Engineering Technology
Open Access Journal

Juniper PUBLISHERS key to the Researchers

Research Article
Volume 3 Issue 4 - October 2020
DOI: 10.19080/ETOAJ.2020.03.555618

Eng Technol Open Acc
Copyright © All rights are reserved by Sarunas Paulikas

Improving of TCP Performance of Embedded Network Devices

Sarunas Paulikas* and Arunas Statkus

Department of Computer Science and Communications Technologies, Vilnius Gediminas Technical University, Lithuania

Submission: October 09, 2020; **Published:** October 22, 2020

***Corresponding author:** Sarunas Paulikas, Department of Computer Science and Communications Technologies, Vilnius Gediminas Technical University, Naugarduko 41, Vilnius, Lithuania

Abstract

This article deals with TCP optimization and performance improvement issues for low-power embedded devices. One of the ways to increase TCP performance in the high-speed networks or low-power devices is to reduce the number of TCP ACK messages generated or received by the system. It has been shown that the TCP congestion window and acknowledgment mechanism are dependent processes with common variables. Therefore, the growth of network bandwidth is coherent with the ACK rate. When the network capacity is increasing, the ACK rate on the channel is increasing as well. This leads to network equipment CPU performance degradation. To resolve this flaw, the ACK rate limiting has been proposed and implemented in the Linux kernel stack.

Keywords: Transport control protocol; Acknowledgement limiting; ACK rate limiting; Linux kernel TCP stack

Introduction

Over the past decade, the use of the Internet has been extending in different areas, e.g. from web browsing in PCs to embedded and mobile devices. A major shift towards Linux-powered devices is seen in the Internet of Things (IoT) and low-power computing segment. This not only liberates from the long and expensive development process of new systems but also allows shortening the systems development life cycle of new products [1]. However, the optimization or performance improvement issues are substantial for all low-power embedded Linux-based devices. The lack of CPU performance is the main bottleneck in said devices for achieving better performance and response time. The main protocol for interconnecting such network devices is the Transport Control Protocol (TCP). TCP runs over more network technologies than any other protocol and provides the highest degree of interoperability. It can have a huge impact on the whole system performance if the embedded CPU is consumed by network stack processing.

TCP optimization is also important to high-end servers or clusters. This optimization or improvement of TCP can enhance data center performance in the service delivery layer, as well as decreasing power consumption due to reduced need for the computing power of CPU [2].

The TCP is a reliable connection-oriented protocol which implements the flow control on the heterogeneous network simultaneously utilizing a sliding window, as well as using congestion avoidance and acknowledgment (ACK) mechanisms. The main problem with TCP is that typical protocol implementation as defined in the Internet Engineering Task Force (IETF) Internet Standards (RFCs), specifically (RFC 3782), is successful at low-speed data links (up to 100Mbps) but unfit for high-speed networks due to slow grow-up of the congestion window (RFC 5681), and poor evaluation of upper bound of the channel [3]. This is because the congestion window growth function, W(t), is squared in the initial phase (RFC 2581) and depends on channel delay: if latency is high, the congestion window growth rate is low. To eliminate the before-mentioned drawback, many TCP congestion control algorithms have been developed. Most common algorithms oriented towards large volumes of data transfer, are the following: HSTCP (RFC 3649), Fast (RFC 3782), STCP (RFC 3286), and Cubic [4]. All these TCP versions rely on end-to-end ACK and congestion window [5], but the general difference is that W(t) is cube-root on the initial phase and depends on latency marginally, compared to typical versions. The congestion window growth is defined in real-time and depends on the latest congestion event, i.e. received ACK [4]. In other words, the sender is allowed to increase the TCP data rate for each incoming ACK. As a result, the unhampered transmission of ACK is very important for successful flow control, and any ACK rate suspension can influence the TCP functionality and performance [6-8]. In consequence, the growth of network

bandwidth is coherent with the growth of the ACK rate. When the network capacity is increasing, the ACK rate on the channel is increasing as well. Consequently, the high-speed network possesses another undesirable feature, namely, the growth of technological expenditures [9].

