Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1149–1156
DOI: 10.15439/2015F284
ACSIS, Vol. 5

# Measurement methodology of TCP performance bottlenecks

Andrzej Bąk and Piotr Gajowniczek
Institute of Telecommunications
Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
email: bak@tele.pw.edu.pl

Michał Zagożdżon
Orange Labs
Orange Polska S.A.
Obrzeżna 7, 02-679 Warsaw, Poland
email: michal.zagozdzon@orange.com

***Abstract*—Transmission Control Protocol (TCP) is still used by vast majority of Internet applications. However, the huge increase in bandwidth availability and consumption during the last decade has stimulated the evolution of TCP and introduction of new versions that are more suited for high speed networks. Many factors can influence the performance of TCP protocol, starting from scarcity of network resources, through client or server misconfiguration, to internal limitations of applications. Proper identification of the TCP performance bottlenecks is therefore an important challenge for network operators. In the paper we proposed the methodology for finding root causes of througput degradation in TCP connections based on passive measurements. This methodology was verified by experiments conducted in a live network with 4G wireless Internet access.**

## I. Introduction

Since the foundation of the Internet the vast majority of network data is transmitted using Transmission Control Protocol (TCP). TCP underlies many 'traditional' Internet applications such as web browsing, email, bulk data transfer etc., but also the relatively new ones, such as HTTP adaptive streaming that is quickly becoming the preferred method for Over-The-Top video delivery. All this makes the TCP performance analysis one of the most important research areas of the Internet networking. Since the beginning of TCP's public use in 1989 a lot of research effort was devoted to improve its performance, and the protocol itself has evolved significantly.

The early TCP version used RTO (Retransmission Time-Out) timer to recover from packet loss which was inefficient even on low speed links. The TCP Reno/NewReno version [1] introduced fast retransmit & recovery mechanism that improved the TCP performance in presence of packet loss. For a long time the TCP Reno was a de-facto standard widely deployed in the Internet. However, the significant increase in network capacity observed during the last decade has stimulated introduction of new TCP congestion control algorithms that are more suited for high speed links, such as Fast TCP [2], BIC [3], STCP [4], CUBIC [5] [6] [7], HTCP [8] [9] [10], HSTCP [11] [12], Compound TCP [13], TCP Westwood [14] etc. The new versions of Linux operating system do even allow switching between different congestion control algorithms without the need to recompile the kernel.

This paper presents the methodology of finding the root causes of throuput degradation in TCP connections on the base of passive measurements obtained from probes capturing traffic on the network links. This methodology combines the detection of TCP source application type (greedy vs non-greedy) [15] with estimation of coefficients related to transmission effectiveness that are based on the RFC 6349 [16]. The proposed approach is supported by results obtained from measurements conducted in the live 4G mobile network of Orange Poland.

## II. Sources of TCP performance bottlenecks

TCP uses congestion and flow control mechanisms to control the transmission rate of the sender process by limiting the amount of data that can be transmitted without waiting for acknowledgement (called the *window size*). Changing transmission rate in response to receiver's limitation in processing incoming data (*flow control*) is based on the current size of the receiver's window ($awnd$). This value is advertised to TCP sender process in segments that are sent as acknowledgements to the received data. Too small values of the receiver's window can however negatively affect the performance of the TCP protocol. Therefore, in newer implementations it can be adapted algorithmically depending on the characteristics of the transmission path (such as throughput and delay).

Congestion control is done by algorithms that aim to 'sense' the bottleneck throughput on the transmission path and adapt the transmission rate to this limit. The sender process keeps the state variable called the congestion window ($cwnd$) that works in a similar way as advertised window, except that its value is set by an algorithm running on the sender side. Usually, the sender starts with a small value of $cwnd$ and tries to increase it each time when an acknowledgement for the previously sent data segment is received. The initial phase of aggressive $cwnd$ increase is called a *slow start* - the $cwnd$ is increased by 1 segment after each acknowledgement which leads to exponential growth in the amount of transmitted data. After encountering data loss the $cwnd$ shrinks; the following increase is usually slower and TCP sender enters the phase called *congestion avoidance*. There are many different congestion control algorithms and their variants - for an excellent review see [17]. However, they all share the same purpose - to maximize the usage of capacity available on the transmission path while also minimizing the probability of data loss.

For the TCP sender process the actual window size is the minimum of the advertised receiver's window ($awnd$) and its own congestion window ($cwnd$). After sending full window of data, TCP must stop transmission and wait for acknowledgement. The acknowledgement related to the earliest outstanding segment that was transmitted will start to arrive after the RTT (Round Trip Time) between the sender and the receiver. Each arriving acknowledgment will trigger transmission of the next segment of data awaiting in the output buffer. Hence, the TCP process can send at most $min(cwnd, awnd)$ of data per round trip time cycle and the instantaneous TCP throughput can be roughly estimated as:

$$TCP_{th} = \frac{min(cwnd, awnd)}{RTT} \tag{1}$$

TCP window that is too small may severely limit the performance of TCP connection. In order to obtain high throughput, TCP must be able to fill the network pipeline with data that will keep the network busy. Therefore, the TCP sender's window size must be greater than the bandwidth delay product:

$$min(cwnd, awnd) \geq C * RTT \tag{2}$$

where $C$ denotes the capacity available for TCP connection on the tranmission path.

