This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Schulte, Lennart; Zimmermann, Alexander; Nanjundaswamy, Puneeth; Eggert, Lars; Manner, Jukka

I'll be a bit late - Packet reordering in mobile networks

Published in:
Journal of Communications and Networks

DOI:
[10.1109/JCN.2017.000108](https://doi.org/10.1109/JCN.2017.000108)

Published: 01/12/2017

Document Version
Peer reviewed version

Please cite the original version:
Schulte, L., Zimmermann, A., Nanjundaswamy, P., Eggert, L., & Manner, J. (2017). I'll be a bit late - Packet reordering in mobile networks. *Journal of Communications and Networks*, 19(6), 693-703. [8277337].
<https://doi.org/10.1109/JCN.2017.000108>

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

I'll be a bit late – Packet Reordering in Mobile Networks

Lennart Schulte, Alexander Zimmermann, Puneeth Nanjundaswamy, Lars Eggert, Jukka Manner

Abstract: Data transfer in mobile networks has increased significantly over the past years, and the capacity of these networks has grown accordingly. Due to their different set of characteristics compared to wired networks, they have also received attention from end-to-end developers on transport and application level. However, there exists no research on packet reordering in modern mobile networks, and how a transport layer should be designed to cope with it. In this paper we discuss how to measure packet reordering in mobile networks with the help of TCP. Our analysis in Finnish mobile networks shows that reordering characteristics depend heavily on the operator, but is present in all tested networks, and negatively impacts TCP when not handled. We find that both number of reorderings in a connection and the extent of the reordering is influenced by the sending rate. Additionally, we present the framework for a reordering prevention algorithm which takes into account the findings by automatically adapting to the current sending rate.

Index Terms: Measurement, mobile networks, packet reordering, TCP

I. Introduction

Mobile networks are of ever increasing significance for our everyday live, as people rely on it to stay in contact, search for information or access their data. The most recent commercially deployed mobile technology is 4G LTE, which significantly increases throughput and reduces latency compared to 3G networks.

These networks have received major attention from end-to-end developers in past years due to their interesting set of characteristics that are completely different from wired networks and cause problems for current transport layers and applications [1]. Still, there are characteristics which have been analyzed in wired networks, but not yet in the mobile network environment. In this paper, we analyze packet reordering in mobile networks. While it received attention in the past in wired networks, there has not been any recent work showing the extent of this characteristic for modern mobile networks (see Section IV).

Huge parts of the current internet, including mobile networks, are based on the Internet Protocol (IP). Due to its nature of being packet switched, the network by definition does not guarantee packets to arrive in the same order as they are being sent, e.g. due to path changes or load balancers (see Figure 1). The

Fig. 1. A packet is routed along another path.

occurrence of packet reordering can cause performance issues and throughput degradation to the Transmission Control Protocol (TCP) [2] and protocols with similar behavior like SCTP if not handled carefully [3], as it causes the protocols to falsely assume congestion in the network. The sending rate is then reduced unnecessarily, which causes the protocol to underperform.

TCP is one of the most widely deployed protocols operating in a huge variety of environments, including mobile networks, where it is used as the underlying protocol for most applications [1]. Therefore, we use it to explain the impact of packet reordering and measure it. TCP has received a huge amount of extensions to cope with a number of different situations. Some of these help to find and analyze packet reordering, i.e. selective acknowledgment (SACK) [4], duplicate selective acknowledgment (DSACK) [5], and the Timestamps option [6]. While these options have been used before to measure reordering, we present a complete measurement methodology as the *first contribution* of this paper. We also point out that each of these options alone (even DSACK or Timestamps) is not enough to accurately describe the reordering characteristic.

The *second contribution* is an investigation into the reordering characteristics in mobile networks. To the best of our knowledge this is the first such extensive study. We analyze traces over a period of one year from several Finnish mobile networks and find that the number of reorderings per connection is dependent on the sending rate. The reordering rate can be as high as 6%. We show that up to 10 % of HSPA connections and up to 23 % of LTE connections are facing situations where TCP would reduce the sending rate unnecessarily due to reordering depending on operator network. In addition, we take a deeper look at the reordering extent and conclude that it is also dependent on the current sending rate of the connection, i.e. with a higher sending rate the reordering extent is potentially higher, meaning that higher speed networks are more likely to experience reordering that forces TCP into reducing its sending rate.

Based on the above seen characteristics, we then show a reordering prevention algorithm as our *third contribution*, which adaptively sets the reordering extent based on the sending rate. All previous algorithms have an absolute threshold or are not

Manuscript received MONTH DAY, YEAR; approved for publication by Editor Name, X Editor, MONTH DAY, YEAR.

L. Schulte and J. Manner are with Department of Communications and Networking, Aalto University, Finland, email: {firstname.lastname}@aalto.fi

A. Zimmermann has no affiliation (results accomplished while at NetApp), email: alexander.zimmermann@rwth-aachen.de

P. Nanjundaswamy and L. Eggert are with NetApp, Munich, Germany, email: {firstname.lastname}@netapp.com

taking the network state into account. This approach avoids the need for a new sample if the sending rate increases, and reduces unnecessarily delaying retransmissions or even retransmission timeout (RTO) when the sending rate decreases.

Section II explains the problem of reordering, and Section III shows how to detect and measure reordering with TCP. After reviewing related work in Section IV, Section V presents our measurements study of reordering in mobile networks. In Section VI we show a reordering prevention algorithm that makes use of our findings.

