Measuring TCP Tail Loss Probe Performance

Andre Ryll, B.Eng.
Betreuer: Lukas Schwaighofer, M.Sc.
Seminar Future Internet WS2013
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: andre.ryll@tum.de

ABSTRACT
This paper analyzes the performance of the TCP Tail Loss Probe algorithm proposed by Dukkipati et al. in February 2013 under various virtual network conditions. To provide accurate and repeatable measurements and varying network conditions the mininet virtual network is used. Variations include the available bandwidth, round trip time and number of tail loss segments. The tests are done by requesting HTTP data from an nginx web server. Results show that TLP is able to decrease the total transfer time in high-speed networks by 38% and the time until data is retransmitted by 81%. These improvements decrease significantly for higher delay links.

Keywords
TCP, TLP, performance, measurements, comparison, mininet, virtual network, HTTP, iptables, netfilter

1. INTRODUCTION
Loss of data in a network transfer is a general challenge in all connection-oriented protocols. For Internet traffic TCP [1] is used as the transport layer for HTTP data. There exist a number of specifications which deal with retransmission behavior of TCP (e.g. [2], [3], [4]). This list has lately been extended by the "Tail Loss Probe" (TLP) algorithm [5]. The TLP internet draft suggests a real-world improvement of the average response time by 7% based on measurements on Google Web servers over several weeks. This paper aims at precisely measuring the response time improvement under various well-defined laboratory conditions to examine benefits and drawbacks of TLP. Variations will include link quality (bandwidth, delay) and the number of lost tail packets. The measurements are done on a single XUbuntu 13.04 Linux machine with a 3.10.6 kernel. To create a simulation with multiple virtual hosts the mininet virtual network is used. Simple HTTP data transfer is accomplished by an nginx1 web server and the lynx text browser. A user space C/C++ application in conjunction with iptables and netfilter queues allows to precisely drop a specified number of packets at the end of a transfer.

The remainder of this paper is organized as follows: Section 2 reviews the TCP protocol, extensions to it which are essential for TLP and the TLP algorithm itself. Section 3 describes the test setup for data acquisition. This includes mininet, iptables, the user space application and the measurement variations. Section 4 presents the results and the advantages of TLP. Finally section 5 sums up the insight of the measurement results and outlines further options for analysis.

2. TCP
The Transmission Control Protocol (TCP) is a reliable, connection-oriented transport protocol (ISO OSI layer 4 [6]). As it is stream-based it works with bytes (grouped in segments). Higher level protocols can transmit packets over TCP (e.g. HTTP), but TCP itself is not aware of packets. Its counterpart is the simpler User Datagram Protocol (UDP) which works connection-less and packet-based. This section aims at providing an overview of TCP and explains details which are important to understand TLP.

To implement the reliability and retransmission-capabilities the TCP header includes, amongst others, the following important fields:

- **SYN flag** Synchronize sequence number, set once during connection setup to set the initial (arbitrary) sequence number.
- **FIN flag** No more data from the sender.
- **ACK flag** The ACK field is valid. Indicates that this segment acknowledges received data. Always set except in the first segment of a transmission.
- **ACK field** The next expected sequence number of the receiver.
- **SEQ field** segment sequence number. The sequence number in the first (SYN) segment minus the current sequence number indicates the packet data offset in the current stream.

Figure 1 shows a graphical TCP flow representation created by the network analyzer wireshark2. In this example a client requested a web page from a server. Flags, content length and transfer direction of a segment are indicated in the green area. The white area shows the sequence and acknowledgment numbers of every segment relative to the first captured segment (implicitly done by wireshark to simplify reading). The connection is established in the first three transfers (TCP 3-way handshake). Afterwards the client sends a HTTP GET request (in this case with a length of

1http://nginx.org/

2http://www.wireshark.org/
It is important to note, that the client acknowledges every segment in this example. This is not required by the TCP specification. It is sufficient to acknowledge every second segment, given that the segments come in within a short time (RFC1122 section 4.2.3.2. specifies 500ms [7], Linux uses a dynamic approach with a maximum of 200ms [8]). The example transfer is also loss-free. To handle data loss several methods exist, which will be outlined in the following.

The original specification only retransmits segments if they have not been acknowledged after a specified time. This is called retransmission timeout (RTO). Several extensions have been made since TCP was initially specified to improve the retransmission behavior. This includes, amongst others: duplicate ACKs, originally specified in [9], later obsoleted by [2], selective ACKs, specified in [3] and early retransmission, specified in [4].