There are various ways to set the values of $awnd$ and $cwnd$. As it was noted earlier, especially for $cwnd$ there is a number of different congestion control algorithms that react differently to the potential congestion detected either by the Retransmission Time-Out (a timer on sender's side) or by receiving a duplicate acknowledgement (Dup-ACK) from the receiver. Both events lead to segment retransmission and $cwnd$ window scaling, and appear in result of changes in the network environment, such as increased load, change in traffic mix, change in link parameters etc.

Another factor limiting the TCP performance is related to sizing TCP socket buffers on both sides of the connection. The problem of buffers being too small is especially visible in the networks with high bandwidth delay product (the maximum buffer space for TCP sockets depends on the operating system in case of typical Internet hosts). The receiver's socket buffer size can significantly influence the performance of TCP connection (receiver's buffer can limit the sending rate of the TCP source). Therefore, proper configuration of the sockets' buffers is very important to assure high TCP throughput [18]. Modern operating systems introduce automated algorithms for tuning the TCP buffers [19] [20] [21] [22] [23].

Similar case is related to sizing the buffers of network devices [25]. TCP sender can emit data in bursts (up to the current $cwnd$ window size). If network buffers are too small, the inevitable data loss will prevent the congestion window from growing and TCP connection will not be able to ramp up the transmission rate to available capacity. It is generally advised that network buffers should be at least twice the size of the network bandwidth delay product to assure high TCP throughput.

Another factor influencing the achievable TCP throughput is related to packet reordering [26] [27] that can be introduced for example by parallel packet processing in network devices. Receiving out-of-order segments can result in duplicate acknowledgements being sent and interpreted as data loss. This may in turn lead to unnecessary retransmissions, $cwnd$ reduction and throughput degradation. On the receiver's side frequent segment reordering may lead to extensive buffering and potential reduction in receiver's window size.

Finally, the throughput limitation can lie within the application itself. For example, in adaptive HTTP streaming the client requests chunks of video file from the server with frequency related to the encoding rate, even if the available capacity would allow transmitting data faster. In this context, TCP sources can be divided into greedy (always trying to utilize the most of available transmission capacity) and non-greedy (where rate is limited by internal behavior of the source). This classification is utilized in the methodology discussed in section III-B.

## III. Detection of the root causes of TCP performance degradation

In this section we describe the algorithm for detecting the root causes of the TCP performance bottlenecks using passive TCP measurements.

### A. *Network measurements*

Following the recommendations from RFC 6349, it is advised to perform the MTU (Maximum Transmission Unit) discovery procedure (see [28] for reference) before starting measurements, to avoid unwanted packet fragmentation.

We assume that the TCP traffic is monitored near the sender (at the client or at the server, depending on the direction of the transmission). The monitoring point must be close enough to the sender in order to precisely estimate the RTT parameter which is required by the proposed bottleneck detection algorithm. The monitored network traffic is saved by the probes in *.pcap* format for further processing.

The throughput of the TCP connection $TCP_{th}$ can be estimated directly from data captured by passive probes as a ratio of data sent and acknowledged during a measurement period to the length $t$ of this period:

$$TCP_{th} = \frac{ACK(t)}{t} \tag{3}$$

$ACK(t)$ denotes the highest acknowlegement sequence number observed up to time $t$. It can be obtained directly from the headers of the captured TCP segments.

Due to the nature of congestion and flow control mechanisms, the TCP sender needs some time before it can reach the desired transmission speed. This time may vary from few seconds to even hours depending on the network RTT, bandwidth, TCP congestion and flow control algorithms etc. For example, during congestion avoidance phase the TCP source needs approximately 30 seconds to increase the transmssion rate by 10 Mbps if the network RTT is 200 ms. Therefore,

for proper estimation of the TCP throughput the measurement time should be long enough. The following approach is suggested to assure that. Assuming some interval $\Delta t$ and threshold $c$, seek for time instant $t$ that satisfies the following condition:

$$\left| \frac{TCP_{th}(t+\Delta t) - TCP_{th}(t)}{\Delta t} \right| < c \quad (4)$$

The above formula approximates the derivative of TCP rate estimator. The measurement time should be long enough to assure that the TCP rate estimator does not significantly change over time. In the experiments presented further in this paper we assumed $\Delta t = 1s$ and $c = 100$ KB/$s^2$.