II. Impact of Packet Reordering

TCP guarantees that the data the application sends reaches the receiver correctly and completely. To do so the TCP receiver generates cumulative acknowledgments (ACKs) to tell that all the data up until a certain point has been received properly [2]. When some data has been lost in the network the receiver sends duplicate acknowledgments (DUPACKs), informing the sender that data has been received, but it is not the next expected data. To account for minor packet reordering the TCP sender waits until the third such DUPACK, called the duplicate acknowledgment threshold (DUPTHRESH). Then, the oldest not acknowledged data is deemed lost and the sender performs a retransmission [7]. Such a situation is always assumed to be caused by congestion in the network, and hence, the sending rate is reduced to not overload the network. When the assumption of congestion loss does not hold because the missing packet was caused by e.g. packet reordering, severe performance degradation can occur [8], [3], [9].

Due to the internet being IP based, the forwarding of packets is not determined on a connection level but for every packet separately. This in turn means that some packets might take a different path, e.g. due to routing changes and handovers, or even inside of multicore switches. This causes the arrival of packets to be different from the order they are sent in, which is called packet reordering. The reordered packet therefore arrives later than earlier sent packets, and causes the receiver to generate a DUPACK for every packet received out-of-order. For the sender, this event is not distinguishable from a congestion loss without additional effort. Hence, if the number of received DUPACKs exceeds the DUPTHRESH, the TCP sender assumes loss and *spuriously* retransmits the data by initiating fast retransmit and fast recovery [7], [10], and reduces the sending rate which unnecessarily reduces throughput.

Figure 2 illustrates how a packet reordering is perceived by TCP. Packets 1 to 5 are sent out, and the receiver then generates an ACK for packet 1. But when packets 3 to 5 arrive, the receiver has not yet seen packet 2. Hence, a DUPACK is sent in response for each of those out-of-order packets. For the first two DUPACKs the sender transmits new data according to the Limited Transmit algorithm [11]. On the third DUPACK, the sender retransmits the missing data and enters fast recovery, since at this point there is no information available to distinguish packet loss from packet reordering.

Besides causing spurious fast retransmission and fast recovery, packet reordering has other negative effects on TCP performance. These include the interruption of TCP's ACK clock, less accurate round trip time (RTT) estimation, increased pressure

Fig. 2. TCP enters recovery due to a late packet

Fig. 3. Detecting reordering with TCP SACK option

on the receiver buffer and slower congestion window (CWND) growth due to a reduced number of acceptable ACKs. Several papers [8], [3], [9] discuss these negative effects.

In order to achieve high interoperability with standardized TCP, the preferred method to mitigate the performance reduction due to reordering is increasing the DUPTHRESH: TCP waits for more DUPACKs before a retransmission and congestion response are initiated, and hence postpones the decision whether it was reordering or actual loss. This is why the extent of the reordering, in addition to its occurrence, is a very important metric for reordering prevention algorithms.

III. Detecting Packet Reordering

The previous section showed that reordering causes performance problems when packets are affected by packet reordering. In this section, we investigate which TCP options are useful in detecting and measuring the reordering characteristic and illustrate this in a descriptive way. The full algorithmic details are available [12] as is the implementation [13].

A. Selective Acknowledgements

The first notable TCP option is SACK [4]. This option is very important for an efficient fast recovery procedure, as it al-

lows retransmitting lost packets much faster [10] and is therefore widely used. It provides a method for the receiver to report data that has been received out-of-order, as presented in Figure 3. As before, five packets are sent out, but packet 2 and 4 arrive late. The receiver responds again with an ACK for the first packet, and DUPACKs for the next two packets, since they arrive out-of-order. The SACK option is indicated in purple. For example, when sending the second DUPACK the receiver informs the sender with SACKs that it has received packets 3 and 5 correctly, which implicates that only packets 2 and 4 are missing. Thereafter, the late packet 2 arrives, for which a new cumulative ACK 3 is sent including the SACK for packet 5.

When only the SACK option is available, it is possible to detect only those reordering events where no retransmissions has yet occurred. If the packet was retransmitted and the ACK for this packet arrives, TCP has no information if the original transmission was acknowledged or if it was in response to the retransmission. The first would result from a reordered packet, the second from a lost packet. This is commonly referred to as the retransmission ambiguity [14]. SACK does not solve this problem and is therefore only useful if there is no ambiguity. Situations like this can happen, since TCP waits until the third DUPACK.

Still, calculating the *extent* of the reordering accurately is only possible when the connection uses SACKs. The extent of the reordering gives a measure on how late a reordered packet is. It can be given in byte or packets, then it is simply called the reordering extent, and in units of time called the reordering delay. Each time a reordering is detected we can calculate both metrics. The reordering extent is the distance between the reordered packet and the highest packet that has reached the receiver. As reordering delay we define the time between arrival of the ACK for a packet and the time it normally should have arrived. We approximate the expected arrival time by setting it to the time the first SACK is received which is higher than the reordered packet. The calculation of the reordering extent and reordering delay are in line with the definition in RFC 4737 [15].