TCP can also encounter a poor performance of short frame traffic that is analyzed in detail in [10,11]. This degradation occurs for two reasons: technological expenditures [9] and CPU overload [12]. The latter concern is analyzed in [13], and it occurs due to the fact that the router CPU load depends on the packet rate but not on the packet length. This situation is more obvious on high-speed networks [12], since with the growth of bandwidth the TCP ACK rate is also increasing. For the traffic of a large number of short packets more efficient routers will be needed.

One of the options to increase the TCP performance on high-speed networks or low-power devices is to reduce the number of TCP ACK generated or received by the system. This can be done through the ACK rate limiting mechanism [14], which is often used in asymmetric and wireless network environment [6-8] or implemented in Linux kernel TCP packet limiting function.

The aim of the current paper is to provide more insight into Linux kernel, as well as analyzing the TCP ACK rate limiting algorithms needed to implement ACK rate limiting in Linux kernel.

TCP Acknowledgement Generation

Linux kernel is an event-driven system that starts processing data after an event or interrupt has been received [15]. If the system receives a packet on the network interface card (NIC), an interrupt is generated to signal the CPU (for simplicity, a system with one CPU is considered). Every system interrupt comes with a unique interrupt number according to which the OS selects an appropriate driver to handle the interrupt. The procedure responsible for selecting the driver is called an interrupt handler. On the Ethernet network, for example, if a packet is received the system calls the NIC driver to handle the appropriate interrupt [16]. The NIC driver first puts the received packet into NIC memory, while in Ethernet case, it performs the CRC and integrity validation checking [17]. If the packet received is not corrupt, it is moved to the system memory which was allocated by the NIC for received data. In a case, when the receiving packet rate is too high for the system to handle the packet processing, or the allocated system memory is not sufficient to store the received data packet, the NIC must drop received data packet to protect the system from heavy data packet floods [18].

After the data has been placed into the system memory, the kernel handler starts to process received data; it selects an appropriate process, i.e. calls schedule() that indicates to the Linux kernel scheduler that it can schedule some other process on the CPU. The schedule() calls the software interrupt handler. This is done in order to queue network hardware interrupts and allow the kernel to consecutively process the received data, not permitting hardware interrupts to overrun the CPU.

The received data packet, based on its type, is sent to the upper layer for further processing. To process the received data packet, the kernel must know what protocol type is inside the received Ethernet data frame. This is done by checking the Ether type field that indicates the upper layer protocol. If it is an IP packet, the kernel calls ip_rc() for data processing. The ip_rc() examines the structure and checks the header checksum for validation. If the packet is valid, it is passed through netfilter for firewall processing. If needed, modifications for the packet are made depending on firewall settings (SNAT, DNAT, etc.). If the packet destination is a local system, ip_local_deliver() is called which makes the final packet assembly and passes the packet to ip_local_deliver_finish() which removes the IP header and finds the upper layer protocol by checking the IP header protocol field [16,19].

If the most recently received protocol is TCP and it is validated, it goes for further processing depending on the state of TCP (SYN, RST or FIN). Then, the kernel tries to find the state and socket it belongs to by calling tcp_v4_lookup() and inet_lookup_skb(), and by checking hash table. However, if the TCP session is in the established state and no conditional state occurs (timeout, out of order packet), the connection goes to "Fast Path" for faster data processing. The function responsible for processing of received data packets is tcp_rcv_established(). It goes through the TCP header by checking the sequence number and putting the received data into the socket buffer for application processing and sends an ACK packet to the data sender [19].

After the TCP data has been processed by tcp_rcv_established() and validated, the kernel calls tcp_ack_snd_check() (see Table 1) for checking if there is a need to send an ACK [15].