### 2.1 Duplicate ACK (DACK)

Duplicate ACKs are acknowledgments from a receiver with the same ACK number, which is not equal to the last expected one of the sender. That means as long as the sender has unacknowledged (but sent) data he expects the ACK number of the receiver to increase with every segment. This explanation is slightly simplified but sufficient for understanding the paper, for a full description see [2]. There exist a number of cases which may lead to duplicate ACKs. First of all, a segment may be lost and more data follows. As the receiver does not receive the segment it expects, he sends an ACK with the sequence number he actually expects. This is done for every segment received after the missing segment. Secondly, segments can arrive at the receiver in a different order than they were sent due to different paths of the segments through the network. Although all data arrives at the receiver this is an error condition as TCP has to be in-order. Lastly, duplicate ACKs may indeed acknowledge duplicate segments. This might for example be caused by a sudden increase in network delay. The transmitters RTO fires and resends a segment which then arrives twice at the receiver. To differentiate between duplicate ACKs from spurious retransmission, out-of-order reception and loss the transmitter waits for three duplicate ACKs before retransmitting data. This mechanism is known as fast retransmit as the transmitter does not wait for the retransmission timeout to fire but immediately resends the data. Fast retransmit is described in [2].

### 2.2 Selective ACK (SACK)

Duplicate ACKs can only inform the transmitter of the next expected sequence. Although more segments after the lost one might have arrived at the receiver the transmitter will need to retransmit data from the point where the first data was lost. To overcome this limitation the selective ACK (SACK) option was added to the TCP header [3]. To use SACK both communication partners need to support it. Every SACK-enabled host sets the SACK-permitted option in the SYN packet of the TCP-Handshake. If both hosts support this option it can be used in further communication. If a segment is lost and SACK is allowed the receiver still replies with duplicate ACKs but the ACKs will now have more information about which following segment was successfully received. The SACK option specifies up to three continuous blocks of data, that have been received after one or more missing blocks (holes). Each block uses a left edge (SLE) of data received and a right edge (SRE) one past the last byte received in that block, both of them are sequence numbers. The transmitter can use the SACK information to precisely resend only data that has been lost and avoids resending data which has been successfully received after a lost segment.

### 2.3 Early Retransmit (ER)

Selective ACKs provide additional information to the transmitter of data in case of a lost segment. Nevertheless they do not speed up the time until a segment is resend. They help to inform the sender which segments need to be resend. To resend a segment the RTO or three duplicate ACKs (fast retransmit) are still used. The aim of the early retransmit (ER) algorithm [4] is to lower the number of duplicate ACKs which are needed to retransmit a segment. To achieve this the ER algorithm tries to estimate the remaining number of segments which can be sent. This depends on how much data is available for sending and how much data is allowed to be transmitted before being acknowledged (so called window size). The ER algorithm does not depend on SACK although it can be used with SACK to calculate a more precise estimate of the remaining number of segments. If the window size or the data left is too small to achieve at least three segments in flight, then fast retransmit will never occur as there is no way to generate three duplicate ACKs. In this situation ER reduces the number of required duplicate ACKs to trigger fast retransmit to one or two segments.
2.4 Forward ACK (FACK)
If the window size is large enough and there is enough data to
send the retransmission of data still requires at least three
duplicate ACKs. To improve this behavior the forward
ack (FACK) algorithm has been proposed [10]. FACK re-
quires SACK in order to work. The FACK algorithm mon-
itors the difference between the first unacknowledged seg-
ment and the largest SACKed block (the forward-most byte,
hence its name). If the difference is larger than three times
the maximum segment size of a TCP segment, then the first
unacknowledged segment is retransmitted. If exactly one
segment is lost this will happen after receiving three dupli-
cate ACKs. So for only one lost segment FACK and fast re-
transmission based recovery trigger at the same time. The
main advantage of FACK is a situation with multiple lost
segments. In these situations it requires only one duplicate
ACK when three or more segments are lost to start a recov-
ery.