In Figure 3 we can calculate the extent of the reordering. For packet 4 this is done when ACK 5 arrives. The packet is one packet late, since the highest packet that has reached the receiver is 5. Packet 2 is ACKed with the arrival of ACK 3 which carries SACK 5. Therefore, the packet is three packets late, since the highest received packet is 5 as indicated by the SACK option. Here, the importance of the SACK option is visible: when considering only the cumulative ACK, we would use packet 3 as the highest one and underestimate the reordering extent. When calculating the reordering delay for packet 2, we take the time between the arrival at the sender of ACK 1 SACK 3 and ACK 3 SACK 5. For packet 4 it is ACK 1 SACK 3,5 and ACK 5.

B. Duplicate SACK

An extension to SACK is the DSACK option [5], which uses the same format. But instead of providing information about out-of-order data, it is sent in response to receiving a packet multiple times. This enables detection of a spurious retransmission: if a DSACK is received for a retransmitted packet, then the sender knows that the original transmission and the retransmission have arrived and implicitly the original transmission was

Fig. 4. Detecting reordering with TCP DSACK option

not lost, but reordered [16]. Figure 4 shows such a situation. Packet 2 arrives late and triggers a retransmission at the sender. When the retransmission of packet 2 arrives at the receiver, a DUPACK is sent with the information that packet 2 has been received again.

Unfortunately, in contrast to SACK, the DSACK option is not negotiated during connection establishment. Therefore, before a connection has seen a DSACK, we can not deduce that there was no reordering: it might be that the receiver just does not support DSACKs and has therefore not sent any in response to reordering events. But if we ever recognize a DSACK, then we know that it has been available throughout the connection and can rely on its information. Additionally, while a lost ACK is covered by the next arriving ACK, a DSACK is only transmitted once. Hence, if the ACK containing the DSACK is lost, no reordering can be detected with this method. Still, the reliability of modern mobile networks is very high [17], making a DSACK lost due to mobile networks unlikely.

When measuring the extent of the reordering, it has to be thought about at which point to do this. When looking at Figure 4, we can see that it is the late original transmission that fills in the missing data. But the DSACK arrives only after the arrival of the retransmission. Hence, we have to calculate the extent of the reordering when the first ACK for packet 2 arrives, and not when the DSACK arrives, at which point the calculation is done the same as with SACK (see before). Then this has to be stored for when the reordering is later really detected. Since beforehand it can not be known if a retransmitted packet was spurious or not, the extent of the reordering has to be stored for *every* retransmission. Since this might cause a lot of overhead, it is generally not advised to implement it in the operating system.

C. Timestamps

The most powerful TCP option when detecting reordering is the Timestamp option [6]. With each packet a timestamp is sent. The receiver replies with the latest received timestamp that advances the cumulative ACKs point. Figure 5 shows an exam-

Fig. 5. Detecting reordering with TCP Timestamps option

ple. Packet 1 has timestamps T1, and this is echoed by the receiver when the ACK for packet 1 is sent (indicated by E1). The following DUPACKs are still sent with replying timestamp T1, since no more cumulative data has been received. Only when the late packet 2 arrives does this change and timestamp T2 is replied. When this packet was lost and not reordered, only then the retransmission with timestamp T9 would advance the cumulative ACK point, which would then cause the receiver to reply with T9 instead of T2. Hence, if the sender receives a new cumulative ACK during fast recovery and the timestamp is smaller than the one for the retransmission, then the original transmission has been reordered and not lost. It is therefore the only TCP option that eliminates the retransmission ambiguity. This is very similar to the Eifel detection algorithm [14], which is only defined for detecting spurious loss recovery, i.e. after an RTO, while this methodology is designed for retransmissions during fast recovery.

In contrast to the DSACK method, this detection works without delay. Hence, the extent of the reordering can be calculated when the reordering is actually detected and no more state is required. The calculation itself, again, is done as explained with SACK.

D. Late by more than RTT

One case stands out, as it can not be recognized by any of the above methods alone: the reordered packet is late by more than the RTT. While the above methods have been used in earlier papers, this is new.

Recalling how the TCP Timestamp option works, we have a look at Figure 5, but imagine that the reordered packet 2 arrives much later. Then, it might happen that the retransmission with timestamp T9 arrives before the reordered original retransmission, triggering an ACK with timestamp T9. This tells the sender that the retransmission was necessary and no reordering is detected.

The problem for the DSACK option is not detecting the reordering, but calculating the reordering extent accurately. The

Fig. 6. Detecting reordering when late by more than RTT

second arrival of packet 2 still triggers the sending of a DSACK, no matter which one arrives first. But the calculation of the extent of the reordering is different. Above, the extent has been calculated when the first ACK for the packet has been received, but this is not true in this case, since the reordered packet was received only when the DSACK was sent. Therefore, the extent has to be calculated at sender side at the time of the DSACK. With DSACK alone it is not possible to solve this ambiguity of which transmission arrives first, which poses a major problem for accurate measurements.

Still, when we combine both the information from timestamps and DSACK, we are able to solve the conundrum (see Figure 6), enabling us to reliably detect reordering even when it is longer than one RTT: at the arrival of the first ACK for a retransmission, we check with timestamps if reordering can be detected. If this is the case, then we are done and reordering has been detected. If no reordering has been detected, but later we see a DSACK for this packet, then we know that it has been delayed for a long time, and we calculate the extent of the reordering at the time the DSACK was received.

However, the ambiguity can not be solved with absolute certainty, since the DSACK option is not negotiated. When we see no DSACK during a connection it might still be that packets are delayed in a way that it can not be detected with timestamps alone, but the receiver does not implement this option.