The tcp_ack_snd_check()receives two arguments: sk, structure of the packet, and variable, ofo_possible, that can take values of 1 and 0. If ofo_possible is 1, the out of order segments have been received (see Table 1). The tcp_ack_snd_check() decides if the TCP ACK must be sent now, i.e. execute tcp_send_ack(), or it can be delayed.

Afterwards, the kernel can send an ACK immediately or the ACK must be delayed, by calling tcp_send_delayed_ack() which adjusts kernel internal TCP timers based on RTT value and system minimum and maximum delay values. In the case, when there is no TCP data to send back over the same TCP session and TCP PUSH [19] bit is not set, the TCP ACK generation must be done basing on the first two conditions: if more than one full-size segment is received, and if the kernel has enough space in the receive buffer [18].

During high data rate, the TCP session with excess ACK rate can directly influence not only network equipment but also connection links and network endpoints. Due to the high TCP ACK

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.

rate, the TCP data sender or receiver can become a bottleneck and will not be able to handle high TCP data and ACK rate. By reducing the TCP ACK rate the system can significantly reduce network load and increase TCP performance in embedded or low-power devices. Therefore, in this article a modification to the Linux kernel TCP ACK sending function has been proposed to reduce the TCP ACK rate and increase TCP session performance.

The TCP ACK Limiting

In Linux kernel, TCP ACK generation in non-interactive data transfer state mainly depends on the two conditions:

a) The value of variable icsk_ack.rcv_mss and

b) The availability of free space in the receives buffer. This applies only if the system does not encounter any specific network issues, like packet loss or reordering (see code in Table 1).

Table 1: Linux kernel input packet processing in C code.

```
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 }
```

As icsk_ack.rcv_mss plays an important role in system load control on heavy loaded systems with the high TCP data rate, it is easier to modify this variable value to control the ACK rate. In order to control ACK rate through icsk_ack.rcv_mss, an additional C variable tp >tcp_ack_rate must be defined. It will increase icsk_ack.rcv_mss by factor and would not impact on any other system code. It will let the system easily reduce or increase the ACK generation rate without impacting on the other kernel code. The only negative impact of tp–>tcp_ack_rate is that the kernel has to store more TCP state variables and use system resources to calculate or change tp–>tcp_ack_rate during the TCP session.

The most important issue of ACK rate limiting and TCP congestion control, in general, is that the operating system of TCP server or client is not aware of network conditions (link throughput, delay, transport technologies, etc.) or how they are changing in course of the time. It means that the TCP must adapt itself and change connection conditions after the occurrence of packet disorder or packet loss on the network or remote system. Also, the TCP data receiver does not know in which TCP state (slow start, congestion avoidance, fast recovery, etc.) the remote system is. If ACK rate limiting is activated too early, the system will have a negative impact on TCP. If it is activated too late, no positive impact on TCP data throughput or system performance will be achieved.

The TCP client sending data rate depends on the congestion window (CWND) size which is controlled by the slow start congestion window algorithm and delay [5]. In order to obtain the best TCP performance using ACK rate limiting, at first the operating system must allow CWND of the TCP session to reach its maximum allowed size or advertised receiver window size (RWND), i.e. the maximum amount of data that the receiver can receive, and the TCP session must be in a congestion avoidance phase. Given these conditions, the ACK rate limiting algorithm can be activated, which gradually, by small fractions over time, reduces the ACK rate. An abrupt reduction of the ACK rate could lead to a suddenly increased network delay product (NDP) or round-trip time (RTT) that could become bigger than retransmission timeout (RTO), and the TCP session enters a slow start phase.

The problem with the TCP data receiver is that it does not have any information or indications about the client-side TCP status, e.g. the CWND size, i.e. whether it is of the maximum allowed size. In addition, the TCP data receiver does not know when the slow start phase stops and the congestion avoidance starts on the TCP data sender. The only information the system can rely on is that on the two conditions defined in RFC 5681:

a) In the congestion avoidance phase, the increase of sending window size must not be bigger than one full-size segment (SMSS) per RTT;

b) In the slow start phase, the sending window is increased by at most one SMSS for each ACK received.