2.5 Tail Loss Probe (TLP)
All previous solutions to recover lost data are based on the
reception of duplicate ACKs to retransmit data before the
retransmission timeout (RTO) expires. In situations where
the last segments of a transfer (the tail) are lost, there will
be no duplicate ACK. So far the only option to recover from
such a loss is the RTO. The Tail Loss Probe (TLP) al-
gorithm [5] proposes an improvement to such situations by
issuing a “probe segment” before the RTO expires. If multiple
segments are unacknowledged and the TLP timer ex-
pires the last sent segment is retransmitted. This is the basic
idea of the TLP algorithm. Further actions in response to
the probe segment are handled by the previously described
mechanisms. If exactly one segment at the tail is lost, the
probe segment itself repairs the loss, a normal ACK is re-
ceived. If two or three segment are outstanding ER will
lower the threshold for fast retransmit and the duplicate
ACK of the probe segment triggers early retransmission. If
four or more packets are lost the difference between the last
unacknowledged segment and the SRE in the SACK of the
probe segment will be large enough to trigger FACK fast
recovery. In theory the TLP improves response time to loss
in all cases. Table 1 sums up the different options.

<table>
<thead>
<tr>
<th>losses</th>
<th>after TLP</th>
<th>mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAA</td>
<td>AAAA</td>
<td>TLP LOSS DETECTION</td>
</tr>
<tr>
<td>AALL</td>
<td>ALS</td>
<td>ER</td>
</tr>
<tr>
<td>ALLL</td>
<td>ALLS</td>
<td>ER</td>
</tr>
<tr>
<td>LLLL</td>
<td>LLS</td>
<td>FACK</td>
</tr>
<tr>
<td>&gt;=5 L</td>
<td>...LS</td>
<td>FACK</td>
</tr>
</tbody>
</table>

Table 1: TLP recovery options. A: ACKed segment, L: lost segment, S: SACKed segment [5]

3. TEST ENVIRONMENT
To evaluate the performance of TLP a network environment
and a possibility to drop tail segments is required. As a
physical test setup is hard to reconfigure and inflexible with
respect to e.g. bandwidth limitation a network simulation
tool has been chosen. All tools are compatible with the
Linux operating system, thus a Ubuntu 13.04 32-Bit ma-
chine with a 3.10.6 kernel is used for the tests.

3.1 Tool Overview
Two network simulation tools have been investigated. The
ns-3 network simulator\(^3\) and the mininet virtual network\(^4\).
ns-3 provides a lot of features for automated testing and data
acquisition, although it requires some effort to write a test
program. As ns-3 is a simulator the complete network runs
in an isolated application. All test programs and algorithms
need to be implemented in C/C++ to make them available
for measuring. The major drawback of ns-3 is, that it cannot
easily interface a recent Linux kernel that supports TLP.
Thus ns-3 could not be used for testing TLP performance.

Mininet provides a lightweight virtual network by using func-
tions build into the Linux kernel. Unlike ns-3 it is not a
single application but a virtualization technique that allows
to create separate network spaces on a single host machine.
They share the same file system but processes are executed
in their isolated space with specific network configurations.
It allows to create everything from simple networks (e.g. 2
hosts, 1 switch) up to very complex topologies, only limited
by available processing power. Furthermore it is easily re-
configurable, uses the underlying Linux kernel and can run
any Linux program in a virtual host. The Linux traffic con-
trol interface can be used to specify delay, bandwidth and
loss on a virtual connection. All properties make it ideally
suited for TLP performance analysis.

The Linux traffic control interface is not able to precisely
drop segments at the end of a transfer. There are two op-
tions to achieve this. Mininet switches can be used together
with an OpenFlow\(^5\) controller which usually tells the switch
how to forward packets by installing rules based on e.g. the
MAC addresses in a packet. If a packet does not match a
rule it is forwarded to the OpenFlow controller, which inves-
tigates the packet and afterwards installs an appropriate rule
in the virtual switch. By not installing any rule this can be
used to forward all packets passing the switch to the Open-
Flow controller which then determines if the packet is at the
tail of a transfer and if so drops it. This mechanism has a
poor performance, because usually only very few packets are
forwarded to the controller for learning and installing appro-
priate rules afterwards. So although this option works, it is
not adequate for rapid tail loss. Another option to drop seg-
ments is the use of Linux iptables\(^6\). As iptables are primarily
used for firewall purposes there is no build in option to drop
a configurable amount of tail segments. Nevertheless ip-
tables can forward packets to a user space application which
then decides to accept or to drop a packet. This is done by
using the netfilter queue (NFQUEUE) as a target. Using the
libnetfilter library a user space application can process these
packets and also access the complete packet content. This
option works locally on a (virtual) machine and uses kernel
interfaces, thus this approach is quite fast compared to the
OpenFlow solution. The user space application is written in
C/C++ and provides a good performance.