The measurement study in Section V show the reordering detected in mobile networks based on the methodology discussed in this section. A proper reordering reaction based on the found characteristics is then outlined in Section VI.

IV. Related Work

Paxson [18] analyzes reordering in 35 locations. He concludes that (a) in two datasets, 12 % respectively 36 % of TCP connections experience reordering, (b) reordering is highly variable between locations, (c) reordering of a TCP connection is asymmetric and (d) the performance impact is often small.

Bennett et al. [8] measure reordering in the Internet Exchange Point (IPX) MAE-East. They send bursts of ICMP requests to 140 different servers over the course of several days. They find that (a) in 90 % of cases, the burst experiences reordering and (b) reordering in the IPX depends on the traffic, which may be caused by link aggregation.

Bellardo et al. [19] perform measurements with TCP con-

nections to 50 different Internet servers. They only checked the occurrence of reordering, not the extent and concluded that (a) 40 % of connections experience reordering, (b) the reordering probability depends on the inter-arrival time (IAT) and (c) reordering is asymmetric.

Gharai et al. [20] measure reordering from three locations all connected to the same Internet backbone with UDP. They only classify a packet as reordered if it would trigger TCP loss recovery. They conclude that (a) 47 % of transfers have at least one reordered packet, (b) the highest measured reordering percentage is 1.65 %, (c) there is no significant difference between the three paths in terms of reordering, (d) there is a strong correlation between reordering, the IAT and sending rate and (e) there is no linear correlation between the total number of reordering events and the number of spurious retransmission.

Jaiswal et al. [21] study out-of-sequence segments in 29 million TCP connections. Their result is that (a) between 12 % and 16 % of retransmissions have been spurious, (b) out-of-order segments are caused by reordering in 7 % to 26 % of cases and (c) 93 % of reordering events have an extent below three segments.

Reordering in high-speed wired networks is also discussed [22]. The authors analyze the average reordering extents and found that it slightly increases with the sending rate. Still, the important part is not the average, but that reordered packets are more likely to trigger a spurious recovery with the same reordering delay when the sending rate is higher, since one reordered packet with a large enough reordering extent is enough to spuriously enter recovery and lower the sending rate.

The above studies show that reordering is present in the internet. For mobile networks we have not found any similar work, except our own [13] where we discuss the impact of client context. While reordering has been mentioned in [1], they are measuring how late packets are while not distinguishing between reordering and packet loss. Hence, they use an out-of-order metric and not a reordering metric.

V. Measurement Study

This section analyzes the reordering characteristics in Finnish HSPA and LTE mobile networks. It operates under the premise that no reordering prevention algorithm is deployed, i.e. fast recovery is triggered after the third DUPACK. We use the methods described in Section III to distinguish packet reordering from packet loss. The cause for packet loss, which may be congestion on the path or wireless loss, is not important for our study as the methods distinguish packet reordering from both.

A. Measurement Methodology

We record measurements in different mobile networks in Finland using the crowd-sourced measurement platform Netradar [23]. Netradar employs a set of servers, and mobile clients connect to them to perform a series of measurements. The client is available for every big mobile phone operating system including Android, iOS and Windows Phone. It can be freely downloaded from app stores. The measurement is either initiated by the user or performed at certain intervals.

The measurement against the servers consists of 10 s TCP bulk transfer throughput tests for upload as well as download,

Fig. 7. Reorderings per connection [#] (log)

Table 1. Highest reordering rate [%] (95 %ile)

	Oper 1	Oper 2	Oper 3
HSPA	0.43	2.6	5.4
LTE	0.31	0.67	6.0

and sets of pings to determine the RTT. These are performed sequentially to not interfere with each other. Additionally, the mobile client gathers context information, such as mobile network operator, mobile technology, location, used basestation and signal strength in percentage [24], [13]. We note that all analyze mobile operator networks are connected to a single Internet Exchange Point (IXP) which is directly connected to the university network as well. Hence, all differences we see in measurements are due to the mobile operator networks. Additionally, there is very little possibility for reordering in the fixed network between operator networks and our servers. Still, we note, that we are not able to distinguish the exact place of reordering to happen.

In this paper we will use the download throughput test to measure packet reordering. For each connection performing the test we save a PCAP packet capture [25] on the measurement server for offline processing. These captures are then analyzed for reordering with the methodology presented in Section III. In total, we obtained over 110 000 measurements from the three main HSPA networks in Finland, and about 60 000 measurements from their LTE networks during all of 2015.

Due to the discussion on detecting reordering in Section III, we require connections to use the TCP SACK as well as TCP Timestamp option.

B. Measurement Results

The first metric is the number of connections affected by reordering. Figure 7 illustrates that reordering depends on the operator as well as network technology (HSPA vs. LTE). In general, more connections on LTE networks are affected by reordering compared to HSPA. In the HSPA network of operator 1, 48 % of connections experience one or more reordering events, while 82 % connections in the LTE network of the same operator experience reordering. The lowest number of connections with reordering events are found in the network of operator 2 with only 1 % respectively 8 %.