As the CWND size growth in the slow start phase is exponential and in the congestion avoidance phase linear [20], the CWND size

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.

most of the time grows by same constant size. This presupposition allows calculating CWND size in the worst-case scenario when the TCP data sender starts from the congestion avoidance phase. To start TCP ACK rate limiting, it is essential to find the time or packet count needed for the TCP sender to set the CWND to the maximum allowed size. The CWND size growth function is defined as [20]:

$$W_{CWND} = W_{CWND0} + \frac{SMSS^2}{W_{CWND}} \qquad (1)$$

where W_{CWND} is a new TCP congestion window, W_{CWND0} is the previous TCP congestion windows, and SMSS is the sender's maximum segment size in bytes.

The second problem with ACK rate limiting is that the client receiving fewer acknowledgments will reduce the sending rate of TCP data if the CWND size is near or less than the bandwidth-delay product (BDP) or the data transmission channel has a high packet capacity (RFC 1072). In this case, the TCP sender can send more data to the TCP receiver and after receiving an ACK can remove sent data packets from the CWND buffer, freeing it for future packets.

The ACK rate from TCP receiver is reduced in _tcp_ack_snd_check() which checks received TCP segment count and __tcp_slect_widow(sk) >= tp–>rcv_wnd values. If the system is overloaded or cannot keep pace with data receiving rate and cannot clean up the receive buffer fast enough, the __tcp_slect_widow(sk) >= tp–>rcv_wnd will slow down the ACK sending rate and reduce the TCP throughput until the buffer is freed [15]. In such a case, the TCP data sender should reduce the TCP data sending rate by reducing the ACK rate. To overcome this problem and prevent TCP performance from degrading when ACK rate limiting is activated, the following conditions must be met:

a) The RWND variable tp–>rcv_wnd and the maximum system allowed CWND size must be large enough to store more than BDP of data in CWND.

b) TCP windows, CWND and RWND, must be monitored periodically, since the RWND size is advertised to the client, and the maximum allowed CWND size is set from the RWND value. Also, the CWND depends on system settings and can change over time as well as being less than advertised RWND size to introduce TCP throughput degradation.

c) RTT and traffic jittery must be measured regularly. TCP traffic can face jitter problems triggered by ACK rate limiting or when the client starts sending data in bursts. This can occur when the CWND size is smaller than the BDP value.

In case the RWND and CWND sizes are big enough and RTT time is low, the ACK sending rate can be reduced significantly without losing performance or observing receive traffic jittering. However, if the ACK rate limiting value tcp_ack_rate_val does not stop growing and continues to increase, the same performance degradation and jittering problems will be encountered at some point and the re-increase of RWND and CWND sizes will be needed to achieve the same TCP throughput.

The main indicator of an excessive ACK rate limiting value is the reduction of throughput. If TCP data throughput is degrading after the increase in ACK rate limit, it must be considered that the ACK limiting algorithm has reached the maximum reduction of ACK rate. The needed RWND value for the sender to have, can be calculated based on the network BDP.

The second condition indicating the maximum ACK rate limiting value is the increase in jitter in RTT measurements. As the jitter is difficult to measure and it also depends on the network as well as on sender-side conditions, it should not be used as an indicator for the maximum of ACK rate limiting. However, the jitter can be used in conjunction with throughput for more accurate detection of the upper limit of ACK rate limiting. In this paper, only the change of TCP data throughput is considered as an indicator for an excessive ACK rate limiting value.

If the packet rate/count does not change during the current period, compared to the value of the previous period, the decision can be made that the ACK rate reduction did not impact on sending TCP data. If it does, the system must fall back to the previous ACK rate limiting value and put this value as a new maximum for ACK rate limiting before the new RWND size is advertised or the network condition changes.

