electronics

MDPI

Article

TCP Acknowledgment Optimization in Low Power and Embedded Devices

Arūnas Statkus [1], Šarūnas Paulikas [1] and Audrius Krukonis [2,*]

1 Department of Computer Science and Communications Technologies, Vilnius Gediminas Technical University, 03227 Vilnius, Lithuania; arunasstatkus@vilniustech.lt (A.S.); sarunas.paulikas@vilniustech.lt (Š.P.)
2 Department of Electronic Systems, Vilnius Gediminas Technical University, 03227 Vilnius, Lithuania
* Correspondence: audrius.krukonis@vilniustech.lt

Abstract: Paper investigates transport control protocol (TCP) acknowledgment (ACK) optimization in low power or embedded devices to improve their performance on high-speed links by limiting the ACK rate. Today the dominant protocol for interconnecting network devices is the TCP and it has a great influence on the entire network operation if the processing power of network devices is exhausted to the processing data from the TCP stack. Therefore, on high-speed not congested networks the bottleneck is no longer the network link but low-processing power network devices. A new ACK optimization algorithm has been developed and implemented in the Linux kernel. Proposed TCP stack modification minimizes the unneeded technical expenditure from TCP flow by reducing the number of ACKs. The results of performed experiments show that TCP ACK rate limiting leads to the noticeable decrease of CPU utilization on low power devices and an increase of TCP session throughput but does not impact other TCP QoS parameters, such as session stability, flow control, connection management, congestion control or compromises link security. Therefore, more resources of the low-power network devices could be allocated for high-speed data transfer.

Keywords: transport control protocol; acknowledgement optimization; ACK filtering; Linux TCP stack

check for updates

Citation: Statkus, A.; Paulikas, Š.; Krukonis, A. TCP Acknowledgment Optimization in Low Power and Embedded Devices. *Electronics* **2021**, *10*, 639. https://doi.org/10.3390/electronics10060639

Academic Editor: Seung-Hoon Hwang

Received: 25 January 2021
Accepted: 4 March 2021
Published: 10 March 2021

1. Introduction

The expansion of the Internet contributed to an exponential increase in network applications and, as a consequence, to increase the speed and productivity requirements of embedded and mobile devices. This rises specific requirements for the operating system and its networking module. Today a major number of the Internet of Things (IoT) and embedded devices use the Linux operating system with the transport control protocol (TCP)/IP stack implemented in Linux kernel, while application-layer protocols are implemented in user space (HTTP, FTP, SSH, etc.). Such a choice of operating system liberates manufacturers from the long and expensive development process and shortens the hardware and software development lifecycle of new Internet-connected equipment [1]. Today the dominant protocol for interconnecting network devices is the TCP and it has a great influence on the entire network operation if the processing power of network devices is exhausted to the processing data from the TCP stack.

The main uniqueness (and in some cases problem) in the default TCP protocol implementation is that the TCP protocol always tries to have the maximum possible utilization of the data transmission link. On high-speed uncongested networks, the bottleneck is no longer the network link but low-performance network devices and their diversity. The transmission of TCP packets consumes all CPU resources of these devices and reduces TCP performance due to increased round-trip time (RTT), jitter, and thus an unstable TCP session.

The optimization of TCP data processing is also important to servers and server farms. It can boost data center performance and decrease power consumption due to reduced CPU load [2].

The aim of the article is to propose and analyze the new TCP ACK rate limiting algorithm implementation in the Linux kernel.

The paper is structured into seven sections. Section 2 gives an overview of TCP performance optimization in related works. In Section 3 we provided the detailed workings of the TCP acknowledgment mechanism in the Linux kernel. TCP ACK rate limiting implementation on Physical Machines and comparison results with and without TCP ACK rate limiting presented in Section 4. Limits of ACK rate-limiting in virtual machines, when particular hardware characteristics are eliminated, investigated in Section 5. Application scenarios and CPU performance of the modified system are addressed in Section 6. Finally, the conclusions are reported in Section 7.

2. Related Work

As defined in RFC 793, TCP is a connection-oriented, end-to-end reliable data communication protocol intended for setting up the reliable link between pairs of processes of network nodes of separate but interconnected data communication networks. It controls the data flow on the network by means of congestion avoidance and acknowledgment (ACK). The critical disadvantage of TCP standard implementation as defined in the Internet Standards (RFCs) of Internet Engineering Task Force (IETF), specifically (RFC 3782), is efficient at low-speed networks ($\leq$100 Mbps) however unsuitable for high throughput links because of sluggish utilization of channel capacity (RFC 5681), and inadequate estimation of the channel's maximum throughput [3]. This is due to congestion window growth at the beginning of the TCP session is a square function of the link delay (RFC 2581): a high delay makes the congestion window size increase slowly. To overcome this disadvantage, many TCP data flow control mechanisms have been introduced. Most of them are focused to handle large amounts of data transfer, e.g., HSTCP, Fast, Cubic, STCP [4]. All these TCP variants dependent on the received data packet acknowledgment rate (ACK) at the network end-nodes and congestion window [5]. Current Linux OS uses the CUBIC congestion control algorithm, which is suitable for high bandwidth networks with high latency, has the cube-root congestion window growth function and low dependence on channels delay at the beginning of TCP session.

The size of the congestion window of a TCP session is calculated in real-time and relays on the ACK flow [4]. The TCP data sending node can increase the sending data rate after received ACK. Therefore, for effective data flow control the unobstructed exchange of ACK is very important, because any loss of ACK reduces TCP efficiency [6,7]. As a result, the utilization of network capacity is directly related to the ACK rate. If the amount of TCP packets on the network grows, the number of ACKs on the network grows as well. Thus, high-speed networks suffer from the increase in technological expenditures related to ACK transmission [8].

TCP performance can also deteriorate while exchanging short packets [9,10]. This is caused by technological expenditures [8] and CPU overload [11].

The latter concern is described in [12], and it happens because the CPU load of the network device depends on the number of TCP packets to process but not on packet size. Such condition is more frequently observed on high-speed networks [11], since the increase of data throughput increases the number of the TCP ACK packets on the network [13]. Therefore, handling the heavy traffic of short TCP packets requires more powerful network devices.

One way to optimize the TCP performance on high-speed networks or embedded devices is to reduce the number of ACKs of the TCP session [14]. This could be achieved by limiting the ACK rate [15], which is often applied on wireless networks [6,7], and could be implemented in Linux kernel as TCP stack function.

Other related works investigate TCP performance optimizations by applying various congestion control algorithms. Authors of [16] investigate optimization possibility by dynamically changing the initial TCP window for short and long TCP flows. The authors of [17] propose an algorithm for bandwidth and the RRT estimation, which leads to improved throughput in the network. TCP performance optimizations are also important in the distributed sensor networks, wireless IoT devices because they generate a large amount of data. Researchers try to optimize performance in these networks not only by improving congestion control [18] but also by introducing negative acknowledgment (NACK) into generic TCP functionality [19] or proposing parallel transmission hybrid methods [20].

3. TCP Acknowledgment Mechanism in Linux Kernel

The Linux is an event-driven operating system (OS) that responds to external and internal events or occurring interrupts [21]. The interrupt has a unique code that indicates the OS to execute an appropriate service function. If the network interface card (NIC) of the Ethernet network device receives a TCP packet, an interrupt occurs and the OS forwards processing of the received data to the NIC driver [22]. The NIC driver puts the data packet into the buffer and checks its integrity [23]. The validated packet is transferred to the OS main memory that was allocated to store the NIC data. If the TCP packets overwhelm the network device or the NIC buffer is full, the NIC drops the received packet and protects the network device from rendering unresponsive [24].