In addition to the typical traffic traces captured at measurement points, the proposed methodology requires running some additional measurements to calculate certain TCP performance indicators (described in section III-D). The first measurement is related to estimation of reference (bottleneck) bandwidth $C_{REF}$. This can be achieved by probing the network bottleneck with UDP traffic. There are many variants of this approach - for examples see [29] or [30]. In our measurements we have used the latter: a train of 50 UDP packets was sent to the receiver and the available capacity was measured simply by dividing the total length of the received UDP packets by the total reception time (under the condition of no packet loss).

To verify if the buffers are properly dimensioned, the *back to back frames* test should be also performed. This test consists of sending the specified number of UDP packets with the maximum possible rate and repeating it while increasing the number of transmitted packets in each trial. The maximum batch size that can be sent without observing packet loss is an indirect measure of buffer size on the transmission path of the stream.

### B. Categorizaton of TCP sources

TCP throughput depends on the amount of data the TCP source emits during a single RTT period, as this value is controlled by the congestion and flow control mechanism. At any time instant the amount of outstanding data (sent but not acknowledged) is limited to $min(cwnd, awnd)$. As was explained in section II the TCP source can send at most $min(cwnd, awnd)$ bytes per RTT period. Therefore, the amount of outstanding data in relation to the RTT is an indicator of instantaneous TCP performance.

If the amount of outstanding data is less than what $cwnd$ and $awnd$ parameters allow, it means that the sender is not fully exploiting the available transmission capacity. The cause may be related to internal sender faults (such as application software or hardware issues, CPU overload etc.), but more often is a result of the consent behavior of TCP source (that may not require more throughput, as it is in case of typical streaming applications where transmission speed is related to the bitrate of the video stream). In the opposite case (if the outstanding data is close to the $cwnd$ or $awnd$), the bottlenecks are introduced either by the network or by the receiver.

Summing up the above discussion, the TCP source may fall into one of the following categories:

- *Internally limited*
  Non-greedy source i.e. a TCP connection that is not fully exploiting the capacity available in the network; the amount of outstanding data is significantly lower than the $cwnd$ and $awnd$ windows would allow.
- *Receiver limited*
  TCP source whose transmission rate is limited by the receiver; the amount of outstanding data is close to the $awnd$ and also lower than the $cwnd$.
- *Network limited*
  TCP source whose transmission rate is limited by the network, i.e. by the available capacity, packet loss rate or network RTT; the amount of outstanding data is close to the $cwnd$.

In order to classify the TCP source into one of the above categories we need the following parameters: outstanding data, RTT, receiver's window size and congestion window size. The first three parameters can be easily obtained from the packet traces captured by the passive probes. However, the congestion window is not directly measurable as it is an internal parameter of the TCP stack at the sender and cannot be directly inferred from the TCP traces. In order to cope with this problem we follow the approach of [15]. The TCP connection state is emulated using the recorded TCP traces to recover the $cwnd$ parameter. We also recover the value of RTO to distinguish between retransmissions induced by the fast retransmit phase and those due to the timer expiration. This is required to precisely track the changes in the $cwnd$ parameter.

### C. Emulation of TCP connection state

The internal state of TCP congestion control mechanism is defined by three main parameters: size of congestion window ($cwnd$), threshold for switching between slow start and congestion avoidance phase ($ssthr$), and the retransmission timer (RTO). These parameters are essential for emulation of the TCP connection state.

After a 3-way handshake procedure, the TCP process enters the established state and the TCP sender starts to transmit data. The sender sets its $cwnd$ parameter to some initial value and begins transmitting in the slow start mode. While in slow start, TCP adds one segment to the $cwnd$ for each acknowledged segment, doubling its $cwnd$ with every RTT period. Therefore, during slow start TCP throughput grows exponentially. The aim of this phase is to quickly probe the network capacity and to estimate the optimum window size without heavily overloading the network.

Slow start phase ends when either the $ssthr$ is reached or the segment loss is detected. TCP detects segment loss by two mechanisms: expiration of RTO timer or reception of duplicate acknowledgements (Dup-ACKs). In the first case the TCP sender retransmits all outstanding data and enters the slow start mode again. In the second case, after receiving 3 consecutive Dup-ACKs the TCP sender enters the recovery phase and employs fast retransmit & recovery mechanism to

recover the lost segment. In contrary to the RTO mechanism, only one segment is retransmitted. The assumption behind this approach is that in this case only one segment is most likely lost and there is no need to follow the go-back-N protocol and retransmit all outstanding data.

The sender sets the $ssthr$ to the half of the $cwnd$ window before the segment loss, sets the $cwnd$ to $ssthr$+3 segments and retransmits the segment pointed by Dup-ACKs. Each time another Dup-ACK arrives, the sender adds one segment to the $cwnd$ (inflating the congestion window). The aim of this is to sustain the TCP throughput (as Dup-ACK indicates that the network is still able to deliver packets).

In the recovery phase, the TCP sender is allowed to transmit new data as indicated by $cwnd$. The recovery phase ends when all outstanding data from the beginning of this phase is acknowledged. When leaving the recovery phase the TCP sets the $cwnd$ back to the $ssthr$ and enters the congestion avoidance mode.