\(^3\)http://www.nsnam.org/
\(^4\)http://mininet.org/
\(^5\)http://www.opennetworking.org/
\(^6\)http://www.netfilter.org/projects/iptables/
3.2 Setup
The final test setup uses mininet with two hosts and one switch on a XUbuntu 13.04 machine with kernel 3.10.6 and a netfilter user space application using iptables. The setup is depicted in figure 2. To configure this setup the following steps are necessary:

First of all mininet must be started with a configuration of two hosts and one switch. This is done by the command:
```
mn --topo single,2
    --link tc,bw=100,delay=2.5ms
```
This configures mininet with a link bandwidth of 100MBit/s and a delay of 2.5ms per link. Thus the round trip time is 10ms. This setup reflects a common high-speed ethernet environment. The two hosts are named h1 and h2. A terminal to the two hosts can be opened via (entered in the mininet console):
```
xterm h1 h2
```
h1 serves as a web server which is started by typing “nginx” in its command window. Furthermore iptables needs to be configured to pass outbound HTTP traffic (TCP port 80) on interface h1-eth0 to a netfilter queue.
```
iptables -A OUTPUT -o h1-eth0 -p tcp --sport 80
    -j NFQUEUE --queue-num 0
```
This forwards all TCP traffic leaving h1 to NFQUEUE 0. The iptables and filter setup on virtual host one (h1) is depicted in figure 3. The inbound traffic is passed directly to nginx, whereas the outbound traffic is either accepted directly (non-TCP) or forwarded to the NFQUEUE. The user space application is named “tcpfilter” and can be configured by command line arguments to drop a specified number of packets at the end of an HTTP transfer (e.g. two packets).
```
./tcpfilter 2
```
Implementation details of the tcpfilter are explained in section 3.3. To request a web page from h1 the lynx web browser with the dump option is used on h2. It just requests the web page and dumps it to /dev/null.
```
lynx -dump 10.0.0.1/pk100.html > /dev/null
```
This is repeated several times in a shell script to automatically acquire a set of data. This finalizes the test setup.

3.3 Tail Loss application
The tail loss application tcpfilter is a custom application written in C/C++. It accesses the netfilter queue 0 and processes its packets. For this purpose the complete packet is copied to user space. After packet processing is done it issues a verdict on every packet which can either be ACCEPT or DROP. If drop is selected the kernel silently discards this packet. As this application works on the OUTPUT chain of iptables the packet then never leaves the network interface. This is a simulated packet loss. The number of dropped segments n_drop is specified as command line argument to tcpfilter. The algorithm used to generate tail loss is shown in algorithm 1.

As TCP is stream based there is no way of determining the last segment. Thus HTTP is used in the application layer to find the last segment of a transfer. The filter is initially in the idle state. As soon as a HTTP segment (TCP on port 80) is going to be sent it checks the contents of that segment for the string “Content-Length”. This indicates that this is the header of a new HTTP transfer. Wireshark analysis show that the content length is always set by nginx for HTML data transfer. It is thus safe to use this field as header indication. The content length is extracted from the header and saved to further track the incoming data. Furthermore the HTTP data length is of this segment is saved to know the maximum data size of a segment.

The filter is now in the transfer state. It is locked on to one transfer by saving its source and destination (IP and port) and the TCP identification. Data that is not belonging to this transfer is accepted, and not processed further. Data for the current transfer is tracked and accepted until it reaches the end of the transfer. As soon as the total transfer size minus the size of the transferred data is smaller than the number of segments to lose at the end times the maximum HTTP size of a segment, the segments are saved in a linked list and no verdict is issued. After a TCP segment for this transfer arrives which has the FIN flag set the saved segments are processed. If the segment with the FIN flag also has HTTP data n_drop segments at the tail of the list are dropped, otherwise n_drop + 1 segments at the tail are dropped. The tcpfilter application thus always drop n_drop packets with HTTP data. After the FIN segment the filter enters idle state again and is ready to track the next transfer.
While the drop candidates are in the list, no verdict is issued which may lead to a delay in sending packets. An analysis of the traffic shows that the window size for this transfer is large enough so that the sender transmits more than 20 segments before waiting for an acknowledgment. The time between the first segment in the drop candidate list and the processing due to receiving the FIN flag is thus very short and should not affect the measurements.