After the TCP packet has been processed and validated, the kernel calls the function to send an ACK [21]. The function sends the TCP ACK basing on such conditions:

1. The received TCP packet or unacknowledged packet is larger than the defined maximum segment size according to RFC 1122 or STD003 specification or time after last ACK has been sent is more than 500 ms (OS depended);
2. The TCP receive window or device NIC buffer size is greater than the negotiated TCP receive window or NIC buffer size;
3. Some data must be immediately piggybacked to the TCP data sender;
4. The network device receives an out-of-order TCP packet.

Otherwise, the TCP ACK sending can be postponed. However, in default, the Linux kernel implementation ACK is sent after two TCP data packets are successfully received.

Therefore, the TCP ACK rate reduction by acknowledging every third, fourth, etc. TCP data packet can notably lower network utilization and spare additional CPU resources for TCP packet processing in embedded or other network devices with limited computing power.

In the Linux kernel, triggering of ACK depends on TCP delayed ACK parameter `icsk_ack.rcv_mss` and if there is sufficient space in the NIC buffer. This applies only if there is no packet loss or reordering during the TCP session. Therefore, the most convenient way to control the ACK rate by factor is through `icsk_ack.rcv_mss`. According to the TCP specification (RFC 1122 and STD003) an ACK should be generated for at least every second full-sized segment. In the Linux kernel, the maximum received segment size (MSS) of the TCP packets after which ACK should be sent is defined in `icsk_ack.rcv_mss` and is used for delayed ACK decision. By controlling the size of `icsk_ack.rcv_mss` we can increase or decrease the frequency of the ACK. To use this variable for the ACK rate control and to minimize alternation of Linux kernel code, a supplementary variable `ack_rate_val` must be introduced. The only negative impact of using additional variables is that the kernel has to dedicate more device resources for their storage and recalculation while processing the received TCP packets.

The main problem with reducing ACK and TCP, in general, is that the OS is not aware of network the condition (throughput, jitter, delay, transport technologies) and its variation. Therefore, TCP must adapt to these changes. The TCP receiver does not know the TCP state of the data sender CP (slow start, congestion avoidance, fast recovery, etc.). If the reduction of ACK starts too soon, it will have a harmful effect on the TCP session throughput, if it started too late, no gain on TCP session throughput or system performance will be attained.

Such an ACK optimization concept has been proposed by authors in [25] and summarized in [26] as shown in Figure 1.

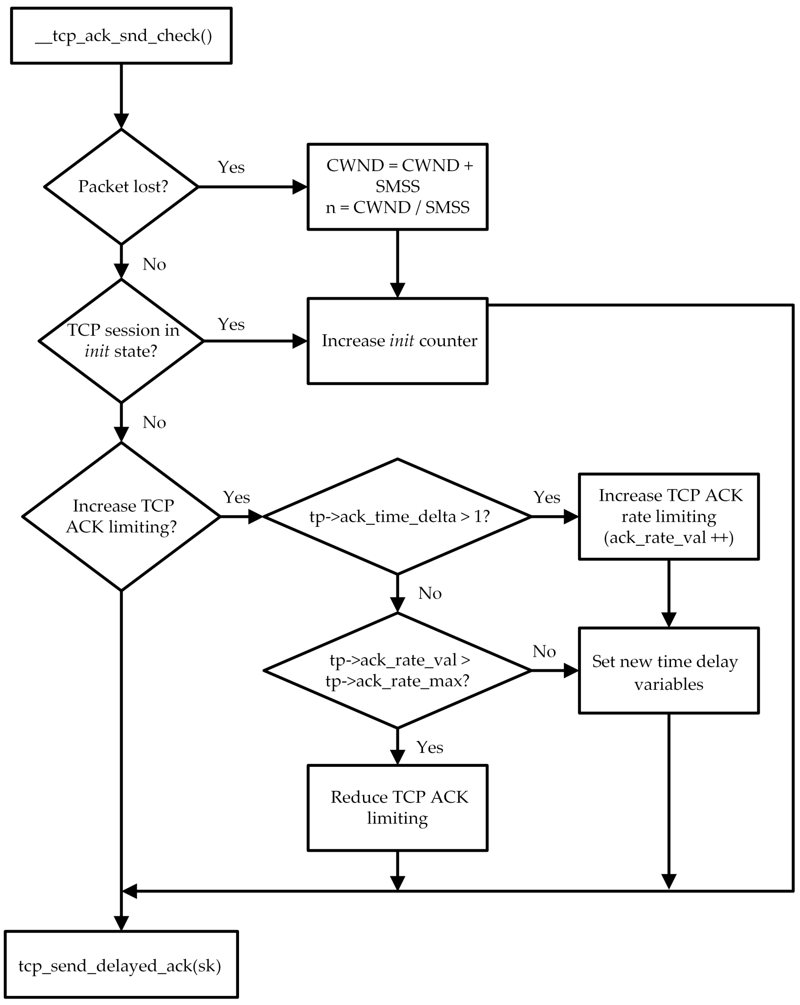

Figure 1. Acknowledgment rate limiting algorithm.

In the beginning, to avoid the activation of ACK rate limiting or to disable it after data packet loss or retransmission, the kernel tracks the TCP session status. The ACK limiting will not start until the TCP initialization state is passed, because of the time needed to check if the TCP session is stable and no packed drops occurred. If a network link is congested, the kernel will not initialize ACK rate limiting and do not allow any reduction of ACK packets.

Afterward, the ACK rate limiting is activated if the client-side TCP congestion window size is maximum, the TCP session has been stable and the throughput of the current TCP session does not decrease.

However, if the TCP session delay increases or its throughput decreases after the last increase of ACK rate limiting, the algorithm falls back to the previous value. The ACK limiting value is at its maximum and additional ACK rate reduction could lead to TCP performance degradation or jittering problems that will be encountered due to the too-low ACK rate. Therefore, the TCP receives window and TCP congestion window sizes must be increased to achieve the same TCP throughput. The increase of the jitter is the side effect of the too-small TCP receiver window caused by too low ACK rate. The TCP data sender buffer would be full and stop data sending until it receives the ACK from the receiver. After receiving an ACK, which acknowledges all the data in the TCP sender buffer, all awaiting TCP packets in the TCP sender buffer are transmitted and that causes packet burst or jitter.

The proposed ACK rate limiting and TCP session state tracking implemented in Linux kernel `__tcp_ack_snd_check()` function (Figure 2).

```
4796 static void __tcp_ack_snd_check(struct sock *sk, int ofo_possible)
4797 {
4798 struct tcp_sock *tp = tcp_sk(sk);
4799
4800 if ( ofo_possible && skb_peek(&tp->out_of_order_queue) ) {
4801   tp->init_pkt_cnt = 0;
4802   tp->ack_pkt_cnt = 0;
4803   tp->ack_time_t1 = 0;
4804   tp->ack_rate_val = 0;
4805   tp->ack_rate_max=tp->rcv_wnd/(3*inet_csk(sk)->icsk_ack.rcv_mss);
4806   if (tp->ack_rate_max > 64 ) {
4808   tp->ack_rate_max = 64;}
4809   tp->ack_time_t2 = 0;
4810   tp->ack_time_delta = 0;
4811   tp->tcp_delayed_snd = 0;
4812}
4813 if ( tp->init_pkt_cnt < tp->start_ack_lmt ) {
4814   tp->init_pkt_cnt += 1 ;
4815 } else {
4816   if (tp->ack_pkt_cnt > tp->start_ack_lmt2 ) {
4817     tp->ack_pkt_cnt = 0;
4818     tp->ack_time_t2 = jiffies;
4819    if (tp->ack_time_t2 - tp->ack_time_t1 <= tp->ack_time_delta  || \
           tp->ack_time_t1 == 0  && tp->ack_rate_max > 64 ) {
4820     tp->ack_rate_val+=1;
4821     tp->ack_time_delta = tp->ack_time_t2 - tp->ack_time_t1;
4822     tp->ack_time_t1 = tp->ack_time_t2 ;
4823     } else {
4824       tp->ack_time_delta = tp->ack_time_t2 - tp->ack_time_t1;
4825       tp->ack_time_t1 = tp->ack_time_t2;
4826       if ( tp->ack_rate_val > 4) {
4827         tp->ack_rate_val-=1;
4828  }}}
4829 tp->ack_pkt_cnt +=1;
4830 }
4831 if (((tp->rcv_nxt - tp->rcv_wup) > \
          (tp->ack_rate_val* inet_csk(sk)->icsk_ack.rcv_mss)  && \
          __tcp_select_window(sk) >= tp->rcv_wnd) || \
4832  tcp_in_quickack_mode(sk) ||
4833 (ofo_possible && skb_peek(&tp->out_of_order_queue))) {
4834  tcp_send_ack(sk);
4835 } else {
4836   tcp_send_delayed_ack(sk);
4837 }}
```