In TCP Reno/NewReno versions, during congestion avoidance phase one segment is added to the $cwnd$ in each RTT period. This means that the $cwnd$ grows linearly over time, increasing TCP throughput more conservatively then in slow start phase. However, it may take long time to recover TCP throughput in the network with high bandwidth delay product. Therefore, new congestion control mechanisms introduce more aggressive approaches for increasing the $cwnd$ during congestion avoidance phase.

In order to track the sender's $cwnd$ we emulate the behavior of the TCP protocol. To obtain high accuracy of the emulation we used the original source code of the Linux kernel version 3.18 [24]. The H-TCP congestion control algorithm code was used as this protocol was employed in our test setup. The code was taken from the following TCP modules:

- `tcp_input.c` - estimation of the RTO algorithm; the following function was used:
  - $tcp_rtt_estimator$
- `htcp.c` - estimation of the H-TCP congestion control algorithm; the following functions were used:
  - $measure_achieved_throughput$
  - $htcp_cong_avoid$
  - $tcp_slow_start$
  - $htcp_alpha_update$
  - $htcp_beta_update$
  - $htcp_recalc_ssthresh$

Each time the data or acknowledgment segment is observed, we run an appropriate piece of the Linux kernel code. When the acknowledgement segment with higher sequence number is observed we calculate the RTT sample (the time difference between reception of acknowledgement segment and observation of data segment for the given sequence number at the monitoring point) and run the $tcp_rtt_estimator()$ function to update the value of the RTO timer.

Next, the $measure_achieved_throughput()$ and $htcp_cong_avoid()$ functions of the H-TCP algorithm are called to update the internal state of the congestion control algorithm. The $htcp_cong_avoid()$ function runs the algorithm for slow start or congestion avoidance phase (depending on whether the $cwnd$ is less or greater then $ssthr$) and updates the $cwnd$ value accordingly.

When 3 duplicate acknowledgements are observed we assume that TCP enters the recovery phase of the fast retransmit & recovery mechanism. We call the $htcp_recalc_ssthresh()$ to update the $ssthr$ value according to the H-TCP algorithm. Each time next DupACK segment is observed, we inflate the $cwnd$ by one segment (as specified by NewReno algorithm).

When all outstanding data at the beginning of the recovery phase is acknowledged, the code for regular acknowledgments is executed (emulating congestion avoidance phase). When out-of-sequence data packet is observed that was not acknowledged and the time since transmission of the original segment is greater than the RTO, we assume retransmission due to the timer expiration. The $cwnd$ is reset to the initial value and the slow start code is executed by the $htcp_cong_avoid()$ function. When the $cwnd$ exceeds $ssthr$, the $htcp_cong_avoid()$ function executes the H-TCP congestion avoidance code again for each observed acknowledgement segment.

To validate the implemented TCP state tracking one can compare the outstanding data calculated from measurements with estimated values of the $cwnd$ (as the amount of outstanding data can approximate the $cwnd$, especially for greedy TCP sources).

For automated detection whether the TCP connection is network, receiver or internally limited, we have implemented the algorithm proposed and described in detail in [15]. Unlike in the original algorithm, we do not however divide the TCP flow into chunks and categorize each chunk individually, but rather do the categorization for the TCP connection as a whole.

An example based on network measurements is shown in Fig. 1 and Fig. 2 for H-TCP-based source. Fig. 1 shows the comparison of outstanding data with the value of $cwnd$ estimated by emulation of the H-TCP congestion control algorithm using Linux kernel source code.

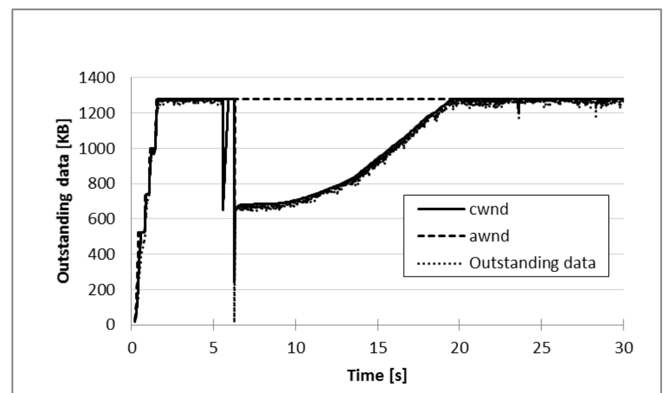


Fig. 1. H-TCP $cwnd$ emulation

The estimated $cwnd$ almost exactly matches the amount of measured outstanding data. Similar results are obtained for the estimation of RTO parameter for the same source (Fig. 2).