Algorithm 1 Creating tail loss

```
Packet p
TransferState s
List candidates
List droppedOnce
if !isHTTP(p) then accept(p)
if exist(p.seq, droppedOnce) remove(p.seq, droppedOnce)
accept(p)
if state == idle and exist(p, "Content-Length") then
  state = xfer
  s.maxHttpSize = p.httpSize
  s.src_dst = p.src_dst
  s.totalLength = extract(p, "Content-Length")
  s.transferred = 0
if state == xfer and p.src_dst == s.src_dst
  bytesRemaining = s.totalLength - s.transferred
  bytesToDrop = n_drop * s.maxHttpSize
  if bytesRemaining < bytesToDrop then
    enqueue(p, candidates)
  else
    accept(p)
  if s.totalLength == s.transferred then
    state = fin
if state == fin and p.fin then
  if p.httpSize == 0 then
    n_drop++
  while size(candidates) > n_drop do
    accept(front(candidates))
  while size(candidates) > 0 do
    enqueue(front(candidates), seq, droppedOnce)
    drop(front(candidates))
  state = idle
```

During transfer processing several nanosecond-accurate timestamps are taken. The first one \( t_{\text{start}} \) during the transition from idle to transfer. The next are recorded for every drop candidate. The timestamp of the first segment is finally selected to be dropped is saved to \( \text{isTLP} \). The change take effect immediately, so there is no need to restart mininet or the whole system. Table 2 shows the configuration for the different algorithms used.

```
sysctl -w net.ipv4.tcp_early_retrans=2
```

The changes take effect immediately, so there is no need to restart mininet or the whole system. Table 2 shows the configuration for the different algorithms used.

<table>
<thead>
<tr>
<th>Option</th>
<th>plain</th>
<th>ER</th>
<th>TLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_early_retrans</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>tcp_fack</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>tcp_sack</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: TCP configuration in /proc/sys/net/ipv4

To evaluate the performance under various network conditions three exemplary types are selected as shown in table 3. They do not necessarily reflect real networks but cover a broad range of different conditions. To acquire the measurement dataset all TCP configurations are tested with all network configurations. The tests increase the number of lost tail segments from 0 to 20 and record the time \( t_{\text{recover}} \) until the first segment is resend and the total transfer time \( t_{\text{total}} \). Due to the usage of the mininet virtual network there is no “natural” tail loss during the measurements.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bandwidth</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>high-speed</td>
<td>100MBit/s</td>
<td>10ms</td>
</tr>
<tr>
<td>mobile</td>
<td>7.2MBit/s</td>
<td>100ms</td>
</tr>
<tr>
<td>satellite</td>
<td>1MBit/s</td>
<td>800ms</td>
</tr>
</tbody>
</table>

Table 3: Network configurations
4. RESULTS
The results are acquired by repeating the measurements for the high-speed and mobile network configuration 100 times and 20 times for the satellite network. The reason for only acquiring 20 samples per number of tail losses in the satellite network is the high round trip time. It takes approximately one hour to obtain a dataset with 420 samples. Table 4 sums up the measurements in the different networks with a loss count of five segments. The previously default option of ER in the Linux kernel is the baseline for comparisons. Tail Loss Probe performs best when the round trip time is low. The total transfer time is decreased by 38% in the high-speed network. On a mobile network the time is still 11% lower. The satellite network does not benefit significantly from TLP. Early Retransmit does not improve the transfer time very much compared to plain TCP configuration. This is expected, as ER requires partial information of the received data and duplicate ACKs. Both conditions are not available at a tail drop. When comparing the time to the first retransmission $t_{\text{recover}}$, TLP reduces the value significantly by 81% (high-speed network). In the mobile network this reduction drops to 19%. A noticeable anomaly is the increase of the recovery time in the plain configuration in the mobile network. As this paper mainly deals with TLP, the evaluation of this anomaly is out of scope.

