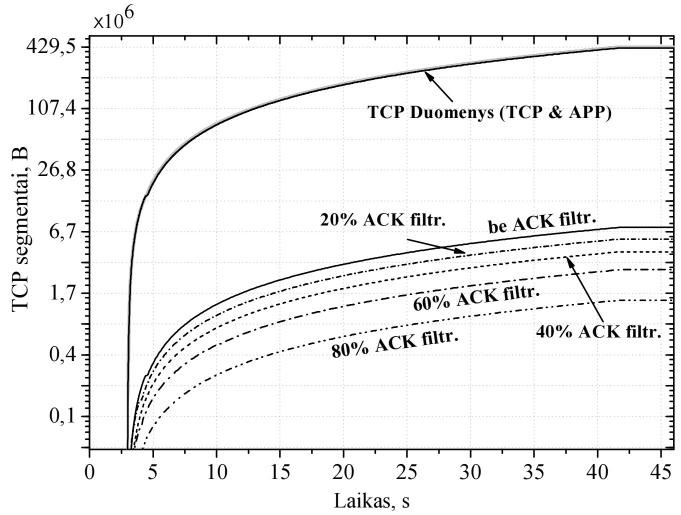
Pirmo eksperimento metu buvo atliktas TCP sesijos stabilumo tyrimas su skirtingomis ACK filtravimo reikšmėmis. Siekta nustatyti poveikį TCP stabilumui ir perduodamų duomenų greitaveikai. ACK filtravimo vertė buvo didinama nuo 0 % (be ACK filtravimo) iki 80 % ir stebima įtaka TCP sesijos stabilumui, duomenų perdavimo greitaveikai bei tinklo elementų apkrovai. Tinklo įrangos stebėjimui buvo naudojamas paprastasis tinklo valdymo protokolas (angl. *Simple Network Managment Protocol – SNMP*) protokolas, kuriuo periodiškai buvo surenkami tinklo elementų apkrovos parametrai, nesukuriant papildomos apkrovos tinklui ir jo įrenginiams.



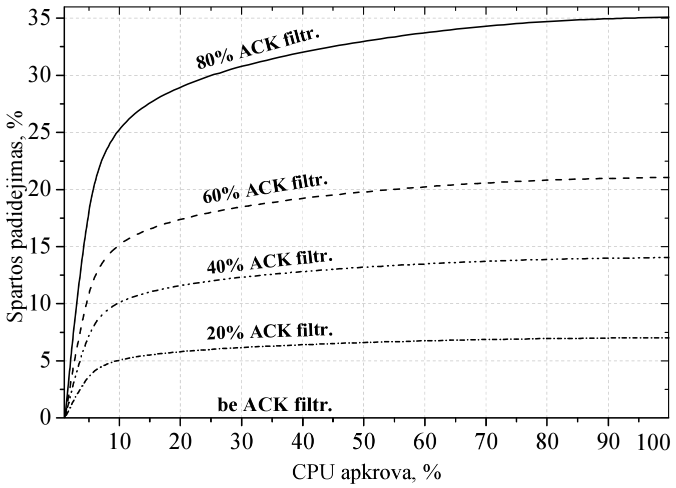
S.1.5 pav. Transporto valdymo protokolo duomenų paketų perdavimo sparta su skirtingomis patvirtinimo paketų filtravimo vertėmis

Iš eksperimento metu surinktų duomenų pateiktų S.1.5 paveiksle pastebima, kad visos tirtos TCP sesijos su skirtingomis ACK filtravimo vertėmis išlaikė vienodą ir pastovų duomenų perdavimo greitį be didesnės įtakos TCP sesijos stabilumui. Maksimali stabili ACK filtravimo vertė buvo pasiekta ties 85 %. Padidinus daugiau buvo stebimas TCP sesijos veikimo nestabilumas ir TCP sesijos nutrūkimai. Iš atliktos statistinės analizės pastebėta, kad didžioji dalis TCP sesijos veikimo sutrikimų buvo sukelti dėl per daug išaugusių paketų užlaikymo (angl. *round-trip time – RTT*), kurios viršijo leistiną paketų persiuntimo laiką (angl. *retransmission timeout – RTO*).