Linux Kernel TCP Stack Modification for ACK Limiting

Basing on the concept of ACK rate limiting explained in the previous sections, ACK rate can be changed easily by increasing icsk_ack.rcv_mss. The default code of Linux kernel (tcp_input.c source code of __tcp_ack_snd_check() function) is modified by adding additional variable tp->tcp_ack_rate, which multiplies icsk_ack.rcv_mss and reduces the ACK rate.

The tcp_ack_rate_val variable defines the necessary number of TCP data segments which must be received before allowing them to send an ACK. In the default or initial state of TCP session, the value is equal to 1, and an ACK is generated after every second full-size TCP data segment (MMS). If no TCP data packet loss occurs during a period, or after receiving WIN/SMSS TCP data packet, tcp_ack_rate_val can be increased.

As was explained before, tcp_ack_rate_val is activated after the client-side CWND is at the maximum size. However, ACK rate limiting can impact on other TCP functions such as fast retransmit or recovery after packet loss. To avoid it, the kernel must also track if any packet loss or retransmission has occurred.

As was explained, ack_rate_val must be activated after the client-side CWND has reached the maximum size and the TCP session has been stable. If ACK rate limiting is activated after packet loss or after RTO, it can impact on other TCP functions, i.e. fast retransmit or recovery after packet loss, and lead to TCP throughput performance degradation or increase in RTO. To avoid it, the kernel must track the TCP session status and packet loss or retransmissions and let the TCP session recover. The Linux kernel ACK rate limiting implementation has been shown in the flow graph (Figure 1).

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.

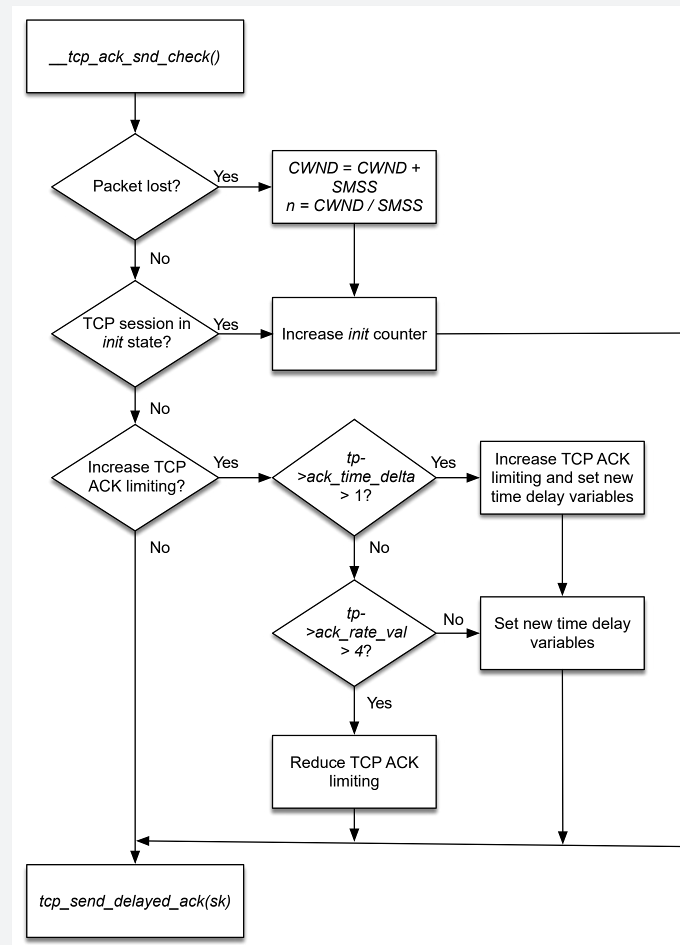

Figure 1: Acknowledgment limiting algorithm.