Figure 2. Modified Linux kernel `tcp_input.c` source code of `__tcp_ack_snd_check()` for checking TCP session state.

The ACK rate limiting is disabled if a packet lost or out of order packet is received, the ACK rate-controlling variables are reset, and the TCP session recovers using fast retransmit mode (Figure 2, Lines 4801–4811). In the initial TCP session state or after out of order packet is received (indicated by `ofo_possible`) the function resets `tp->init_pkt_cnt`, which controls the start of ACK rate limiting. Further, `tp->init_pkt_cnt` is compared to `tp->start_ack_lmt`, which is obtained from `tcp_ack_limit_init()` (Figure 3). If the TCP session is in the initial state, it goes through the slow start phase. During this phase, ACK rate limiting is disabled and the maximum allowed ACK rate, `tp->ack_rate_max`, is calculated as ratio RWND/3 MSS.

```
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;
}
```

Figure 3. Changes in Linux kernel `tcp.h` source code for a new `tcp_ack_limit_init()`.

Consequently, the function increments `tp->init_pkt_cnt` and checks TCP session state. When `tp->init_pkt_cnt` becomes greater than `start_ack_lmt`, the function can proceed (Figure 2, Line 4813) and start ACK rate limiting. At first, the function must record the current state of TCP session, and after it goes into the loop once more to take time measurements. It increases `tp->ack_pkt_cnt` until it reaches `tp->start_ack_lmt2`; the value is increased after every TCP data segment has been received. At the start of TCP session, `tp->start_ack_lmt2` is set to the value of `tp->start_ack_lmt` and represents TCP data segment after which ACK rate limiting can start. If the time needed to receive the number of TCP segments stored in `tp->start_ack_lmt2` is smaller or equal to previous time measurements, `tp->tcp_ack_rate_val` can be increased, yet `tp->tcp_ack_rate_val` must be decreased, but not less than the initial one (Figure 2, Line 4827). It prevents the variable from becoming zero, which will start the generation of ACKs for every received data segment.

4. TCP ACK Rate Limiting on Physical Machine

The TCP session parameters on high-speed, not congested networks, such as RTT, throughput, etc., mostly depend on network hardware and the diversity of its components. To test the impact of ACK rate limiting on TCP session performance and eliminate the effects of other factors the proposed ACK rate limiting algorithm has been experimentally tested using the setup shown in Figure 4.

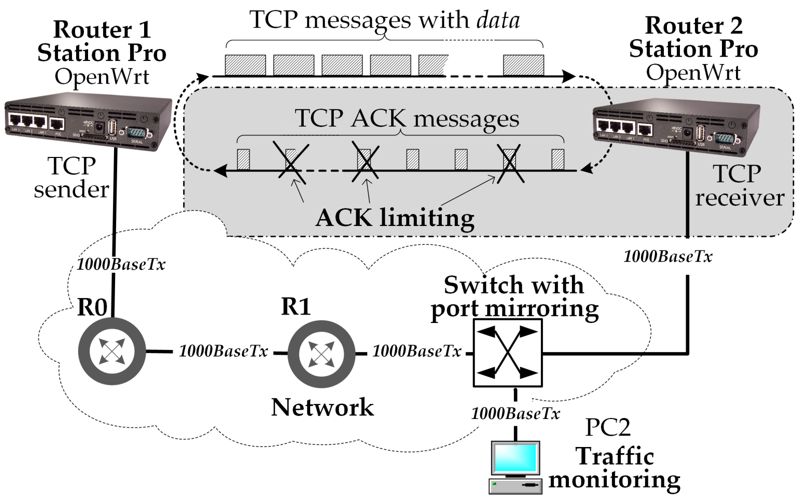

Figure 4. Test setup of transport control protocol (TCP) acknowledgment (ACK) rate limiting on physical machine.

The test-bed consists of two identical Ubiquiti RouterStationPro routers with installed OpenRTW image build for MIPS-based CPU architecture systems and computer for network traffic capture and inspection. Both routers have MIPS-based architecture CPU and amount of RAM limited to 128 MB. The ACK rate limiting has been enabled on router 1. The router 2 without ACK limiting has been used for comparison. The routers were interconnected using 1 Gbps links, TCP offloading was disabled on NICs of both sides. The throughput of 100 s duration TCP session has been measured using the Linux `iperf` tool that is commonly used for network performance measurements and generation of TCP data streams. The same tests have been performed on both routers with identical settings. The throughput measurement data have been acquired using Linux `dmesg` tool by outputting values of TCP ACK limiting variables via the kernel messaging subsystem.

The tests have been conducted imitating a real-world situation of TCP data transmission from the TCP client to the server. In such a configuration, the TCP packets sending device or the TCP client is the data processing bottleneck, because of few CPU resources compared to the server. Therefore, the increase of the TCP session throughput by limiting the ACK rate is because of reduced data processing on the TCP client side. The test results of TCP throughput are shown in Figures 5 and 6.

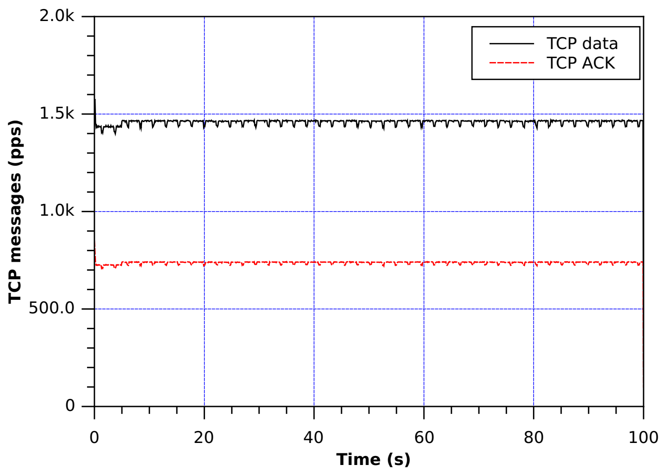

Figure 5. TCP data and acknowledgment packet rate on router without ACK rate limiting.