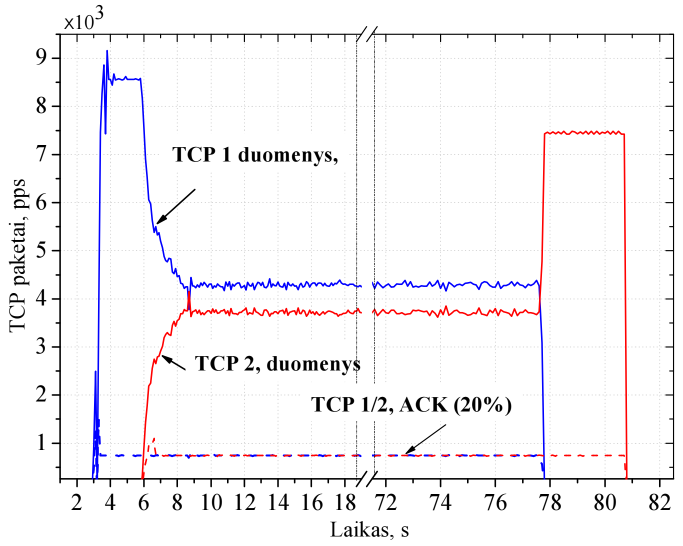
Atlikus SNMP tinklo įrenginių apkrovos analizę (1.6S pav.) buvo ištirta TCP ir ACK paketų daroma įtaką jiems. Iš tyrimo metu gautų duomenų, pateiktų S.1.6 ir S.1.7 paveiksluose, pastebėta, kad tinklo lementų apkrova yra tiesiogiai proporcinga TCP ir ACK paketų skaičiui. Nors ACK ribojimas neryškiai sumažina duomenų srautą tinklo įrenginiuose, tačiau dėl sumažinto ACK paketų skaičiaus, kurį apdoroja tinklo įrenginiai, matomas ryškus apkrovos sumažėjimas. Iš gautų duomenų, pateiktų S.1.7 paveiksle, stebimas tinklo maršruto parinktuvų CPU apkrovos priklausomybė nuo skirtingų ACK filtravimo verčių. Tyrimo metu pateikti tik Cisco 881 ir 1841 tinklo maršrutų parinktuvų CPU apkrovos duomenys, nors panašios apkrovos tendencijos buvo stebimos ir su kitais tinklo įrenginiais. Naudojant 80 % ACK filtravimo vertę maršruto parinktuvų CPU apkrova buvo sumažinta daugiau nei 25 % arba duomenų perdavimo greitaveika padidėjo ~30 %.



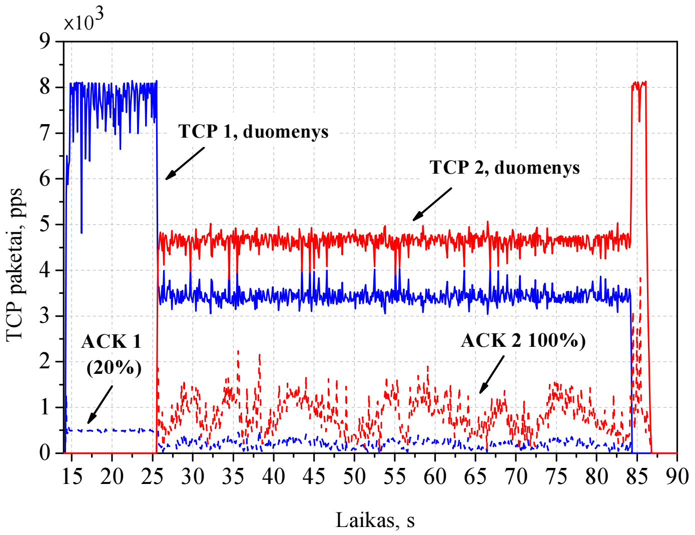
S.1.6 pav. Duomenų siuntimo metu gautų duomenų segmentų eiliškumas per sesijos laiką



S.1.7 pav. Maršruto parinktuvo centrinio procesoriaus apkrova prie skirtingų patvirtinimo paketų filtravimo verčių



S.1.8 pav. Dviejų transporto valdymo protokolo srautų paketų perdavimo sparta su patvirtinimo paketų filtravimu



S.1.9 pav. Dviejų transporto valdymo protokolo srautų su ir be patvirtinimo paketų filtravimo