While analyzing the relationship of the number of reorderings to the sending rate, we find that there is a close to linear correlation in all of the networks. Hence, it is visible that the amount of reordering depends on the sending rate. We plot the

Fig. 8. Reordering rate [%]

Fig. 9. Reordering extents [byte] (log)

distribution of the reordering rate, i.e. percentage of packets re-ordered during a connection, in Figure 8. This is in line with the Reordered Packet Ratio defined in RFC 4737 [15]. We plot only those connections that contain reordering. The bin size is 0.02 %. The figure shows that in most connections containing reordering only a very small fraction of the total packets is re-ordered, as all of the operators have the biggest part in the bin containing rates of 0 % to 0.02 %. Table 1 shows the 95 %iles for each network. Here we see, that the range of reordering rates is very different between operators, ranging as high as 6 % for the LTE network of operator 3.

Additionally, we analyze how the reorderings are spread throughout the connection by counting the number of reorderings in a certain interval. In general we find that increasing the interval also increases the number of reorderings. This means that reordering is generally evenly distributed throughout the connection. However, in the LTE network of operator 1 we find clusters of reorderings in the order of tens of ms, which happen only infrequently, i.e. every few seconds on average.

As described in Section II, only events with a reordering extent larger than DUPTHRESH cause a spurious retransmission. Figure 9 shows the amount of reordering events that have an extent of more than three full sized segments. In HSPA networks with fewer reordering events per connection, the chance that the extents of these events are exceeding three segments is higher. With operator 1, which has the highest number of reordering events per connection, only about 20 % have an extent larger than three. Operator 2, which has the lowest number of connections with reordering, has over 70 % of reordering events with extents larger than three. In LTE networks, this effect is even more visible. In the network of operator 1, only few reordering

Fig. 10. Number of spurious recoveries per connection [#] (For readability graph Y-axis starts at 0.7)

Fig. 11. Reordering delay [ms]

extents are large, while for operator 2, over 90 % of reordering extents exceed three segments.

Not all of these reordering events reduce the TCP performance unnecessarily. On the one hand, a reordering event can be co-located with loss, i.e. during a recovery phase there can be both reordered and lost packets. In this case, halving the CWND is still mandatory due to the loss, and the impact of reordering is small. On the other hand, reordering events can occur as a burst as we saw above. This results in only one spurious recovery per burst. Figure 10 shows how many connections are affected by spurious recoveries caused by reordering. For every fast recovery we check the cause for each retransmission. A recovery is spurious if *all* the retransmissions are caused by reordering. The figure shows that up to 10 % of all HSPA and up to 23 % LTE connections suffer from spurious loss recoveries due to reordering. The higher number in LTE networks is especially dire as firstly, the use of this technology is growing and secondly, it offers higher data rates, but recovering from the needless CWND halving takes longer than for lower data rates.

Another metric of interest is the reordering delay, i.e. the time a packet is delayed due to reordering. Figure 11 shows that the reordering delay rarely exceeds 600 ms. It is visible that for most networks the graph is plateauing at around 30 ms and increases again at around 50 ms. With our current setup, we are not able to conclude reasons for this, as this would require network internal probing. This is seen as future work.

Still, as seen earlier, the reordering extent can be very high even if the reordering delay is very limited. The analysis of this behavior reveals that the magnitude of the reordering extent also depends on the current sending rate and not only the reorder-

Fig. 12. Heatmap reordering extent over sending rate

ing rate. As Figure 12 shows, this is indeed the case in mobile networks. We plot the reordering extent over the throughput average as a heatmap, the average being calculated over a time-frame of 250 ms [1] at the time of the reordering event. The heatmap shows a spot darker the more samples it has. In the left part of the graph we see that the reordering extent is quite low compared to the right part. For example, with a sending rate of 10 Mbit/s the majority of reordering extents are below 100 kB while at 20 Mbit/s it is already 200 kB, and with 70 Mbit/s it is around 600 kB. The drawn line indicates this behavior. What we see is that above this line there are hardly any samples.

While the left part, where the data points are mostly from measurements in HSPA, shows quite a clear upper bound, the right part of Figure 12 with a sending rate of 60 Mbit/s to 80 Mbit/s results from measurements in LTE networks. Here we see a huge spread of samples. Analyzing this more deeply, we find that a lot of reordering happens in bursts, where a chunk of packets arrive late. When this happens, all the reordered packets are transmitted shortly after one another, leaving little to no room for in-order data. The first reordered packet then behaves much like we expect from the current sending rate. But for all following packets the reordering extent gets smaller and smaller, which explains the large part of smaller extents in the figure.

The following example illustrates theoretically why there are higher reordering extents with higher sending rate: Assume a path with a reordering delay of 20 ms, and a bottleneck bandwidth of 3 Mbit/s. Because reordered segments arrive 20 ms later, the receiver can receive at most 7500 byte out-of-order before a reordered packet. Hence, with a sender maximum segment size (SMSS) of 1460 byte, the largest possible reordering extent is 5 segments. If the bottleneck bandwidth changes to 4 Mbit/s, the maximum reordering extent will increase to 7 segments, although the reordering delay remains constant. Generally, when a segment is reordered and therefore late, other packets surpass

it. The number of packets depends on the sending rate.

In conclusion, we see that reordering is very much present in mobile networks, depending heavily on the operator's network. We find that there is a surprisingly large number of connections containing reordering. Additionally, the reordering extents can be excessively large, which can cause problems for TCP. These large extents result from the increasing capacity in modern networks and not from very high reordering delays. Hence, in future networks these extents are likely increasing. In general, we found that both the number of reorderings during a connection and the reordering extent are dependent on the sending rate. We have found no evidence that the geographic location throughout Finland has an impact on the result.