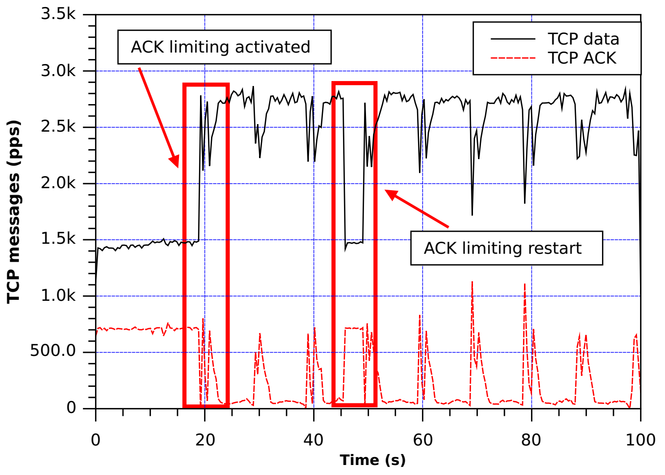

Figure 6. TCP data and acknowledgment packet rate on router with ACK rate limiting.

The TCP session throughput and the ACK rate on the router without the ACK rate limiting are uniform during the whole test period (Figure 5). However, on the router with the ACK rate-limiting (Figure 6) TCP session throughput starts to increase after approximately 20 s from the beginning of data transmission. At this point, the ACK rate limiting algorithm starts to reduce ACKs. The upper ACK rate limit has been set to 2000 packets and after reaching it the ACK rate is lowered.

At the same time, the ACK rate rapidly reduces until the ACK rate limiting algorithm detects the increase of RTT of the current TCP session (Figure 6). After approximately 45 s ACK rate limiting algorithm stops reducing ACKs after multiple drops of TCP packets.

The TCP session goes into a slow-start phase and the ACK rate limiting algorithm resets ACK rate limiting variables. After approximately 51 s, the ACK rate limiting restarts.

The performed tests indicate an increase of TCP session up to 60% on the router with ACK rate limiting enabled. However, on different devices with different hardware configurations, the increase of TCP session throughput can be influenced by many factors, especially if TCP offloading is turned on and the data transmitting and receiving devices have unique settings.

Figures 5 and 6 show that TCP session throughput on the router with modified Linux kernel TCP stack for ACK rate limiting is identical to TCP throughput of the router with unmodified kernel from the beginning of TCP sessions and until ACK rate limiting starts at 19 s. The spikes and peaks are visible in Figure 6 caused by smaller `ack_pkt_cnt` used for testing of ACK rate limiting algorithm to observe a faster ACK rate-limiting function grow and simulate the situation with small TCP RWND on the receiver side.

This indicates that the Linux kernel TCP stack modifications do not degrade the overall TCP session performance and successfully functions even after packet drops.

In order to investigate the effect of ACK rate limiting on the TCP link delay TCP session, RTT (Figures 7 and 8) has been examined.

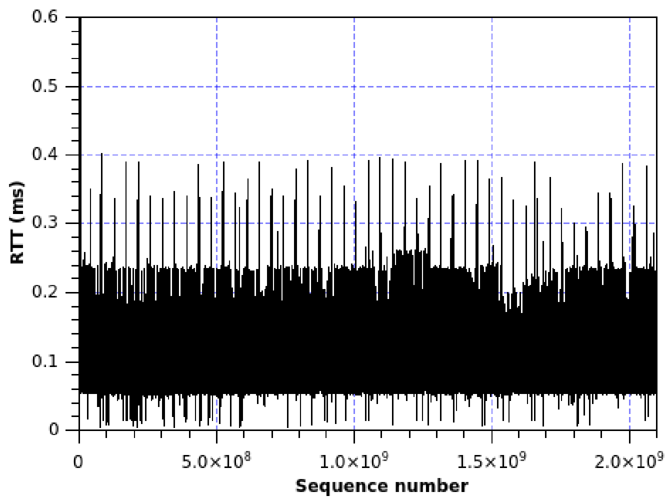

Figure 7. RTT deviation on router with an unmodified Linux kernel.

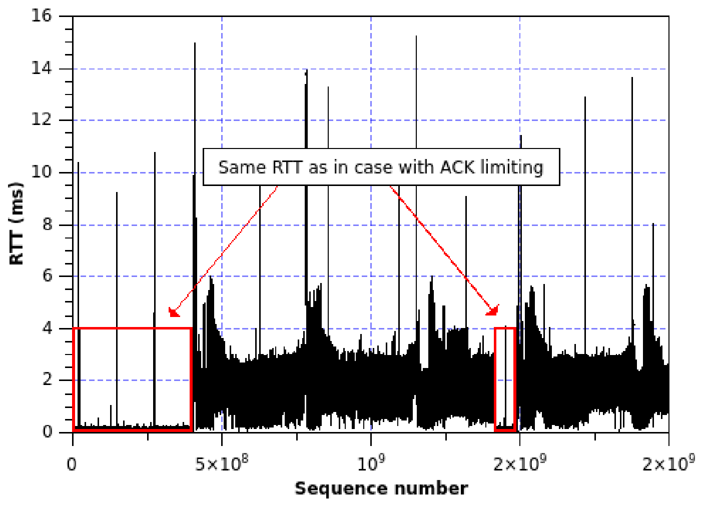

Figure 8. RTT deviation on router with a modified Linux kernel.

By comparing results in Figures 7 and 8, an increase of RTT have been observed on the router with the modified kernel after ACK rate limiting has been activated and until a packet drop occurred. After that, RTT decreases and becomes the same as on the router with the unmodified kernel. Such behavior can be explained by the fact that during the test, the TCP data sending system does not hav enough processing power to handle increased data traffic after ACK rate limiting is enabled. However, even in such a situation, the TCP data sender benefits from the ACK rate limiting because it still requires fewer recourses to process TCP data.

5. TCP ACK Limiting on Virtual Machine

To explore the limits of the application of ACK rate limiting and eliminate the impact of particular hardware characteristics, similar tests have been performed on virtual machines running on KVM virtualization software with the default configuration. Virtual routers have been implemented using an open-source OpenWRT Linux distribution because it is commonly installed on embedded or low computing power devices [27]. Router 1 (Figure 9) has improved the Linux kernel with implemented ACK rate limiting.

The virtualization has been implemented using the computer with one $\times$ 86 architecture CPU, and 512 MB RAM was dedicated for each OpenWRT router. On virtual machines, the default NIC device virtio, that implements packet segmentation in the NIC driver, has been replaced by Intel e1000 connected to the physical NIC of the KVM server using the network bridge. Additionally, on NICs of KVM server and TCP devices, the TCP offloading features have been disabled [28,29]. The full setup of the performed tests is shown in Figure 9.

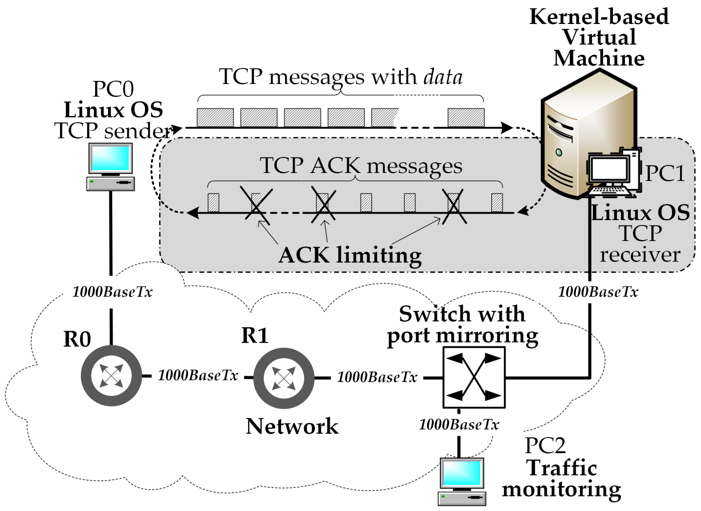

Figure 9. Test setup of the TCP ACK rate limiting on the virtual machine.

