RACK for SCTP

Felix Weinrank, Michael Tüxen
Münster University of Applied Sciences
Department of Electrical Engineering and Computer Science
Stegerwaldstraße 39
48565 Steinfurt, Germany
{weinrank,tuexen}@fh-muenster.de

Erwin P. Rathgeb
University of Duisburg-Essen
Faculty of Business Administration and Economics
Computer Networking Technology Group
Schützenbahn 70
45127 Essen, Germany
erwin.rathgeb@uni-due.de

***Abstract*—RACK for TCP is a loss detection and recovery algorithm which relies on the notion of time instead of counting packet losses. The algorithm has been integrated in Microsoft Windows, Linux and FreeBSD as a full-featured alternative to the existing TCP loss recovery algorithms. *Cheng et al.* claim that the algorithm offers a significant improvement, especially for tail losses and cases with packet reordering within the network. We adopted RACK for the SCTP protocol and improved its mechanisms by utilizing built-in SCTP specific features. To evaluate the benefits and drawbacks of RACK and our modifications to the algorithm, we integrated RACK in the SCTP model of the OMNeT++/INET simulation environment. Our simulations show a positive impact in common networking scenarios regarding loss recovery and reordering tolerance.**

I. Introduction

The majority of the widely deployed loss detection algorithms for reliable transport protocols use a combination of counting packet losses and timer based mechanisms. For example, the specifications of the Stream Control Transmission Protocol (SCTP) [1] and the Transmission Control Protocol (TCP) [2] distinguish between timer based and fast retransmissions. In 2015, *Cheng et al.* (from Google) introduced a new loss detection and recovery approach: "Recent ACKnowledgment" (RACK), published as an IETF draft [3]. Instead of triggering retransmissions by counting gaps in acknowledged sequence number spaces or timeouts, RACK uses the notion of time to detect lost packets. RACK promises a faster loss detection and less spurious retransmissions caused by network reordering. It is intended as a full replacement for existing recovery algorithms. Although RACK has been tailored for the TCP protocol, its methods can also be applied for other transport protocols such as Quick UDP Internet Connections (QUIC) or SCTP. In a first step, we adopted the RACK algorithm to the SCTP protocol and evaluated its benefit in certain scenarios using a discrete event simulation framework. Based on those results, we improved the algorithm by utilizing built-in SCTP specific features.

II. RACK for TCP

The RACK algorithm is meant as a fully featured replacement for existing TCP loss recovery algorithms. It has already been integrated into the Linux kernel, Microsoft Windows and FreeBSD. Since the RACK algorithm requires a selective acknowledgement of packets, the TCP SACK extension [4] is a mandatory requirement for the usage with TCP. Beneficial for its deployment is that the RACK algorithm requires no changes or additional extensions, apart from SACK at the receiver side.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

A. Loss Detection

For every outgoing packet, the sender records the transmission time in the packet's local meta information storage. Upon the arrival of a selective acknowledgement (SACK) from the receiver, the sender picks the most recently delivered (sent and acknowledged) packet. All outstanding packets, which have been sent significantly before the most recently delivered packet, are assumed lost and queued for retransmission. If packets are outstanding but not overdue, RACK arms a timer to detect lost packets without waiting for a subsequent acknowledgement. The RACK draft [3] suggests to calculate the timeout by adding the round trip time of the most recently delivered packet (RackRTT) and the network reordering window (ReoWND). The ReoWND is calculated dynamically in order to compensate for delays caused by network reordering. An example of the RACK operation is shown in Figure 1. In this example, the second packet (DATA #2) is lost and the receiver reports it as missing upon receiving the third packet. When the sender receives the first SACK, acknowledging packet (DATA #1), no gap is reported. The acknowledgement for the third packet (DATA #3) triggers the retransmission of the lost second packet (DATA #2) since a subsequently sent packet has been acknowledged and the ReoWND timeout has been passed.

B. Reordering Window

The cause for gaps in the received sequence number space is not limited to loss. If packets get reordered within the network, the receiver will also notice a gap. To avoid spurious retransmissions caused by packet reordering, traditional loss recovery algorithms require multiple gap reports before a retransmission gets triggered. RACK makes another approach by calculating a dynamic reordering window, based on the detected network characteristics. The sender detects the occurrence of network reordering by recognizing original data packets being delivered out of order in the sequence number space. This is done by recording the highest sequence number which got selectively

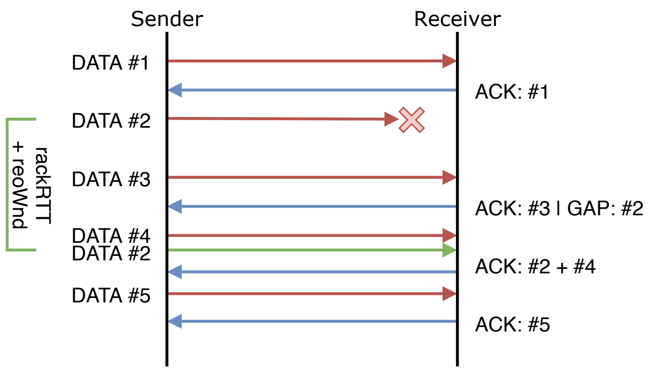

Fig. 1. Retransmission triggered by a RACK timer.