Atliekant dviejų, vienu metu veikiančių TCP sesijų, su ir be ACK filtravimo, veikimo tyrimą (S1.8 pav.) nustatyta, kad TCP sesijos greitaveikos ir augimo algoritmai, kurie tiesiogiai priklauso nuo gaunamų ACK paketų skaičiaus, mažai priklauso nuo ACK filtravimo ir gali efektyviai konkuruoti ir veikti viename duomenų perdavimo tinkle. Net esant 80 % ACK filtravimo vertei nebuvo pastebėta didesnių TCP sesijos veikimo sutrikimų. Iš tyrimo metu gautų duomenų nustatyta, kad TCP sesija su ir be ACK filtravimo mažai skiriasi. Abi TCP sesijos beveik tolygiai pasidalino duomenų perdavimo kanalą. Nors filtruojama TCP sesija ir gauna mažiau tinklo resursų, tačiau tai yra dalinai dėl didesnio filtruojamos TCP sesijos vėlinimo, kurį sukuria papildomas tinklo įrenginys.

Atlikus pakartotinį eksperimentą, dviejų vienu metu veikiančių TCP sesijų, tik su nedideliu paketų praradimu tinklo įrenginiuose, buvo ištirtas ACK filtravimo ir TCP atsistatymo algoritmų veikimas esant duomenų praradimui (S.1.9 pav.). Eksperimento metu abi TCP sesijos patiria pastovų duomenų siuntimo sutrikimą, po kurio įvyksta TCP atsistatymo procesas. Kaip ankstesniame tyrime (S.1.8 pav.), filtruojama TCP sesija išlieka konkurencinga ir yra tolygi lyginant su nefiltruojama TCP sesija. Nors ACK ribojimas nėra išjungiamas TCP sesijos atsistatymo metu, naudojamo filtravimo metodas (S.1.4 pav.) leidžia nedidelius ACK paketų srauto padidėjimą jų nefiltruojant. Dėl to TCP sesija greičiau atsistato po paketų praradimų (S.1.9 pav.).

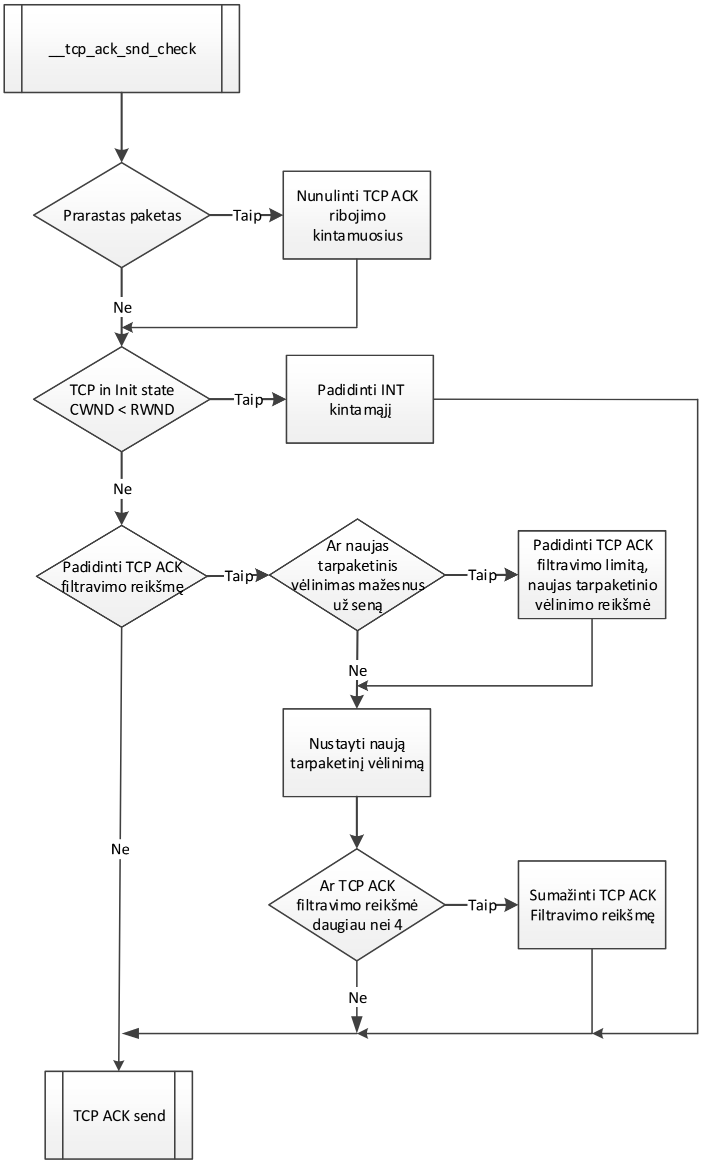
2. Transporto valdymo protokolo patvirtinimo paketų filtravimas Linux branduolyje

Nors ACK filtravimo metodas tinklo įrenginiuose yra paprastas ir lengvai įgyvendinamas, tačiau tai labiau tinka pastovių parametrų tinkluose su žinoma tinklo pralaida ir vėlinimu. Esant kintančiam vėlinimui ar nepastoviems tinklo parametrams ACK ribojimas tinklo įrenginiuose gali veikti neefektyviai ir sukelti papildomus TCP duomenų paketų persiuntimą.