<table>
<thead>
<tr>
<th>TCP cfg.</th>
<th>plain</th>
<th>ER</th>
<th>TLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>100MBit/s, 10ms RTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{\text{total}}$</td>
<td>0.3595</td>
<td>+0.6%</td>
<td>0.3574</td>
</tr>
<tr>
<td>$t_{\text{recover}}$</td>
<td>0.2396</td>
<td>+1.3%</td>
<td>0.2365</td>
</tr>
<tr>
<td>7.2MBit/s, 100ms RTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{\text{total}}$</td>
<td>1.2091</td>
<td>-1.7%</td>
<td>1.2300</td>
</tr>
<tr>
<td>$t_{\text{recover}}$</td>
<td>0.4392</td>
<td>+7.8%</td>
<td>0.4073</td>
</tr>
<tr>
<td>1MBit/s, 800ms RTT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_{\text{total}}$</td>
<td>8.2145</td>
<td>-0.1%</td>
<td>8.2248</td>
</tr>
<tr>
<td>$t_{\text{recover}}$</td>
<td>2.4792</td>
<td>+0.3%</td>
<td>2.4720</td>
</tr>
</tbody>
</table>

Table 4: Recovery algorithm comparison (transfer size: 100 packets, losses: 5). ER configuration serves as baseline.

![Figure 4](image1.png)

Figure 4: Number of tail losses and time until the first segment is retransmitted. Plain (green), ER (blue), TLP (red). Transfer size 100 segments. High-speed network.

![Figure 5](image2.png)

Figure 5: Number of losses and total transfer time. Plain (green), ER (blue), TLP (red). Transfer size 100 packets.
Figure 4 plots the recovery time versus the number of lost tail segments. The standard deviation is plotted along for each measurement. For a better reading the samples have been slightly shifted on the plot, but the number of losses is always an integer. The results show that the plain implementation and ER perform almost equally. TLP is faster with a factor of approximately 4.5.

Of special interest is the behavior of TLP with a loss of exactly one segment. In this case TLP increases the retransmission timer to accommodate for an eventually delayed ACK. TCP can concatenate two ACKs into a single one if data comes in within short time. To make this concatenation possible the TCP implementation in the Linux kernel waits up to 200ms [8]. This is also the value the TLP retransmission timer is increased when only a single segment is in flight (cf. WCDelAckT in [5], sec. 2.1). Although the data loss is repaired by the tail loss probe segment, it takes approximately twice the time until the transfer is complete (compared to multiple segments in flight).

Figure 5 compares the total transfer time in the three network configurations. In the case of no loss all implementations are equally fast. This also shows that the additional TLP code has no impact on lossless transfers. As noted previously TLP performs bad with a single lost segment. An interesting trend in the mobile and in the satellite network is the slightly increasing transfer time in dependence of the number of lost segments. The tcpsplitter drops all segments at once and after the duplicate ACK from the tail loss probe segment ER should immediately resend all segments. Thus the number of tail loss segments should not have such a significant impact. Furthermore the increase in transfer time is not linear. The time increases after 1, 2, 4, 8 and probably 16 packets, which are all exponentials of 2. The reason for the increasing time is most likely the congestion control algorithm used, but this topic is out of the scope of this paper.

The measurements in the satellite network also show that TLP has no substantial benefit for high-delay lines. The recovery time of 2.4s is also higher than expected by the TLP paper. If the flight size is greater than one segment TLP calculates the retransmission timer by multiplying the smoothed RTT by two. This would be 1.6s for the satellite network. For the mobile network this is 0.2s, the measurements show an average of 0.33s. In the high-speed network it should be 0.01s, measured is 0.045s. So the Linux TLP implementation always calculates a higher retransmission timer than specified in the paper. This does not have to be an error in the TLP code but can also be the consequence of the RTT measurement implementation in the Linux kernel.

5. CONCLUSION

The results show that the Tail Loss Probe is an improvement to TCP communication in all tested cases. There is no situation where a non-TLP test result is better. The largest improvement in the time until the first segment is retransmitted (~81%) is recorded in the high-speed network. Total transfer time in such a network with a tail loss is decreased by 38%. The TLP draft reports real-world values from a test with the Google web servers of up to 10% improvement in response time. The values presented in this paper are not comparable to the TLP draft values because in the TLP draft the values are calculated from all transmissions, including those without any tail loss. To compare them one needs to know how many of the transmissions encountered tail loss. It is important to note that TLP has no benefit for a single lost tail segment. Furthermore in high RTT networks TLP does not improve the transfer time. Although especially in these networks a decrease in transfer time would be a great advantage.

The tests included only a small variation of possible measurements. Further options are the transfer size, although this should have no effect with a constant number of tail drops, a generally lossy line or variations in the transfer window size. Furthermore this paper intentionally left out aspects of congestion control and TLP’s interference with it. Measurements in this domain require a much complexer user-space application.

6. REFERENCES