Before sending the TCP ACK, the Linux kernel checks the TCP state. If a packet loss occurs or out of order TCP data segments are received, the ACK rate limiting controlling variable values are reset, and ACK limiting is disabled (Figure 1, "Packet lost?"). Afterwards, the TCP session data flow is restored using fast retransmit mode (Figure 1, "TCP in Init state"). Also, reset of start_ack_lmt forces the Linux kernel to reset init_pkt_cnt, which controls the initial stage of TCP session, and to set new maximum allowed ACK message rate ack_rate_max. This variable is used to stop the ACK rate limiting, and its value is set to one third of rcv_wnd value divided by receive Maximum Segment Size (MSS) but not greater than the length of 64 segments. Finally, delta time values are reset.

If the TCP session is not in the initial state, the algorithm checks if the level ACK rate limiting could be increased (Figure 1, "Increase TCP ACK limiting"). It is done by calculating throughput of the current TCP session. If the calculated throughput is greater or the same as in the previous stage, ACK rate limiting is increased and the new throughput value is recorded.

Additional TCP state variables are added to the existing Linux kernel code for tracking TCP sessions. After a packet loss or after receiving out of order TCP data segments ACK rate limiting values must be reset, and the TCP session must be allowed to be recovered after a packet loss. If even one segment is lost, the ACK limiting algorithm must be disabled, and fast retransmit is allowed to restore the TCP session flow to a normal state.

First, the tp–>init_pkt_cnt, which is used in the initial stage of ACK rate limiting, is reset. Further, tp–>init_pkt_cnt is compared to tp–>start_ack_lmt which is received at the start of every new TCP session from tcp_ack_limit_init(). The tp–>init_pkt_cnt controls the start of ACK rate limiting process. At first, TCP session must be allowed to go through the slow start phase before starting ACK rate limiting. During this phase, the function also defines the new maximum allowed ACK rate, tp>ack_rate_max, that is used to stop the ACK rate limiting after reaching its maximum and preventing packet loss. It is set to one-third of the RWND value divided by receive MSS. In addition, checking of the updated tp–>ack_rate_max is made to see if the number of segments does not exceed

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.

64. Finally, the values of delta and time variables are reset. They are used by the function for time calculations and for making the decision to increase tp–>ack_rate_cnt.

The upper limit of ACK rate limiting is defined by tp>ack_rate_max which is calculated at the initialization stage in tcp_ack_limit_init() (Table 2).

Table 2: Changes in Linux kernel tcp.h source code for a new `tcp_ack_limit_init()`

```
01 static inline int cp_ack_limit_init(const struct sock *sk)
02{
03  if ( sysctl_tcp_ack_limit > 1 ) {
04    return sysctl_tcp_ack_limit;
05  } else {
06  return ((rwind/ad_mss + 2)*(rwind/ad_mss + 1))/2 -1;
07}
```

The defined algorithm allows changing the upper and lower limits of ACK rate limiting via tp–>start_ack_lmt and tp–>ack_pkt_cnt. It calculates the worst case in which TCP CWND can grow. The new variables can be passed, modified or disabled in the Linux OS kernel in real time via the Linux pseudo file system proc, without modifying the Linux kernel source.

The proposed modification of TCP stack allows reducing both the unneeded packet processing and ACK rate. Also, suggested calculation of lower and upper limits of ACK rate limiting guarantees safe and stable ACK rate limiting. The described ACK rate limiting mechanism can be easily implemented and used in the Linux kernel; the code changes introduced do not affect the functionality or packet flow of TCP segments, but allow more efficient utilization of system resource.

Conclusion

The proposed modification of TCP stack allows reducing both the unneeded packet processing and ACK rate. Also, suggested calculation of lower and upper limits of ACK rate limiting guarantees safe and stable ACK rate limiting. The described ACK rate limiting mechanism can be easily implemented and used in the Linux kernel; the code changes introduced do not affect the functionality or packet flow of TCP segments, but allow more efficient utilization of system resource.