Viena esminių ACK filtravimų problemų yra tai, kad TCP duomenų siuntėjas ir gavėjas nieko nežino apie duomenų perdavimo tinklo parametrus ir kito TCP sesijos nario esamą būseną. TCP duomenų siuntėjas gali būti lėto starto fazėje ar persipildymo vengimo stadijoje. Norint optimaliai išnaudoti ACK filtravimą, būtina leisti TCP duomenų siuntėjui pasiekti savo maksimaliai stabilų išsiuntimo langą (CWND) ir būti persipildymo vengimo būsenoje. Esant šioms sąlygoms galima aktyvuoti ACK filtravimą TCP duomenų gavėjo pusėje. ACK filtravimo reikšmė turi būti didinama palaipsniui Δt periodais. Staigus ACK filtravimo vertės padidėjimas gali sukelti ryškų TCP vėlinimo padidėjimą, dėl ko gali būti viršytas persiuntimo laikas (RTO). Kai kliento nepatvirtintas duomenų kiekis tinkle priartėja prie duomenų tinklo vėlinimo dydžio (BDP) ar TCP duomenų išsiuntimo lango vertės (CWND), stebimas TCP paketų užlaikymo tinkle verčių (RTT) padidėjimas ir duomenų srauto greitaveikos sumažėjimas.

Esamas TCP standartas neturi specifinio lauko ar kito būdo informuoti duomenų siuntėją apie TCP sesijos būseną bei esamą duomenų išsiuntimo lango (CWND) dydį. TCP persipildymo algoritmai skiriasi nuo operacinės sistemos, todėl darbo metu buvo pasiremta RFC standartais, kurie aprašo kaip turėtų kistis išsiuntimo langas (CWND) dydis lėto starto ir persipildymo vengimo metu. Išsiuntimo lango dydis negali didėti daugiau nei 1 leidžiamas siuntėjo maksimalus segmento dydis (angl. *sender maximum segment size – SMSS*) per vieną

TCP paketų apsikeitimo periodą, TCP persipildymo vengimo fazėje (RFC 5681). Ir ne daugiau kaip *SMSS* baitų po kiekvieno gauto ACK lėto starto fazėje. Persipildymo vengimo fazė aprašoma pokyčio funkcija (Bott 2014; Stevens 1997; Floyd *et al.* 2000; Alrshah *et al.* 2014):


__tcp_ack_snd_check
Prarastas paketas
Taip
Nunulinti TCP ACK ribojimo kintamuosius
Ne
TCP in Init state CWND < RWND
Taip
Padidinti INT kintamąjį
Ne
Padidinti TCP ACK filtravimo reikšmę
Taip
Ar naujas tarpaketinis vėlinimas mažesnus už seną
Taip
Padidinti TCP ACK filtravimo limitą, naujas tarpaketinio vėlinimo reikšmė
Ne
Nustayti naują tarpaketinį vėlinimą
Ne
Ar TCP ACK filtravimo reikšmė daugiau nei 4
Taip
Sumažinti TCP ACK Filtravimo reikšmę
Ne
TCP ACK send

S.2.1 pav. Patvirtinimo paketų filtravimo algoritmo įgyvendinimas Linux branduolyje

$$W_{\mathrm{CWND}} = W_{\mathrm{CWND}} + \frac{SMSS \cdot SMSS}{W_{\mathrm{CWND}}}, \tag{S.2.1}$$

čia W_{CWND} – siuntimo lango dydis baitais, o *SMSS* – didžiausias leidžiamas TCP segmento dydis. Linux OS branduolyje lango didėjimo funkcija aprašyta pasikartojančia funkcija, kuri yra tik aproksimacija RFC 5861 funkcijai (Bott 2014; Started *et al.* 2001; Bhuiyan *et al.* 2009; Socolofsky *et al.* 1991).