VI. Reordering Prevention

The above measurement study gives us information on how an algorithm should be designed to prevent the negative impact of reordering, especially when taking into account the impact of the sending rate. In this section, we will describe the design and show that the principle of such a reordering prevention algorithm is working as intended.

A. Algorithm

While there are algorithms that require network support [26], the general principle of reordering prevention algorithms is to increase the DUPTHRESH, resulting in delaying retransmissions until it can be determined if the packet was reordered or not. If the DUPTHRESH is big enough, a situation as presented in Figure 3 appears. Normally, the DUPTHRESH is set based on the measured reordering extent during a connection, so that the DUPTHRESH is one packet more than the maximum reordering extent in packets. There are several algorithms implementing it this way [27], [28], [29], [30] and especially Linux. But this approach also has drawbacks [28]: waiting too long may result in an RTO, a significant increase in end-to-end delay for lost segments, and a delayed response to congestion. When we look at the situation of the reordering extent after the throughput changes, as it does frequently in mobile networks [1], the DUPTHRESH is not up-to-date any more, according to our measurement study in Section V-B: with a decrease in throughput the reordering extent decreases as well. However, the DUPTHRESH of these algorithms stays constantly high as measured before, which increases the possibility for all three drawbacks.

Others [31], [32] and especially TCP-NCR [33] take a different approach: they delay retransmissions for a specific time without reacting to reordering measurements. TCP-NCR always sets the DUPTHRESH to the equivalent of one RTT, finding this the maximum possible time a retransmission can safely be delayed. This has the benefit of reacting to the current sending rate. But it does so at the cost of delaying every retransmission, even when no reordering is present on the path at all.

Our algorithm TCP-aNCR modifies TCP-NCR, so that the general principle of adaptiveness to throughput remains, but the difference is that it sets the delay for retransmissions based on the perceived reordering in the network as well. This small but significant change results in a suitable reordering reaction for mobile networks and other fast changing environments.

For every newly measured reordering extent we calculate the fraction of the reordering extent to the current sending rate, which we call relative reordering extent. Hence, this fraction is constant if the reordering extent changes due to a changed sending rate. Whenever the DUPTHRESH is calculated, this procedure is done in reverse: the highest seen relative reordering extent is multiplied with the *current* sending rate, resulting in an appropriate measure under the current conditions.

The crucial part of the algorithm is the estimate of the current sending rate. Currently, we use the TCP parameter *Pipe* [10], which gives a measure for the bandwidth-delay product (BDP). Therefore, it scales with the sending rate. It has the additional benefit, that it also works with rate limited connections.

The full algorithm details are outlined online [34].

B. Setup

We implement the detection from Section III and the above algorithm in Linux kernel 3.19 [35]. We test TCP-aNCR, TCP-NCR and Linux in a network of XEN virtual machines, which are connected via a bridge. The network is emulated with Hierarchical Token Bucket (HTB) [36] for the bottleneck link speed and netem queue [37] to set queue length, RTT and reordering on a middle node between sender and receiver. All nodes run Debian Linux with kernel 3.19. To avoid situations where the measured delay is not determined by the reordering-tolerant TCP extensions themselves but more by the various buffers in the network we limit each network interface card (NIC) queue to 20 packets and limit the FIFO queue of the bottleneck node to the BDP. Last, we enhance netem's method of specifying reordering to directly set the desired fraction of packets that should be reordered for later delivery as well as their desired delay.

The TCP stacks on the senders and the receiver are configured via the `sysctl` interface to disable buffer auto-tuning and set the send and receive buffers to 30 MB. In some cases the Linux auto-tuning algorithm underestimates the send buffer when facing reordering, which leads to sender limited behavior during recovery. To avoid such possibilities, we set the buffers big enough for any case. We also disable the saving of path metrics, to avoid their re-use for repeated connection instances.

In addition we note that the Protection Against Wrapped Sequences (PAWS) [6] implementation in Linux has a window of one. This means that packets that are late by one tick of the Timestamp option are discarded. Since this is likely the case if reordering occurs, we disable this procedure in the receiver. We perform our test in a controlled environment and can assure that packets would not trigger PAWS in the intended way.

In the following, we analyze the behavior of a unidirectional bulk TCP data transfers with a duration of 60 s generated by *flowgrind* [38]. If not otherwise specified, the network has an RTT of 40 ms and the default reordering rate is 2 % with a reordering delay of 20 ms. From our measurement study in Section V-B we see that these are possible values also in mobile networks. While we concur that this setup is still synthetic and does not simulate real mobile environments, it enables us to analyze the behavior of the algorithm with packet reordering in situations where the sending rate is changing.

C. Analysis

First, we analyze the behavior of TCP-aNCR compared to TCP-NCR with a static bottleneck throughput in order to compare the retransmission delay. We measure both over a network path with a bottleneck link speed of 20 Mbit/s. Figure 13 shows the DUPTHRESH and CWND progression of TCP-NCR and TCP-aNCR. The comparison of the DUPTHRESH progression in Figure 13a shows that for TCP-NCR the DUPTHRESH periodically has large spikes compared to TCP-aNCR. Both algorithms adapt the DUPTHRESH according to the current amount of outstanding data. However, in the case of a packet loss TCP-NCR's DUPTHRESH increases to the value of one RTT, while TCP-aNCR sets it to the largest detected reordering event. This results in much lower delays for retransmissions when using TCP-aNCR, while still avoiding spurious recoveries.