The network has been built from three interconnected computers using 1 Gbps Ethernet connections. The TCP packet capture and analysis have been performed on a separate computer. To obtain more accurate results, TCP offloading features have been turned off on all computers' physical NICs. This is done to rid-off TCP offloading influence on TCP session behavior and limit factors impacting TCP session characteristics to the KVM server CPU computing power.

The proposed Linux kernel TCP stack modification for the ACK rate limiting performs two main functions. The first function is to set values of ACK rate limiting variables during runtime based on the characteristics of the TCP packet flow over a defined period. The second function decides if ACK must be sent. It is important to note that ACK could be sent even without reaching the specified ACK rate limit if the TCP ACK timeout expires.

To evaluate the ACK rate limiting algorithm and, in more detail, analyzing TCP session behavior during normal operation or after packet drop occurs, the values of variables responsible for ACK rate limiting have been outputted using Linux `dmesg` tool. For this purpose, the additional printing commands have been introduced in the ACK rate limiting implementation code.

The change of the values of maximum ACK rate limiting `ack_rate_max` and current ACK rate `ack_rate_val` variables is shown in Figure 10.

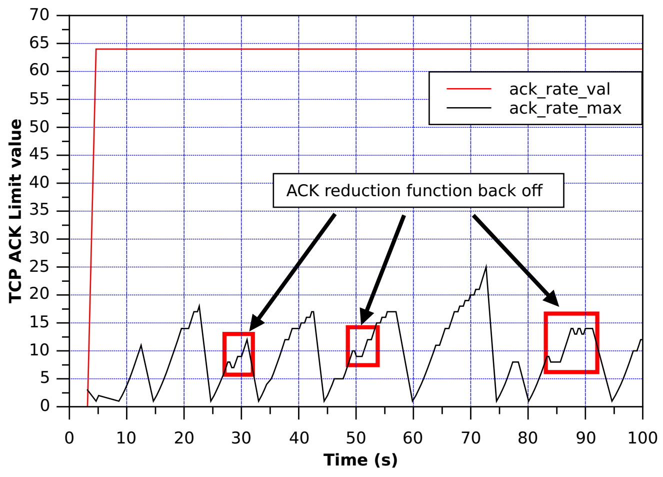

Figure 10. The change of `ack_rate_max` and `ack_rate_val` during TCP session.

It can be seen that `ack_rate_val` starts to increase after approximately 10 s of the beginning of the TCP session, its value changes from zero and reaches its maximum at approximately 20–25 s. At approximately 30 s, the algorithm notices a degradation of TCP throughput and the first decrease off `ack_rate_val` is observed, which indicates that ACK rate limiting is functioning. At the same time, the decrease of `ack_rate_val` is again observed at approximately 50 s and 80 s, which indicates that the TCP packets transmission rate has been reduced. Additionally, TCP data and ACK rates have been monitored with wireshark to analyze the impact of the ACK limiting algorithm on overall TCP session throughput.

During the performed tests smaller `ack_pkt_cnt` have been chosen. This variable sets the number of TCP packets that must be transmitted before the ACK rate could be reduced. Small `ack_pkt_cnt` makes the ACK rate limiting algorithm responds more rapidly and increases `ack_rate_val` more frequently to examine the TCP stability in short TCP sessions.

To analyze the impact of the ACK rate limiting algorithm on TCP sessions throughput the identical tests have been performed on identical `OpenWRT` images with the same network configurations. The one has been compiled with, and the other without, Linux kernel TCP stack modification. The TCP packet stream at the maximum rate for 100 s has been generated using `iperf` tool. The network traffic has been captured and analyzed with `wireshark` tool running on a separate network traffic monitoring PC (Figure 9).

The TCP sessions throughputs with and without ACK rate limiting are shown in Figures 11 and 12, and the corresponding TCP sessions RTT measurements in Figures 13 and 14. The decrease of RTT in a virtual environment can be explained by the reduced packet per second (PPS) rate. With enabled ACK filtering, we can reduce packet rate, this reduces the interrupt rate in VM system, this can have a significant impact on system performance and latency [30].

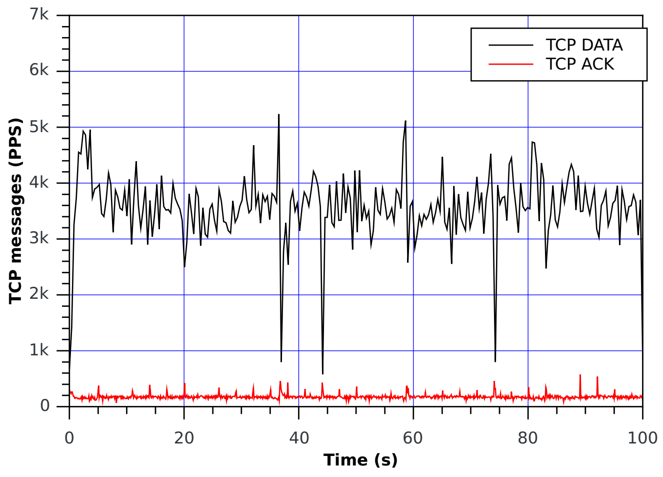

Figure 11. TCP session throughput on router with an unmodified Linux kernel.

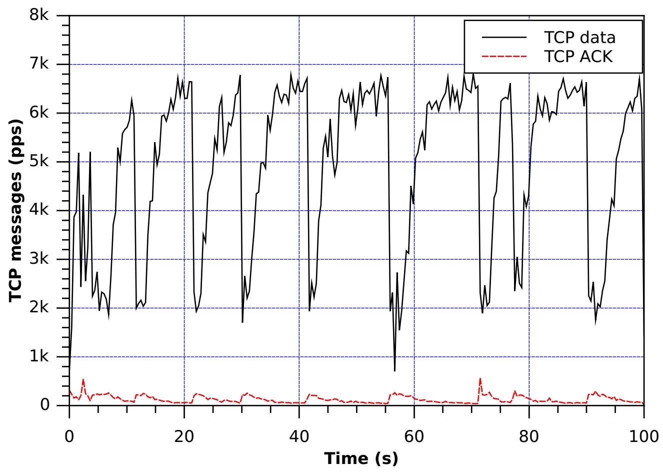

Figure 12. TCP session throughput on router with a modified Linux kernel.

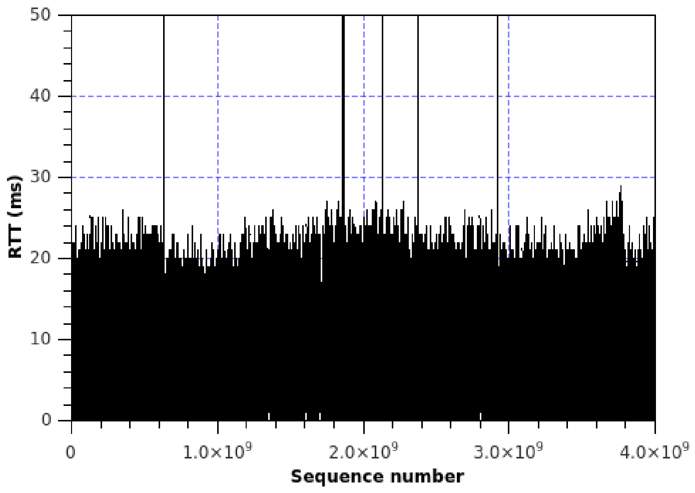

Figure 13. TCP session RTT on router with an unmodified Linux kernel.