Žinant W_{CWND} ir ACK generavimo dažnį ir pritaikę Linux OS branduolio W_{CWND} didėjimo funkciją galime surasti ACK paketų skaičių, po kurio TCP siuntėjas pasieks maksimalią leidžiamą W_{CWND} vertę ACK_{N}. Šį pokytį aprašo sekos funkcija:

$$ACK_{\mathrm{N}} = \frac{\left(\frac{W_{\mathrm{CWND}}}{SMSS} + 2\right) \cdot \left(\frac{W_{\mathrm{CWND}}}{SMSS} + 1\right)}{2} - 1 . \tag{S.2.2}$$

Čia ACK_{N} yra ACK paketų skaičius, reikalingas norint pasiekti leidžiamą W_{CWND} vertę.

Žinant ACKN vertę ir W_{CWND} augimo greitį Linux OS branduolyje buvo pasiūlytas naujas ACK filtravimo algoritmas, pateiktas S.2.1 paveiksle. Naujas ACK filtravimo algoritmas dinamiškai kontroliuoja ACK paketų skaičių, pagal kintamus duomenų perdavimo tinklo ir TCP sesijos parametrus, didindamas ar mažindamas ACK filtravimo vertę. Prieš kiekvieną Linux operacinės sistemos ACK siuntimo procesą yra atliekamas papildomas tikrinimo veiksmas, kuris tikrina esamą ACK filtravimo vertę bei atlieka ACK filtravimo vertės korekciją.

Persipildymo vengimo fazė yra labiau tiesinė funkcija ir užtrunka daug ilgiau nei lėto starto fazė. Žinant TCP gavėjo lango dydį RWND (angl. *TCP Receiver Window – RWND*) vertes arba maksimalų leidžiamą TCP serverio išsiuntimo lango (CWND) dydį galima apskaičiuoti laiko periodą, kuris yra būtinas norint pilnai išnaudoti TCP duomenų siuntėjui išsiuntimo langą. Žinant gavėjo lango dydį (RWND) lango dydį, kurį perduodam TCP duomenų siuntėjui, TCP duomenų gavėjas gali surasti ribines ACK filtravimo vertes.

3. Patvirtinimo paketų filtravimo heterogeniniuose tinkluose eksperimentinė patikra

Norint nustatyti Linux branduolyje atliktų pakeitimų įtaką sistemos veikimui bei jos stabilumui, buvo atlikti TCP spartos ir OS stabilumo tyrimai virtualiose sistemose. Tyrimo metu buvo pasirinkta virtuali aplinka dėl lankstesnės testavimo sistemos sukūrimo galimybės. Tai leido gauti daug identiškų sistemų, su minimaliais pokyčiais, naudojant skirtingus Linux OS branduolius.

Pirmo tyrimo metu buvo pasirinkta Linux branduolio virtualizavimo įrankis (angl. *Kernel based Virtual Machine* – KVM) su OpenWRT Linux OS distribucija. Jis naudojamas mažos galios įrenginiuose (Dutt *et al.* 2012; Maxim *et al.* 2012; Rathore *et al.* 2013; Tafa *et al.* 2011).

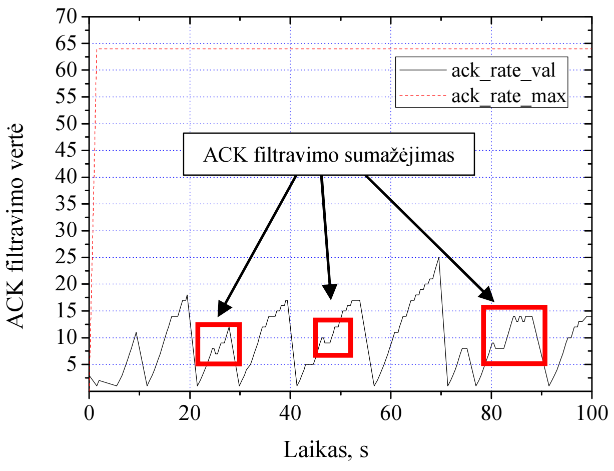
Tyrimo metu Linux virtualizavimo įrankyje buvo išskirtas vienas fizinis centrinis procesorius (CPU) su ribotu sisteminių resursų kiekiu ir 512 MB darbinės atminties. Virtualios sistemos tinklo nustatymams buvo parinktas Intel e1000 tipo valdiklis, su išjungtu TCP automatiniu klaidų skaičiavimu (angl. *Cyclic Redundancy Check – CRC*) tinklo adapteryje.