Figure 14 shows the behavior of Linux and TCP-aNCR when the capacity of the path increases over time. The graph starts with a capacity of 10 Mbit/s for the first 5 s, then it is increased by 20 Mbit/s every 10 s. The reordering rate is set to 10 %. Figure 14a shows that Linux's DUPTHRESH jumps to a higher value and then stays constant for some time, while TCP-aNCR increases more smoothly overall. This originates from the two different ways the DUPTHRESH is calculated. Linux only increases the DUPTHRESH when a higher reordering extent is measured. These are exactly the points in the figure where the DUPTHRESH increases. In most cases the increase is accompanied by a spurious retransmission (indicated by little crosses at the bottom of the figure).

When looking at TCP-aNCR's behavior, we do not see any spurious retransmissions when the sending rate increases. The DUPTHRESH automatically increases throughout the connection, since the sending rate is taken into account. Additionally, it is on par with the Linux DUPTHRESH in most parts of the connection, although no new sample is needed.

Only towards the end the two algorithms diverge significantly. The explanation is found when looking at the CWND plot: at 35 s the CWND of the TCP-aNCR connection increases sharply and flattens out towards the end. The Linux connection CWND stays below that. We see a similar start of the fast increase between 40 s and 43 s but then it stops. Figure 14a shows that between 45 s and 50 s Linux spuriously retransmits very often because the DUPTHRESH is limited at 300. Linux sets an upper limit to avoid the negative impacts of a too high DUPTHRESH as discussed previously. In this case it is not big enough, and the growth of the CWND is heavily reduced due to the frequent spurious fast recoveries. TCP-aNCR has no such problems due to its adaptive algorithm. Additionally, we note that the lower CWND of TCP-aNCR compared to Linux in Figure 14b is a result of the way the congestion control works, in which the reordering reaction algorithm takes no part in.

In Figure 15 we show the behavior the other way around; now the capacity for a connection is halved. The link capacity of 20 Mbit/s is initially occupied by the connection alone, Linux or TCP-aNCR respectively. After 5 s a background flow joins and claims half the capacity for itself. In Figure 15b we see that in the first five seconds the CWND for both connections rises until around 140 segments. Then, when the background traffic starts for each of the connections, the CWND drops and remains

Fig. 13. Behavior of TCP-NCR and TCP-aNCR in static network conditions. (The legend in 13b applies to both figures.)

Fig. 14. Capacity increases every 10 s. Linux spuriously retransmits throughout the connection.

Fig. 15. Another flow joins after 5 s. TCP-aNCR reduces DUPTHRESH, and with it delay spikes.

around 80. In Figure 15a, the Linux DUPTHRESH increases to 71 segments in the first phase, and after that it remains constant as is to be expected. The TCP-aNCR DUPTHRESH on the other hand increases with the CWND until 5 s as we have seen in previous graphs, but it then adapts to the new conditions alongside the CWND and is reduced to around 40 segments. We can see that no spurious retransmissions occur, which means that the DUPTHRESH is still high enough to prevent them. This also signifies that the Linux DUPTHRESH is much too high and delays necessary retransmissions unnecessarily.

But the question is how much delay is actually saved by exploiting the dependence on the sending rate. Our measurement study in Section V-B shows that this characteristic is prevalent in mobile networks, in which a sudden change in sending rate is also common [1]. We run the same setup as in the graph before, one flow starts and after 5 s another flow joins the bottleneck, with different bottleneck link capacities ranging from 10 Mbit/s

to 50 Mbit/s. After the capacity is shared among the two flows, we collect the current RTT estimate and the time the retransmission was additionally delayed by the reordering algorithm from the Linux kernel. We show averages of all retransmissions that occurred between 20 s and 60 s (to give the system time to reach equilibrium) over 10 iterations each. Figure 16 shows the average increase in delay spike normalized to the RTT. The optimal value would be 0.5 RTTs, since this is the reordering delay, which is denoted as the lowest line in the graph. Of the three different algorithms, TCP-aNCR behaves best and has the lowest delay spikes around 0.7 RTTs, followed by Linux with 0.9 and TCP-NCR with around 1.1 RTTs. TCP-NCR always waits one full additional RTT, thus having the longest delay. Linux adapts its DUPTHRESH to the reordering conditions, but it fails to adapt to the changing sending rate. TCP-aNCR adapts to both measured reordering and the different sending rate, and can therefore achieve the lowest delay spikes.

Fig. 16. Average increase in delay spike due to reordering prevention (normalized to the current RTT).

VII. Summary

In this paper we introduce a complete reordering methodology based on the sender side of TCP and point out that we need the TCP Timestamp option as well as the DSACK option to accurately measure the reordering extent.

Then, we show that reordering is an issue in modern networks, and that the reordering depends on the used mobile network technology. This has to be taken into account, when designing transport layer algorithms. We show that the number of reorderings as well as the possible reordering extent is increasing in proportion to the sending rate of the connection. Hence, we find that more connections in LTE networks are affected by reordering than in the slower HSPA networks.