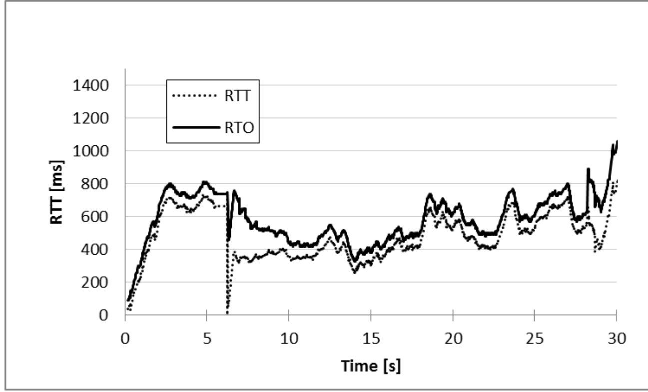


Fig. 2. H-TCP RTO estimation

The following figures present analogous results for the TCP Reno/NewReno based source. The accuracy of $cwnd$ emulation is shown in Fig. 3, RTO estimation in Fig. 4.

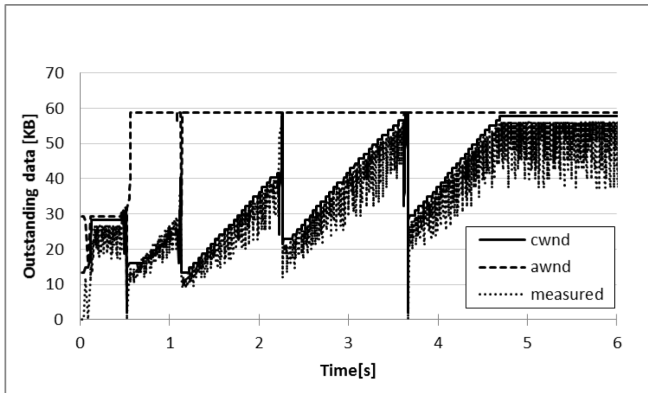


Fig. 3. TCP Reno $cwnd$ emulation

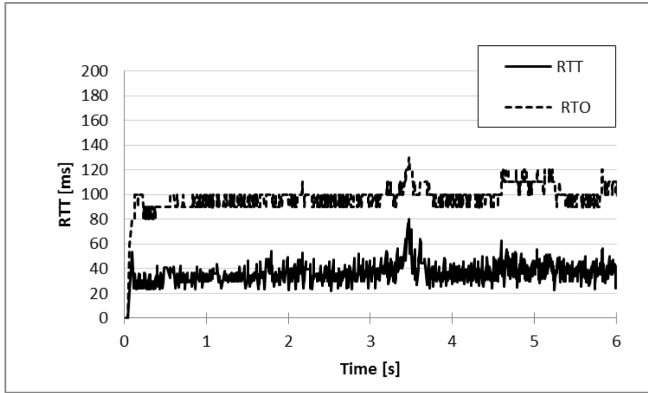


Fig. 4. TCP Reno RTO estimation

### D. TCP performance metrics

Based on RFC 6349, the following metrics are recommended to test the TCP effectiveness.

- *TCP Throughput Ratio* $W$. This metric is calculated as a percentage ratio of achieved throughput $TCP_{th}$ to the reference throughput $C_{REF}$ and should approach 100% for good connections.

$$W = \frac{TCP_{th}}{C_{REF}} * 100 \quad (5)$$

- *TCP transmission effectiveness* $E$. It is a percentage ratio of non-retransmitted data to the total amount of data sent during the measurement period and should also approach 100% for effective connections.

$$E = \frac{D - D_{RET}}{D} * 100 \quad (6)$$

$D_{RET}$ denotes the amount of data retransmitted during the measurement period.

- *Buffer Delay* $T$. To calculate this parameter one needs the reference delay $RTT_{MIN}$ calculated beforehand from measurements taken when the network load is minimal. The *tcptrace* tool can be used for this task. Alternatively, $RTT_{MIN}$ may be approximated by the minimal RTT observed during the actual measurement period. Denoting an average RTT observed within the measurement period as $RTT_{AVG}$, the Buffer Delay can be calculated as:

$$T = \frac{RTT_{AVG} - RTT_{MIN}}{RTT_{MIN}} * 100 \quad (7)$$

As the name implies, this parameter is related to buffer size in the network nodes and can be interpreted as a measure of buffer load imposed by the measured TCP connection (mostly related to the buffer at the bottleneck link). If we assume that buffer size $B$ conforms to the following formula:

$$B > 2 * C_{REF} * RTT_{MIN} \quad (8)$$

then the Buffer Delay should be greater than 200%.

### E. Root cause analysis

For the root cause analysis we use the emulation of the TCP sender state derived from passive measurements and the metrics of TCP connection performance calculated on the base of measurements. The general algorithm is depicted in Fig. 5.