References

1. Khomh F, Yuan H, Zou Y (2012) Adapting Linux for Mobile Platforms: An Empirical Study of Android. Proceedings of 28th International Conference on Software Maintenance (ICSM), Riva del Garda, Italy, pp. 629-632.
2. Hanford N, Ahuja V, Farrens MK, Tierney B, Ghosal D (2018) A Survey of End-System Optimizations for High-Speed Networks. ACM Computing Surveys 51(3): 1-36.
3. He K, Rozner E, Agarwal K, Gu IJ, Felter W, et al. (2016) AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, pp. 244-257.
4. Nguyen TAN, Gangadhar S, Sterbenz JPG (2016) Performance Evaluation of TCP Congestion Control Algorithms in Data Center Networks. Proceedings of the 11th International Conference on Future Internet Technologies, Nanjing, China, pp. 21-28.
5. Jacobson V (1988) Congestion avoidance and control. Proceedings of the IEEE Conference IEEE SIGCOMM, Stanford, California, USA, pp. 314-329.
6. Ohta Y, Nakamura M, Kawasaki Y, Ode T (2016) Controlling TCP ACK transmission for throughput improvement in LTE-Advanced Pro. Proceedings of the IEEE Conference on Standards for Communications and Networking (CSCN), Berlin, Germany.
7. Liu Q, Xu K, Wang H, Shen M, Li L, et al. (2016) Measurement, Modeling, and Analysis of TCP in High-Speed Mobility Scenarios. Proceedings of the IEEE 36th International Conference on Distributed Computing Systems, Nara, Japan.
8. Bak A, Gajowniczek P, Zagożdżon M (2015) Measurement methodology of TCP performance bottlenecks. Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS), *Łódź*, Poland, pp. 1149-1156.
9. Kajackas A, Pavilanskas L (2006) Analysis of the Technological Expenditures of Common WLAN Models. Electronics and Electrical Engineering 72(8): 19-24.
10. De Silva G, Ch Chan M, Ooi WT (2016) Throughput Estimation for Short Lived TCP Cubic Flows. Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Hiroshima, Japan.
11. Chatzimisios P, Vitsas V, Boucouvalas AC (2002) Throughput and delay analysis of IEEE 802.11 protocol. Proceedings 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, Liverpool, UK.
12. Tierney B, Kissel E, Swany M, Pouyoul E (2012) Efficient data transfer protocols for big data. Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science), Chicago, USA, pp. 1-9.
13. Farrera MP, Fleury M, Ghanbari M (2006) Router Response to Traffic at a Bottleneck Link. Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, TRIDENTCOM.
14. Balakrishnan H, Padmanabhan VN, Katz RH (1997) The effects of asymmetry on TCP performance. Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking MobiCom '97, Budapest, Hungary.
15. Seth S, Venkatesulu MA (2008) TCP/IP architecture, design and implementation in Linux, Wiley-IEEE Computer Society.
16. Bhuiyan H, Mcginley M, Li T, Veeraraghavan M TCP Implementation in Linux: A Brief Tutorial.
17. Implementing the Internet Checksum in Hardware Internet access.
18. Stevens WR, Rago SA (2013) Advanced Programming in the Unix Environment, (3rd edn), Addison-Wesley Professional.

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.

19. Unzner MA (2014) Split TCP/IP Stack Implementation for GNU/Linux Internet access. Diplomarbeit.

20. TCP Congestion Control. IETF RFC 5681.

This work is licensed under Creative Commons Attribution 4.0 License

DOI: 10.19080/ETOAJ.2020.03.555618

Your next submission with Juniper Publishers will reach you the below assets

- Quality Editorial service
- Swift Peer Review
- Reprints availability
- E-prints Service
- Manuscript Podcast for convenient understanding
- Global attainment for your research
- Manuscript accessibility in different formats

  (Pdf, E-pub, Full Text, Audio)
- Unceasing customer service

Track the below URL for one-step submission
https://juniperpublishers.com/online-submission.php

How to cite this article: Sarunas P, Arunas S. Improving of TCP Performance of Embedded Network Devices. Eng Technol Open Acc. 2020; 3(4): 555618. DOI: 10.19080/ETOAJ.2020.03.555618.