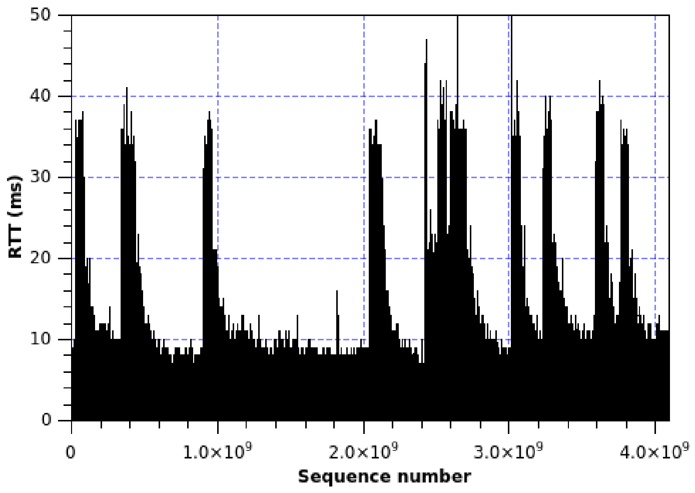

Figure 14. TCP session RTT on router with a modified Linux kernel.

From TCP sessions throughput results (Figures 11 and 12), it is seen that in a virtual environment the router with ACK rate limiting performs better, exhibits more uniform data rates over a given period and TCP sessions throughput is up to 30% higher. The router with a modified Linux kernel has almost twice smaller RTT compared to the router with the unmodified kernel as shown in Figures 13 and 14, a RTT histograms have been calculated (see Figures 15 and 16).

In Figure 12, noticeable short drops of TCP session throughput of 10–20 s period could be observed. It correlates with peaks of RTT of the TCP session (Figure 14) and is due to excessive ACK rate reduction. Such an undesirable effect can be eliminated by lowering the rate of the ACK reduction in the ACK rate limiting algorithm. From obtained TCP sessions RTT data given in Figures 13 and 14, a RTT histograms have been calculated (see Figures 14 and 15). It can be observed that in the router with ACK, rate limiting-enabled maximum values of the RTT histogram are shifted towards lower RTT and are more concentrated between 5–10 ms. The maximum values of RTT histogram on the router without ACK rate limiting are spread more widely, between 10–20 ms. The shorter RTT is important and attractive to multimedia and real-time applications.

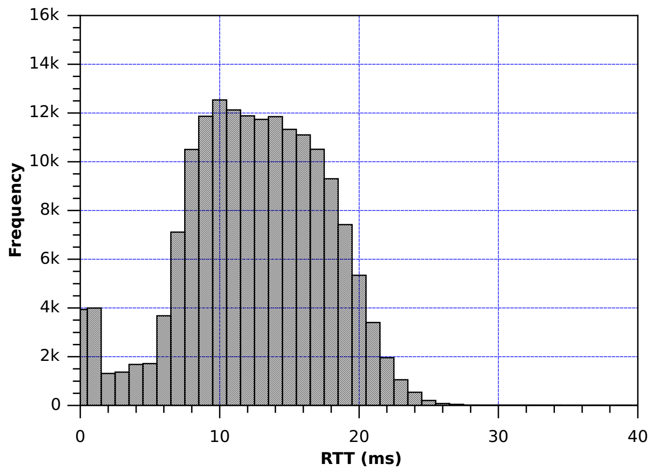

Figure 15. TCP session RTT histogram on router with an unmodified Linux kernel.

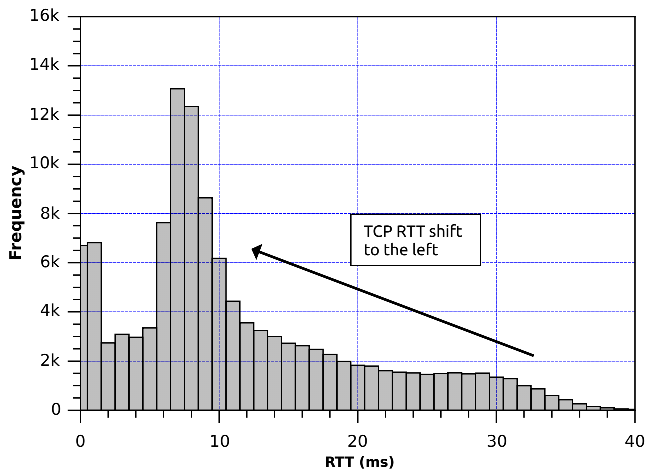

Figure 16. TCP session RTT histogram on router with a modified Linux kernel.

The recurrent RTT peaks observed in the Figure 14 correlates with the peaks of TCP session throughput (Figure 11) and are caused by high CPU utilization. Therefore, a more conservative ACK rate limiting reduces even more the TCP session RTT and maintains the lower average RTT for a longer period.

It is important to note that tests results show that even on the router with unmodified Linux kernel ACK rate (Figure 12) is much lower than could be expected according to RFC 5681 and RFC 1122, which requires that "ACK should be generated for every second full-size segment", and is close to the router with enabled ACK rate limiting. It can be explained that the router without ACK rate limiting encountered the high CPU demand caused by the high TCP packet rate and in order to limit the CPU load the kernel had to reduce ACK rate and consequently TCP session throughput. Therefore, the ACK rate limiting under the same circumstances (Figure 17) still can halve the ACK rate and consequently free router resources allocated for ACK checking and further processing. It allows the TCP receiver under heavy data traffic to reach much greater TCP throughput and shorter RTT. Furthermore, ACK rate limiting also lessens a load on the TCP packet sender, because of the reduced number of received ACKs and fewer resources needed to handle them.

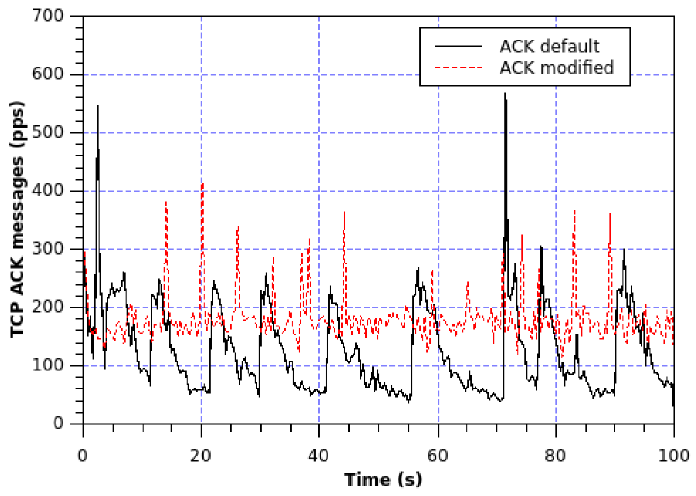

Figure 17. Comparison of ACK transmission between a modified and an unmodified Linux kernel.

The last tests of ACK rate limiting have been performed to examine how the reduction of ACK rate influences the growth of the congestion window (CWND) on TCP packet sending device after a packet drop. The change of CWND size and slow start threshold (SSTRESH) during the TCP session has been analyzed using the Linux `tcpprobe`, the TCP traffic tracking and recording kernel tool based on `kprobe`. The CWND size is directly proportional to the ACK rate, therefore, after the reduction of ACK rate, a reduction of the CWND growth rate could be expected. The tests result on CWND size and SSTHRSH are shown in Figures 18 and 19. In both cases, with and without ACK rate limiting, the TCP session CWND growth and recovery behave identically and no significant difference in TCP Cubic congestion control operation has been observed.