Patvirtinimo paketų filtravimo Linux branduolyje tyrimui buvo sukurtas duomenų perdavimo tinklas sudarytas iš TCP serverio, veikiančio Linux virtualioje aplinkoje (KVM), ir TCP kliento. Duomenų srauto ir tiriamųjų sistemų stebėjimui bei gautų duomenų analizavimui prie duomenų tinklo buvo prijungtas didesnių sisteminių resursų kompiuteris PK 3.

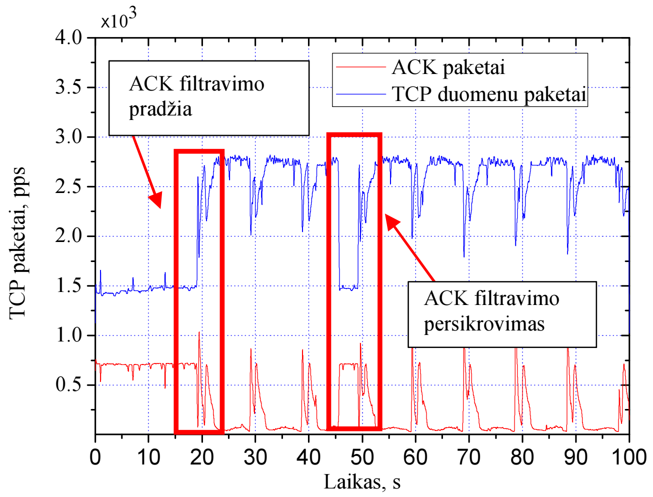
Norint ištirti Linux OS branduolyje įdiegto ACK filtravimo algoritmo veikimą buvo stebima ACK filtravimo vertė *tp->ack_pkt_cnt* ir jos kitimas laike (S.3.1 pav.). Taip pat stebėta ar ACK filtravimo atsitraukimo algoritmas veikia padidėjus TCP duomenų tarppaketiniam vėlinimui (sumažėjus TCP perdavimo greitaveikai). Norint paspartinti ACK filtravimo atsitraukimo algoritmo veikimą ACK filtravimo greitis buvo padidintas naudojant statinę algoritmo augimo *tp->ack_rate_val* vertę.

Iš eksperimento metu gautų duomenų nustatyta, kad ACK filtravimo ir atsitraukimo algoritmas kinta priklausomai nuo tinklo ir TCP sesijos parametrų vertės. ACK filtravimo algoritmo kintamųjų verčių pokytis laike buvo stebimas naudojant <*dmesg*> programą. Tyrimo metu nustatyta, kad pasiūlytas ACK filtravimo augimo ir atsitraukimo metodas veikia ir nesukuria TCP sesijos ar sistemos veikimo nestabilumo.

Lyginat eksperimentiškai gautus rezultatus su nemodifikuota Linux OS sistema buvo matomas ryškus TCP greitaveikos padidėjimas. TCP greitaveika su ACK filtravimu buvo vidutiniškiai 30 % didesnė, o trumpais laiko tarpais ji viršijo 40 % (S.3.2 pav.).



S.3.1 pav. Patvirtinimo paketų ACK filtravimo veikimas Linux branduolyje



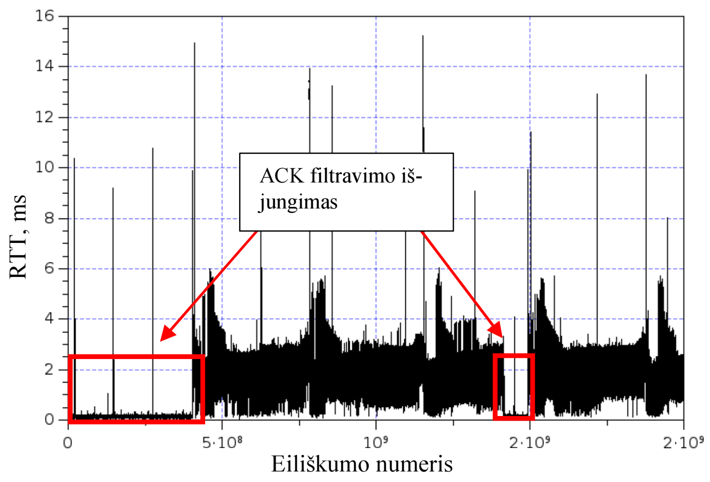
S.3.2 pav. Duomenų siuntimo greitaveikos kitimas įjungus patvirtinimo paketų filtravimo algoritmą