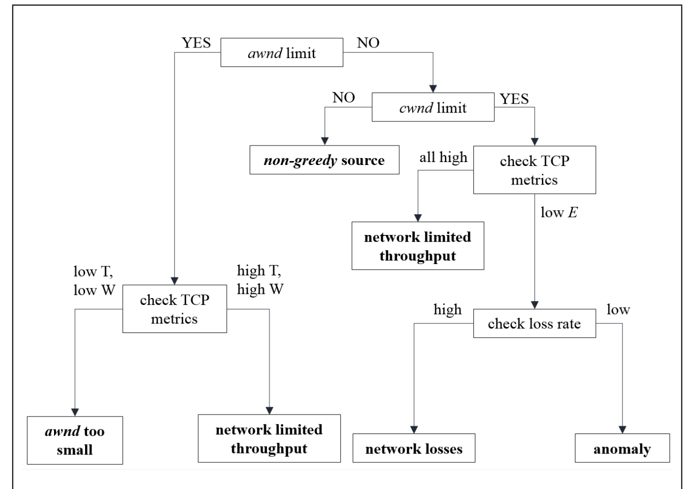


Fig. 5. General algorithm for root cause analysis

In the proposed approach the first task is to check if the throughput is not limited by $awnd$. This can be done by analyzing the behavior of the outstanding data using emulation of the TCP sender state. If it is the case, then it is advised to

check the TCP connection metrics. Low $T$ (low buffering) and low $W$ (low bandwidth utilization) together with large $T$ (lack of retransmissions) support the hypothesis that the advertised $awnd$ value is indeed limiting the sender's performance. If however the $T$ and $W$ are relatively large, the true limitation may lie in the network itself and the $awnd$ value reached by the sender is large enough for high connection efectiveness.

If neither $awnd$ nor $cwnd$ (estimated from emulation of TCP sender state) is the limiting factor then the achieved throughput results from the internal sender constraints (non-greedy source). Low value of Buffer Delay may additionally support this hypothesis.

If the $cwnd$ imposes the limit on achieved throughput, it is advised to check TCP connection metrics. High effectiveness of transmission together with large Buffer Delay confirm that TCP throughput is limited by a bottleneck link in the network. However, if the level of observed retransmissions is high (low $E$), then the reason behind low throughput may lie in excessive packet loss in the network resulting e.g. from faults, bad conditions on wireless access link etc. It has to be noted that TCP retransmissions occur naturally in result of congestion control algorithm continuous attempts to fit the transmission rate to the bottleneck bandwidth, but the excessive level of retransmissions is suspicious and has to be checked further.

Finally, the case when transmission effectiveness is low but the packet loss is also low has to be treated as an anomaly that requires further investigation.

### F. Validation of the proposed approach

The proposed approach was validated by conducting measurements in the real network. The measurement setup is depicted in Fig. 6.

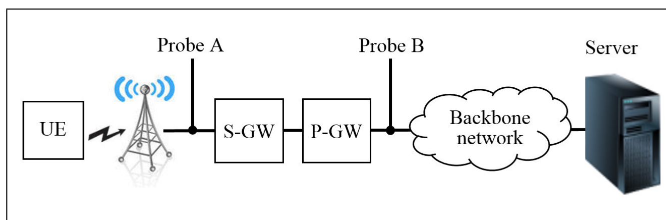


Fig. 6. Measurement setup

Measurements were done in commercial 4G mobile network of Orange Poland with real user traffic served in the background. The setup consisted of the UNIX-based web server connected to the backbone network. The TCP traffic can be monitored at the server and/or at the mobile device (with *tcpdump*). Additionally, two hardware monitoring probes were installed in the mobile access network. The monitored TCP traffic was saved in *.pcap* format for further processing.

We ran a number of tests based on downloading files from the server to the mobile device. Two types of experiments were carried out. In the first case the files were downloaded from the server in a greedy mode. The server was configured to transmit data with maximum possible rate so that the network available capacity was the only limiting factor for TCP throughput.

In the second type of the experiment the socket buffer size at the server was limited below the bandwidth delay product of the network which is approximately 100 KB (40 $ms\ RTT * 20\ Mbps\ C_{REF}$). As can be seen from Fig. 7, the tested TCP connection begins in slow start and within few seconds the $awnd$ and $cwnd$ parameters reach their maximum sizes. This is possible due to large network buffers that can accommodate thousands of packets. After the next few seconds there is a packet drop (signalled by 3 Dup-ACK segments), TCP connection retransmits the lost segment and enters the congestion avoidance phase.

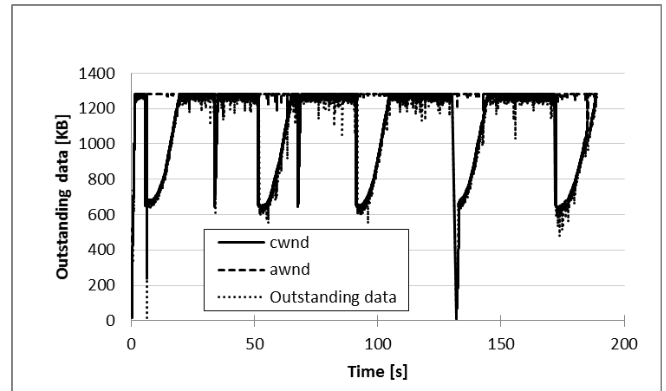


Fig. 7. Network limited TCP connection

While in congestion avoidance the $cwnd$ follows the H-TCP congestion control algorithms. At 130 sec. time instant we observe a retransmission due to the RTO expiration. TCP falls back to the slow start mode and after reaching $ssthr$ (set to 0.5 of the $cwnd$ before segment loss) it switches again to congestion avoidance phase. Notice that the estimated $cwnd$ follows the amount of measured outstanding data very accurately.