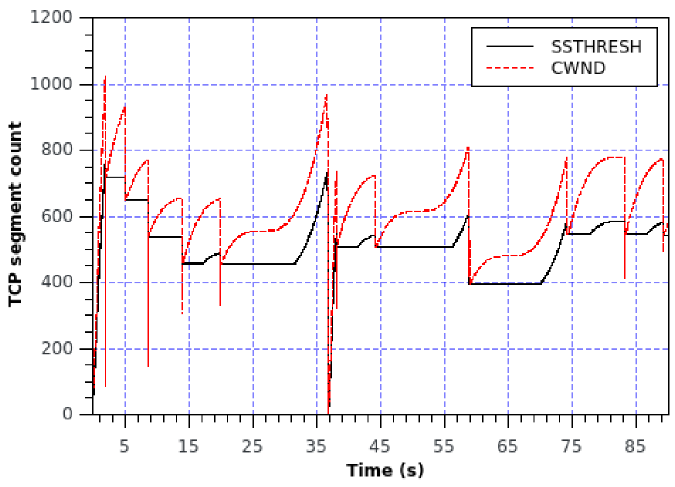

Figure 18. Slow start threshold and congestion window on TCP sender with an unmodified Linux kernel.

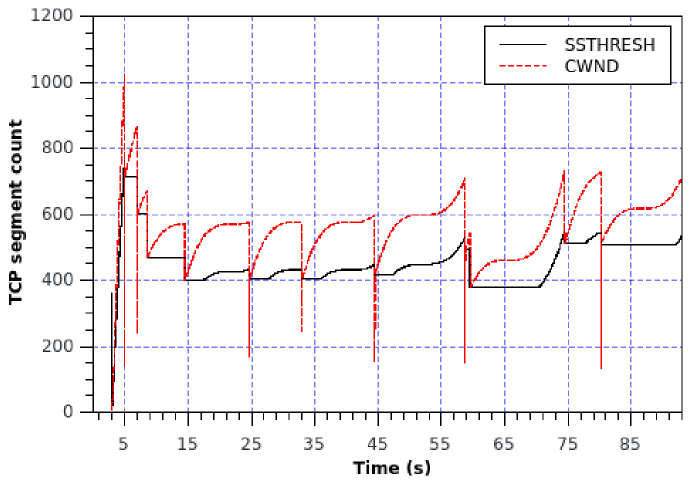

Figure 19. Slow start threshold and congestion window on TCP sender with a modified Linux kernel.

Most importantly, in the Linux kernel with ACK rate limiting, TCP Cubic operates as in the congestion avoidance phase that is critical to the TCP session recovery process. The ACK rate limiting influences TCP session CWND growth (see Figure 19) and makes TCP sessions more competitive to obtaining more network resources over other TCP sessions. The network device with a modified Linux kernel has a slightly lower SSTRESH and smaller CWND, therefore, TCP Cubic avoids CWND the exponential growth phase. Such TCP Cubic behavior is influenced by shorter RTT that leads to a high TCP data rate. However, packet drop due to the tRTO timer expiration could happen at lower SSTRESH and smaller CWND size. Nevertheless, during the test period, the network device with the modified Linux kernel has more constant and smaller TCP sessions CWND size (Figure 19). This makes the TCP packet transmission rate more uniform and the utilization of device NIC buffer is more stable.

6. Discussion

The results presented in previous sections show that the presented ACK rate limiting implementation does not harm TCP performance and QoS parameters. The TCP session throughput increases and exhibits more uniform data rates, RTT remains the same or even smaller comparing to the system with the unmodified kernel. The TCP session CWND growth and recovery behave identically to the system without ACK limiting and no significant difference in TCP Cubic congestion control operation has been observed.

The proposed modification of the TCP stack should not introduce any effect on TCP flow control, connection management, congestion control, etc. It reduces the unneeded packet processing by limiting the ACK rate. The ACK limiting will not start until the TCP initialization state is passed, the client-side TCP congestion window size is maximum, the TCP session has been stable. If a network link is congested, the kernel will not initialize ACK rate limiting and do not allow any reduction of ACK. The ACK rate limiting is disabled if a packet lost or out of order packet is received, and the TCP session recovers, as usual, using fast retransmit mode.

Modifications should not compromise security. TCP SYN flooding attacks are used in TCP init state, while ACK limiting is not yet activated. TCP ACK "overclocking" attack is based on generating several ACK for every received TCP data segment. Proposed modifications suggest the opposite solution, ACK limiting.

The lower network utilization spares additional CPU resources for TCP packet processing in embedded or other network devices with limited computing power. The relation between receiving the system with ACK limiting performance and CPU load is shown in Figure 20. The increase of the performance should be understood as relative CPU load reduction caused by employing ACK rate limiting, e.g., in case of 80% ACK drop the performance increased by 30% in comparison with the unmodified system under 25% CPU load. Meanwhile, if the CPU load of an unmodified system is more than 60% (more than routers overload threshold) in case of 80% ACK drop the increase of system performance is approx 32%.

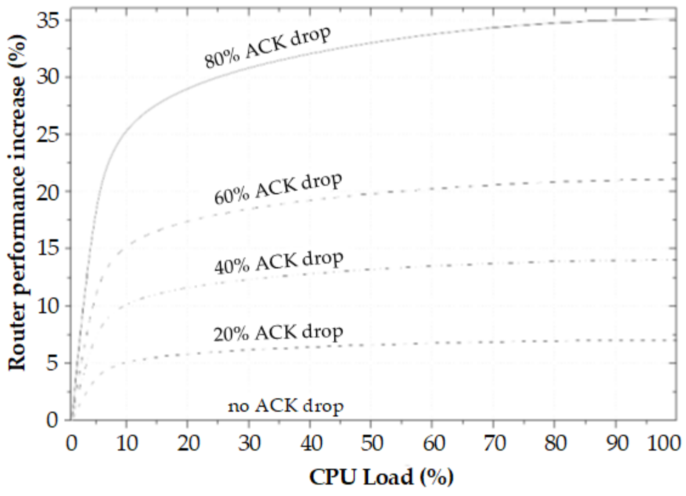

Figure 20. Relation between performance and CPU load in receiving system with ACK limiting.

The reduction of needed CPU resources for TCP data processing will benefit not only TCP data receiver with ACK limiting enabled, but also the TCP data sender, even not using ACK limiting. The TCP sender will have fewer TCP packets to process because the TCP data receiver will not send back ACK for every second TCP data segment.

The significant impact of ACK rate limiting would be e.g., customer-premises equipment (CPE) such as routers and modems that are used in small offices or at home, NAS, or other IoT devices that are used for data storage and usually have limited CPU resources.

Reduced TCP data rates free a significant amount of CPU resources and increase TCP data throughput.

The reduction of ACK messages could also lead to better performance of the network in general, i.e., routers, firewalls, proxies, load balancers, which are directly affected by high TCP data rates and could have a significant gain in CPU resources.

7. Conclusions

The proposed ACK rate limiting algorithm can be used in embedded and low-power devices without any impact on TCP session stability or degradation of its performance. However, the upper limit of ACK rate reduction should be carefully determined based on the CWND and the network conditions.

The algorithm tried to achieve the best TCP session throughput and did not overload network device CPU under current link conditions and communicating peers' configurations by measuring the receiving TCP data packet rate and delay. The results of performed tests suggested that the ACK rate limiting could increase TCP throughput and reduce the CPU load of TCP data processing up to 50% in tested configurations.

The ACK rate limiting did not affect the performance of network devices or TCP data throughput on not congested network links. However, in some cases, the fluctuation of RTT leaded to the expiration of the t_{RTO} timer of the TCP session and caused TCP packet drops. More often such a big fluctuation of RTT occured on network devices experiencing the heavy CPU utilization and that resulted in TCP packet retransmission. Even with this disadvantage, the performed test results show, on average, much better TCP session throughput with ACK rate limiting enabled.