Papildomai atlikus TCP paketų vėlinimo analizę, pastebėta, kad virtualioje aplinkoje modifikuotos Linux OS sistemos vėlinimas yra ženkliai mažesnis lyginant su nemodifikuota ir buvo ~ 10 ms vertės. Tuo metu standartinės sistemos vėlimas buvo 20 ms ir daugiau. Naudojant ACK filtravimą buvo gaunamas iki ~ 50 % vėlinimo sumažėjimas, kas ne tik padidino duomenų perdavimą, tačiau ir sumažino sistemos apkrovimą dėl mažesnio naudojamo TCP duomenų išsiuntimo lango (CWND)vertės.

Norint įvertinti fizinių komponentų (centrinio procesoriaus, operatyvinės atminties greitaveikos, Ethernet tinklo valdiklių) poveikį ACK filtravimo tyrimas buvo atliktas virtualias sistemas pakeičiant fiziniais įrenginiais. Kaip ir ankstesnio eksperimento metu TCP klientas ir TCP serveris buvo sujungti per Ethernet duomenų tinklą, 1Gbit/s spartos jungtimi, o jų perduodami tinklo duomenys buvo nukreipiami analizei į trečia kompiuterį. TCP klientui ir serveriui buvo išjungti automatiniai klaidų aptikimą (CRC) bei paketų apjungimo funkcijos. Vienintelis esminis skirtumas yra tai, kad šio tyrimo metu sisteminis našumas buvo mažesnis serverio pusėje, tai yra TCP perduodamų duomenų greitaveika ribojo tik TCP serverio sisteminiai parametrai ir centrinio procesoriaus pajėgumas. Kadangi didėjant perduodamų duomenų kiekiui proporcingai auga ir gaunamų TCP paketų skaičius, pasiekus tam tikrą ribą yra stebimas siuntimo greitaveikos sustojimas, dėl per didelio centrinio procesoriaus apkrovimo.

Atlikus pakartotinį bandymą su Linux operacinė sistema, kurioje nebuvo naudojamas ACK ribojimas, matyti ženklus išsiunčiamų TCP duomenų greitaveikos padidėjimas. Iš eksperimento gautų duomenų, pateiktų S.3.2 paveiksle pastebima, kad TCP sesijos pradžioje (iki 20 s) yra toks pat TCP išsiunčiamų duomenų greitis. Po ~ 20 s stebimas išsiunčiamų duomenų padidėjimas iki 60 %. Iki ~ 20 s ACK filtravimo algoritmas laukia kol pilnai atsidarys išsiuntimo langas TCP serveryje, po kurio yra aktyvuojamas ACK filtravimo algoritmas TCP duomenų gavėjo pusėje. Tyrimo metu ACK filtravimo algoritmas sustoja kai duomenų siuntėjas vėl pasiekia maksimalią išsiunčiamų duomenų greitaveiką dėl centrinio procesoriaus apkrovos, o padidėjęs vėlinimas priverčia sumažinti ACK filtravimo vertę.

Priešingai nei virtualioje aplinkoje yra stebimas vėlinimo padidėjimas (nuo 0,2 ms iki 2,5 ms), kuris yra susietas su didėjančia *tp->ack_rate_val* verte. Nors atsiradęs vėlinimo padidėjimas yra ženklus, tačiau jis neturi didelės neigiamos įtakos duomenų perdavimo greitaveikai ir TCP sesijos stabilumui (S.3.3 pav.).



S.3.3 pav. Paketų užlaikymo tinkle kitimas duomenų siuntimo metu su patvirtinimo paketų filtravimu

Atlikus eksperimentus su sistema, kurios neriboja sisteminiai resursai, o centrinio procesoriaus našumas yra daug didesnis, buvo gauti duomenys, kurie patvirtino, kad ACK ribojimas neturi didelės įtakos TCP sesijos stabilumui, nors ir buvo gauti mažesni TCP greitaveikos rezultatai. Trumpais laiko tarpais pastebimas neryškus sistemos su TCP filtravimu pranašumas, tačiau vidutinėje greitaveikoje pastebima, kad nemodifikuotas Linux OS branduolys išlaiko geresnę duomenų perdavimo greitaveiką. Tai paaiškinama tuo, kad TCP duomenų gavėjas nesugeba apdoroti gaunamų duomenų ir mažina priėmimo greitaveiką didinant RTT laiką. Tokiomis sąlygomis ACK filtravimo algoritmo veikimas neturi teigiamos įtakos TCP sesijos greitaveikai arba stebimas neryškus jos sumažėjimas.