**Network limited TCP connection**
min rtt [ms] = 23.5
avg rtt [ms] = 462
avg out data [B] = 1103400
avg awnd [B] = 1275910
avg cwnd [B] = 1114465
measured throughput [Mbps] = 15.8
outstanding data/rtt [Mbps] = 19
fast retransmits = 6
RTO expirations = 5
TCP efficiency (E) [%] = 99.98
buffer delay (T) [%] = 1866

According to the proposed TCP throughput measurement methodology, the tested TCP connection is clearly network limited. The outstanding data follows the $cwnd$ closely while the TCP efficiency $E$ is high which means no excessive packet loss inside the network. The buffer delay $T$ is also very high indicating that TCP connection is transmitting a lot of data to the network (see text in the relevant frame). The network capacity $C_{REF}$, measured with the UDP protocol immediately before starting the test transfers (in the same conditions that influenced maximum throughput achievable in the location

during the experiment), is about 18 Mbps. Therefore, the TCP connection in this case utilizes almost 90% of available capacity (TCP throughput ratio $W$ is also high).

In the second experiment the socket buffer of the receiver was limited to about 60 KB. In this case the outstanding data follows the $awnd$ (see Fig. 8) indicating that the TCP connection is limited by the client.

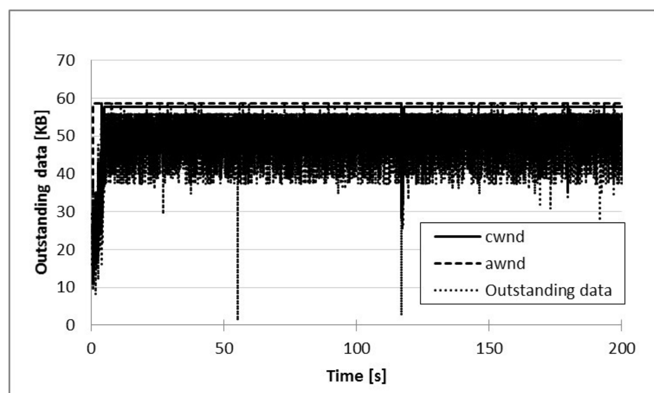


Fig. 8. Receiver limited TCP connection

This reasoning is also justified by low value of buffer delay $T$ which is now below 100%, meaning that TCP is not filling the network with data (see frame). The achieved TCP throughput is about 12.7 Mbps.

**Receiver limited TCP connection**
min rtt [ms] = 17.5
avg rtt [ms] = 32.4
avg out data [B] = 50956
avg awnd [B] = 58616
avg cwnd [B] = 57615
measured throughput [Mbps] = 12.7
outstanding data/rtt [kbps] = 12.5
fast retransmits = 5
RTO expirations = 0
TCP efficiency (E) [%] = 99.99
buffer delay (T) [%] = 85

## IV. Conclusions

The paper presents the methodology for identifying the root cause of the TCP connection performance bottlenecks. We have used the Linux kernel source code to implement the algorithm for estimation of the internal TCP connection state. Such approach allows to infer the dynamics of the TCP congestion window which is otherwise unavailable from passive TCP monitoring. The knowledge of the internal TCP state ($cwnd$, $ssthr$, RTO) is essential in understanding the observed behavior of the TCP connection and allows identifying the source of the TCP throughput limitations. In our approach it is used together with analysis of the TCP performance metrics proposed in RFC 6349. Such combined approach, complimented with additional active measurements (probing available capacity, measuring bottleneck buffers) can be helpful in tracing down network problems related to TCP-based applications.

## References

[1] T. Henderson, S. Floyd, A. Gurtov, Y. Nishida, *RFC 6582: The NewReno Modification to TCP's Fast Recovery Algorithm*

[2] D.X. Wei, Cheng Jin; S.H. Low, S. Hegde, *FAST TCP: Motivation, Architecture, Algorithms, Performance* IEEE/ACM Transactions on Networking, vol.14, no.6, pp.1246-1259, Dec. 2006, doi: 10.1109/TNET.2006.886335

[3] L. Xu, K. Harfoush, I. Rhee, *Binary increase congestion control for fast, long distance networks* Proc. of IEEE INFOCOM, vol. 4, pp. 2514–2524, March 2004

[4] T. Kelly, *Scalable TCP: improving performance in highspeed wide area networks* Computer Communications Review, vol. 32, no. 2, April 2003.