The degradation of TCP data throughput can be observed in a case when the TCP receiver is the bottleneck. The CPU load of the TCP receiver limited the TCP packet receiving rate and thus TCP session throughput was limited.

The proposed modification of the TCP stack should not introduce any effect on TCP flow control, connection management, congestion control, etc., or impact the security. However, a more thorough examination of possible side effects of the proposed ACK limiting on TCP flow control, connection management, congestion control, and connection security in different QoS scenarios would be carried out.

Author Contributions: Conceptualization, A.S. and Š.P.; methodology, A.S.; software, A.S.; validation, A.S. and Š.P.; formal analysis, A.S., Š. P. and A.K.; investigation, A.S., Š.P. and A.K.; resources, A.S.; writing—original draft preparation, A.S. and Š.P.; writing—review and editing, A.K.; visualization, A.S. and Š.P.; supervision, A.S. and Š.P.; funding acquisition, Š.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khomh, F.; Yuan, H.; Zou, Y. Adapting Linux for Mobile Platforms: An Empirical Study of Android. In Proceedings of the 28th International Conference on Software Maintenance (ICSM), Riva del Garda, Italy, 23–30 September 2012; pp. 629–632.
2. Hanford, N.; Ahuja, V.; Farrens, M.K.; Tierney, B.; Ghosal, D.A. Survey of End-System Optimizations for High-Speed Networks. *ACM Comput. Surv.* **2018**, *51*, 1–36. [CrossRef]
3. He, K.; Rozner, E.; Agarwal, K.; Gu, I.J.; Felter, W.; Carter, J.; Akella, A. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 244–257.
4. Nguyen, T.A.N.; Gangadhar, S.; Sterbenz, J.P.G. Performance Evaluation of TCP Congestion Control Algorithms in Data Center Networks. In Proceedings of the 11th International Conference on Future Internet Technologies, Nanjing, China, 15–17 June 2016; pp. 21–28.
5. Jacobson, V. Congestion avoidance and control. In Proceedings of the IEEE Conference IEEE SIGCOMM, Stanford, CA, USA, 16–18 August 1988; pp. 314–329.

6. Ohta, Y.; Nakamura, M.; Kawasaki, Y.; Ode, T. Controlling TCP ACK transmission for throughput improvement in LTE-Advanced Pro. In Proceedings of the IEEE Conference on Standards for Communications and Networking (CSCN), Berlin, Germany, 31 October–2 November 2016.
7. Liu, Q.; Xu, K.; Wang, H.; Shen, M.; Li, L.; Xiao, Q. Measurement, Modeling, and Analysis of TCP in High-Speed Mobility Scenarios. In Proceedings of the IEEE 36th International Conference on Distributed Computing Systems, Nara, Japan, 27–30 June 2016.
8. Kajackas, A.; Pavilanskas, L. Analysis of the Technological Expenditures of Common WLAN Models. *Electron. Electr. Eng.* **2006**, *72*, 19–24.
9. De Silva, G.; Chan, M.C.; Ooi, W. T. Throughput Estimation for Short Lived TCP Cubic Flows. In Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Hiroshima, Japan, 28 November–1 December 2016.
10. Chatzimisios P.; Vitsas, V.; Boucouvalas, A.C. Throughput and delay analysis of IEEE 802.11 protocol. In Proceedings of the 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, Liverpool, UK, 30–31 October 2002.
11. Tierney, B.; Kissel, E.; Swany, M.; Pouyoul, E. Efficient data transfer protocols for big data. In Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science), Chicago, IL, USA, 8–12 October 2012; pp. 1–9.
12. Paredes-Farrera, M.; Fleury, M.; Ghanbari, M. Router Response to Traffic at a Bottleneck Link. In Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, Tridentcom, Barcelona, Spain, 1–3 March 2006.
13. Cui, L.; Cui, X.; Lee, W. A Segment-based SACK Scheme for TCP Over the Error- prone Links. *Wirel. Pers. Commun.* **2011**, *61*, 383–402. [CrossRef]
14. Hu, H.; Kachan, D.; Siemens, E. Improving TCP Performance on CSMA/CA Connections. In Proceedings of the International Conference on Applied Innovation in IT, Köthen, Germany, 26–27 March 2014.
15. Balakrishnan, H.; Padmanabhan, V.N.; Katz, R.H. The effects of asymmetry on TCP performance. In Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking MobiCom '97, Budapest, Hungary, 26–30 September 1997.
16. Nie, X.; Zhao, Y.; Li, Z.; Chen, G.; Sui, K.; Zhang, J.; Ye, Z.; Pei, D. Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning. *IEEE J. Sel. Areas Commun.* **2019**, *37*, 1231–1247. [CrossRef]
17. Cardwell, N.; Yuchung, C.; Stephen, C.G.; Hassas, S.; Jacobson, V. BBR: Congestion-Based Congestion Control. *Commun. ACM* **2016**, *60*, 0001–0782. [CrossRef]
18. Chansook, L. Improving Congestion Control of TCP for Constrained IoT Networks. *Sensors* **2020**, *20*, 4774.
19. Gonzalo, O.; Lara-Cueva, R.; Martínez, D.; de Almeida, C. Performance Analysis of a Novel TCP Protocol Algorithm Adapted to Wireless Networks. *Future Internet* **2020**, *12*, 101.
20. Weifeng, S.; Shumiao, Y.; Yuanxun, X.; Zhenquan, Q. Parallel Transmission of Distributed Sensor Based on SCTP and TCP for Heterogeneous Wireless Networks in IoT. *Sensors* **2019**, *19*, 2005.
21. Seth S.; Venkatesulu M.A. *TCP/IP Architecture, Design and Implementation in Linux*; Wiley-IEEE Computer Society Pr: Los Alamitos, CA, USA, 2008.
22. TCP Implementation in Linux: A Brief Tutorial. Available online: http://www.ece.virginia.edu/mv/research/DOE09/publications/TCPlinux.pdf (accessed on 18 October 2019).
23. Implementing the Internet Checksum in Hardware Internet Access. Available online: https://tools.ietf.org/html/rfc1936 (accessed on 18 October 2019).
24. Stevens, W.R.; Rago, S.A. *Advanced Programming in the Unix Environment*, 3rd ed.; Addison-Wesley Professional: Boston, MA, USA, 2013.
25. Statkus, A. Performance Investigation of Data Packets Transport Control Protocol in Heterogeneous Networks. Ph.D. Thesis, Vilnius Gediminas Technical University, Vilnius, Lithuania, 2017.
26. Paulikas, S; Statkus, A. Improving of TCP Performance of Embedded Network Devices. *Eng. Technol. Open Access* **2020**, *3*, 555618.
27. Dutt, S.; Habibi, D.; Ahmad, I. A low-cost Atheros system-on-chip and Open-WRT based testbed for 802.11 WLAN research. In Proceedings of the IEEE Region 10 Annual International Conference TENCON, Cebu, Philippines, 26–30 November 2012.
28. Tafa, I.; Beqiri, E.; Paci, H.; Kajo, E.; Xhuvani, A. 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. In Proceedings of the 3rd IEEE International Conference on Intelligent Networking and Collaborative Systems INCoS, Fukuoka, Japan, 30 November–2 December 2011.
29. Rathore, M.; Hidell, M.; Sjodin, P. KVM vs. LXC: Comparing performance and isolation of hardware-assisted virtual routers. *Am. J. Netw. Commun.* **2013**, *2*, 88–96. [CrossRef]
30. Rizzo, L.; Lettieri, G.; Maffione, V. Speeding up packet I/O in virtual machines. In Proceedings of the Architectures for Networking and Communications Systems, San Jose, CA, USA, 21–22 October 2013.