Bendrosios išvados

Disertacijoje pristatytas transporto valdymo protokolas patvirtinimo paketų ribojimo tyrimas. Gautos išvados:

1. ACK ribojimas, atliekamas tinklo įrenginiuose, neturi neigiamos įtakos TCP sesijos stabilumui ir greitaveikai, o TCP sesija išlieka stabili ACK paketų skaičių sumažinus iki 80 %.

2. Tinklo maršrutų parinktuvų apkrova tiesiogiai priklauso nuo ACK paketų skaičiaus. Augant TCP duomenų perdavimo greitaveikai didėja ir TCP ir ACK paketų skaičius. Su 80 % ACK filtravimu, esant maršrutų parinktuvų centrinių procesorių apkrovai 60 %, našumą galima padidinti iki 32 %.
3. ACK ribojimas, atliekamas tinklo įrenginiuose ir Linux operacinės sistemos branduolyje, gali sėkmingai veikti ir konkuruoti dėl tinklo resursų su kitomis TCP sesijomis, veikiančiomis tame pačiame duomenų perdavimo kanale kaip ir nefiltruojama TCP sesija nors buvo pastebėtas ne visai tolygus duomenų kanalo pasidalinimas.
4. ACK ribojimas, atliekamas tinklo įrenginiuose, neturi neigiamos įtakos TCP sesijos stabilumui ir veikimui esant pastovioms tinklo sąlygoms. Tačiau kintant TCP sesijos parametrams ir paketų užlaikymo laikams (RTT) būtinas ACK filtravimo verčių perskaičiavimas.
5. Naudojant ACK filtravimą IEEE802.11a nevienalyčiuose duomenų perdavimo tinkluose ACK skaičių galima sumažinti iki 30 %, taip sumažindami duomenų perdavimo tinklo įrangos apkrovą ir padidina duomenų tinklo pralaidumą iki 10 %~~.~~
6. Dinaminio ACK filtravimo algoritmo įdiegimas Linux operacinės sistemos branduolyje neturi neigiamos įtakos TCP sesijų stabilumui bei sistemos veikimui, kai yra išlaikomos ribinės veikimo sąlygos ir TCP priėmimo lango vertė yra didesnė už tinklo duomenų vėlinimo rodiklį.
7. Esant ribinėms ACK filtravimo vertėms ar padidėjusiai sistemos apkrovai stebimas ryškus TCP RTT vėlinimo deviacijos padidėjimas, kuris gali viršyti t_{RTO} laiką ir sukelti TCP duomenų paketų persiuntimą ir siuntimo laiko pailgėjimą.
8. Patvirtinimo paketų ribojimas tinklo įrenginiuose ir Linux OS branduolyje leidžia padidinti TCP duomenų pralaidumą iki 50% ir sumažinti nereikalingą centrinio procesoriaus apkrovą iki 32% tinklo įrenginiuose. Pasirinktos ACK filtravimo vertės neturi viršyti nustatytų ribinių verčių, nes tai gali sukelti nereikalingą TCP duomenų paketų persiuntimo laiką ir TCP našumo sumažėjimą.

Annexes[1]

Annex A. Created Intellectual Property Cores

Annex B. The Co-authors’ Agreement to Present Publications Material in the Dissertation

Annex C. The Copies of Scientific Publications by the Author on the Topic of the Dissertation

[1]The annexes are supplied in the enclosed compact disc.

Arūnas STATKUS

PERFORMANCE INVESTIGATION OF DATA PACKETS
TRANSPORT CONTROL PROTOCOL
IN HETEROGENEOUS NETWORKS

Doctoral Dissertation

Technological Sciences,
Electrical and Electronics Engineering (01T)

Arūnas STATKUS

DUOMENŲ PAKETŲ TRANSPORTO PROTOKOLO
HETEROGENINIUOSE TINKLUOSE NAŠUMO TYRIMAS

Daktaro disertacija

Technologijos mokslai,
elektros ir elektronikos inžinerija (01T)

2017 07 18. 10,5 sp. l. Tiražas 20 egz.
Vilniaus Gedimino technikos universiteto
leidykla „Technika“,
Saulėtekio al. 11, 10223 Vilnius,
http://leidykla.vgtu.lt
Spausdino BĮ UAB „Baltijos kopija“
Kareivių g. 13B, 09109 Vilnius