[5] H. Jamal, K. Sultan, *Performance Analysis of TCP Congestion Control Algorithms* Int. Journal of Computers and Comm., Issue 1, vol. 2, 2008

[6] S. Ha, I. Rhee, L. Xu, *CUBIC: a new TCP-friendly high-speed TCP variant* SIGOPS Oper. Syst. Rev. 42, 5 (July 2008), 64-74, doi: 10.1145/1400097.1400105

[7] D.J. Leith, R.N. Shorten, G. McCullagh, *Experimental evaluation of Cubic-TCP* Proc. of PFLDnet, 2008

[8] G. Armitage, L. Stewart, M. Welzl, J. Healy, *An Independent H-TCP Implementation under FreeBSD 7.0 – Description and Observed Behaviour* ACM SIGCOMM Computer Communication Review, vol. 38, no. 3, July 2008

[9] D. Leith, R. Shorten, *H-TCP: TCP for high-speed and long-distance networks* Proc. of PFLDnet, 2004

[10] D.J. Leith, R.N. Shorten, Y. Lee, *H-TCP: A framework for congestion control in high-speed and long-distance networks* Proc. of PFLDnet, 2005

[11] S. Floyd, *RFC 3649: HighSpeed TCP for large congestion windows*

[12] S. Floyd, *RFC 3742: Limited Slow-Start for TCP with Large Congestion Windows*

[13] K. Tan, J. Song, Q. Zhang, M. Sridharan, *A compound TCP approach for high-speed and long distance networks* Proc. of INFOCOM 2006, pp.1-12, 2006, doi: 10.1109/INFOCOM.2006.188

[14] S. Mascolo, C. Casetti, M. Gerla, M.Y. Sanadidi, R. Wang, *TCP Westwood: Bandwidth estimation for enhanced transport over wireless links* Proc. of ACM MOBICOM, 2001, pp. 287–297

[15] M. Schiavone, P. Romirer-Maierhofer, F. Ricciato, A. Baiocchi, *Towards Bottleneck Identification in Cellular Networks via Passive TCP Monitoring* Lecture Notes in Computer Science, vol. 8487, pp. 72-85, 2014

[16] B. Constantine, G. Forget, R. Geib, R. Schrage, *RFC 6349: Framework for TCP Throughput Testing*

[17] A. Afanasyev, N. Tilley, P. Reiher, L. Kleinrock, *Host-to-Host Congestion Control for TCP*, IEEE Communication Surveys and Tutorials, vol. 12, no. 3, pp. 304–342, July 2010

[18] R.S. Prasad, M. Jain, C. Dovrolis, *Socket Buffer Auto-Sizing for High-Performance Data Transfers* Journal of Grid Computing, 2003, vol. 1, Issue 4, pp 361-376

[19] J. Semke, M. Mathis Mahdavi, *Automatic TCP Buffer Tuning*, Computer Communication Review, ACM SIGCOMM, vol. 28, no. 4, October 1998

[20] M.K. Gardner, W.-C. Feng, M. Fisk, *Dynamic Right-Sizing in FTP (drsFTP): Enhancing Grid Performance in User-Space* Proc. of IEEE Symposium on High-Performance Distributed Computing, July 2002

[21] M. Mathis, R. Reddy, *Enabling High Performance Data Transfers* Jan. 2003; available at: http://www.psc.edu/networking/perf tune.html

[22] M. Fisk, W. Feng, *Dynamic Right-Sizing: TCP Flow-Control Adaptation* Proc. of the 14th Annual ACM/IEEE SC2001 Conf., November 2001

[23] E. Weigle, W. Feng, *A Comparison of TCP Automatic Tuning Techniques for Distributed Computing* Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing, 2002

[24] Linux kernel 3.18, https://www.kernel.org/

[25] M. Hirabaru, *Impact of Bottleneck Queue Size on TCP Protocols and Its Measurement*, IEICE Trans. of Commun., vol. E89-B, no. 1, Jan 2006

[26] Yi Wang, Guohan Lu, Xing Li , *A Study of Internet Packet Reordering* Lecture Notes in Computer Science, vol. 3090, pp. 350-359, 2004

[27] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, D. Towsley, *Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone* IEEE/ACM Trans. Netw. 15, 1 (Feb 2007), 54-66. doi: 10.1109/TNET.2006.890117

[28] M. Mathis, J. Heffner, *RFC 4821: Packetization Layer Path MTU Discovery*

[29] N. Hu , Li (Erran) Li, Z. Morley Mao, P. Steenkiste, J. Wang, Locating Internet Bottlenecks: Algorithms, Measurements, and Implications SIGCOMM Comput. Commun. Rev. 34, 4 (August 2004), 41-54, doi: 10.1145/1030194.1015474

[30] N. Hu, P. Steenkiste, Evaluation and Characterization of Available Bandwidth Probing Techniques IEEE Journal on Selected Areas in Communications, vol. 21, no. 6, August 2003