Last, we introduce a reordering algorithm TCP-aNCR which takes the additional insight into account by adapting to the current sending rate. We show that the algorithm works as intended in a controlled environment, and that it reduces the delay for retransmissions. A much deeper analysis, especially in real life is future work. One possibility is to deploy the algorithm to the Netradar measurement server.

Acknowledgment and Disclaimer

Lars Eggert has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

REFERENCES

- [1] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance," in *Proc. ACM SIGCOMM*, 2013.
- [2] J. Postel, "Transmission Control Protocol," RFC 793.
- [3] E. Blanton and M. Allman, "On Making TCP More Robust to Packet Reordering," *SIGCOMM CCR*, vol. 32, no. 1, 2002.
- [4] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018.
- [5] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," RFC 2883.
- [6] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "TCP Extensions for High Performance," RFC 7323.
- [7] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681.
- [8] J. C. R. Bennett, C. Partridge, and N. Shectman, "Packet Reordering is Not Pathological Network Behavior," *IEEE/ACM ToN*, vol. 7, no. 6, 1999.

- [9] K.-C. Leung, V. O. Li, and D. Yang, "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges," *IEEE TPDS*, vol. 18, no. 4, 2007.
- [10] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP," RFC 6675.
- [11] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit," RFC 3042.
- [12] A. Zimmermann, L. Schulte, C. Wolff, and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP," Internet-Draft, 5 2015.
- [13] L. Schulte, G. R. Moktan, M. Särelä, and J. M. J. Manner, "Towards Understanding Interruption and Reordering in HSPA/HSPA+ Networks," in *Proc. IEEE STWiMob*, 2014.
- [14] R. Ludwig and M. Meyer, "The Eifel Detection Algorithm for TCP," RFC 3522.
- [15] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser, "Packet Reordering Metrics," RFC 4737.
- [16] E. Blanton and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions," RFC 3708.
- [17] E. Dahlman, S. Parkvall, J. Skold, and P. Beming, *3G Evolution; HSPA and LTE for Mobile Broadband*. Academic Press, 2007.
- [18] V. Paxson, "End-to-end Internet Packet Dynamics," *IEEE/ACM ToN*, vol. 7, no. 3, 1999.
- [19] J. Bellardo and S. Savage, "Measuring Packet Reordering," in *Proc. ACM IMW*, 2002.
- [20] L. Gharai, C. Perkins, and T. Lehman, "Packet Reordering, High Speed Networks and Transport Protocol Performance," in *Proc. IEEE ICCCN*, 2004.
- [21] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Measurement and Classification of Out-of-Sequence Packets in a Tier-1 IP Backbone," *IEEE/ACM ToN*, vol. 15, no. 1, 2007.
- [22] J. Feng, Z. Ouyang, L. Xu, and B. Ramamurthy, "Packet reordering in high-speed networks and its impact on high-speed TCP variants," *Computer Communications*, vol. 32, no. 1, 2009.
- [23] "Netradar Measurement Platform," Aalto University, 2013. [Online]. Available: <http://www.netradar.org/>
- [24] S. Sonntag, L. Schulte, and J. Manner, "Mobile network measurements - It's not all about signal strength," in *Proc. IEEE WCNC*, 2013.
- [25] Tcpdump Developers. tcpdump/libpcap. [Online]. Available: <http://www.tcpdump.org>
- [26] A. Sathiaselalan and T. Radzik, "Reorder notifying TCP (RN-TCP) with explicit packet drop notification," *International Journal of Communication Systems*, vol. 19, no. 6, 2006.
- [27] E. Blanton, R. Dimond, and M. Allman, "Practices for TCP Senders in the Face of Segment Reordering," Internet-Draft, 02 2003.
- [28] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A Reordering-Robust TCP with DSACK," in *Proc. IEEE ICNP*, 2003.
- [29] K.-C. Leung and C. Ma, "Enhancing TCP Performance to Persistent Packet Reordering," *KICS Journal of Communications and Networks*, vol. 7, no. 3, 2005.
- [30] S. Landström, H. Ekström, L. Larzon, and R. Ludwig, "TCP-Aix: making TCP robust to reordering and delay variations," Research Report in Luleå University of Technology, 2006.
- [31] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka, "A New TCP for Persistent Packet Reordering," *IEEE/ACM ToN*, vol. 2, no. 14, 2006.
- [32] S. Bhandarkar, N. E. Sadry, A. L. N. Reddy, and N. H. Vaidya, "TCP-DCR: A Novel Protocol for Tolerating Wireless Channel Errors," *IEEE TMC*, vol. 4, no. 5, 2005.
- [33] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton, "Improving the Robustness of TCP to Non-Congestion Events," RFC 4653.
- [34] A. Zimmermann, L. Schulte, C. Wolff, and A. Hannemann, "Making TCP Adaptively Robust to Non-Congestion Events," Internet-Draft, 5 2015.
- [35] L. Torvalds. (2015) The Linux Kernel. [Online]. Available: <http://www.kernel.org/>
- [36] M. Devera. (2016) Hierarchy Token Bucket. [Online]. Available: <http://lartc.org/manpages/tc-htb.html>
- [37] Linux foundation. (2014) netem. [Online]. Available: <http://www.linuxfoundation.org/collaborate/workgroups/networking/netem>
- [38] Flowgrind Developers. (2014) Flowgrind - TCP traffic generator. [Online]. Available: <http://www.flowgrind.net/>