or cumulatively acknowledged. If an subsequent SACK selectively or cumulatively acknowledges an unacknowledged and also never retransmitted sequence number below the number recorded previously, the corresponding packet has been reordered. This mechanism only works for packets which have not been retransmitted. *Cheng et al.* suggest an initial reordering window of a quarter RTT. A reordering window that is too small leads to spurious retransmissions, detectable by the Duplicate Selective Acknowledgement (DSACKs) extension. If the sender receives a DSACK notification, the reordering window can be assumed as being too small. Thus the sender increases the reordering window by a quarter RTT for every round trip in which duplicates are reported. Once no more reordering is detected, the inflated reordering window should be kept for a period of 16 loss recoveries before being reset to its initial value of a quarter RTT. If the sender has not detected any reordering for the particular connection, the reordering window is set to zero during the loss recovery operation. Lowering the reordering window to zero lets RACK behave even more aggressively than the regular recovery algorithm, specified in RFC6675 [5]. RACK's dupthresh approach considers all outstanding packets lost, which have been sent before an acknowledged packet, even if they have not been reported missing by three gap reports.

C. Tail Loss Probing

In addition to the passive loss detection algorithm, RACK introduces active tail loss probing (TLP). Tail loss occurs if either the last payload segment(s) or the last acknowledgements of a transmission are lost. Since no subsequent packets are transmitted, lost packets can not be detected by hitting the dupthresh limit or a RACK timeout. According to Google [6], tail loss is a common problem for traffic that follows a request/response style pattern. Google also reports that 70% of the losses on their google.com search engine are recovered by retransmission timeout (RTO). RACK's TLP mechanism addresses this issue by introducing a probing timeout (PTO) timer. Whenever new data gets transmitted or a SACK cumulatively acknowledges data, RACK schedules a probing timer. In case the PTO timer fires, RACK sends out a probing packet to trigger an acknowledgement from the receiver. The probing packet contains unsent data or, if not available, retransmits the most recently sent packet. The RACK draft [3] suggests to schedule the probing timer in relation to the smoothed round trip time (SRTT) and the amount of data in flight. In case a SRTT value is available, the probing timeout is two times the SRTT plus an additional delay. If only a single packet is outstanding, the additional delay is set to worst case delayed ack timer (WCDelAckT = 200ms), otherwise it is set to two milliseconds. The RACK draft [3] suggests to set the WCDelAckT value to 200 milliseconds, it represents a potential long delayed ACK timer at the receiver. In case a SRTT value is not available yet, the probing timer is set to one second.

III. RACK for SCTP

SCTP is a reliable, connection oriented transport protocol. Even though originally designed for the transmission of small signaling messages in the Signaling System No. 7 (SS7), SCTP has evolved into a universal transport protocol over the last years. Additionally to the usage in SS7, SCTP has become widely deployed as the underlying transport protocol for Data-Channels, which are a fundamental part of Web Real-Time Communication (WebRTC). Both SCTP peers transmit application data by using streams within the association, each stream gets identified by a unique 16 bit stream identifier. Payload data is transmitted using DATA chunks which belong to a particular stream and are identified by a unique Transmission Sequence Number (TSN). If a SCTP packet containing one or more DATA chunks arrives, the receiver acknowledges the related TSNs, cumulatively or selectively, by using a SACK chunk. While RACK for TCP requires additional extensions to be supported by both peers, SCTP already provides the underlying mechanisms for RACK in its basic RFC4960 [1] specification. We adopted RACK for the SCTP protocol and developed additional improvements by using built-in SCTP specific functionalities as well as by using optional extensions. In contrast to the RACK specification, our SCTP implementation maintains the meta information per DATA chunk and not per packet. Also, SCTP reports missing packets as well as duplicate packets by default without additional extensions.

IV. Performance Evaluation

We implemented RACK in the SCTP model of the INET framework [7] for the OMNeT++ simulation environment [8] to validate the adoption and our modifications to the RACK algorithms by comparing them to SCTP's default mechanisms. By using the discrete event simulation framework, we are able to precisely evaluate the internal mechanisms in a highly flexible and deterministic environment. Since a key part of RACK is the robustness against packet reordering, we extended the INET model by a mechanism to simulate a variety of reordering patterns within the network. We simulate packet reordering by delaying SCTP packets leaving the SCTP module. The delay is determined for every single packet, based on a configurable reordering pattern. At the IETF 102 conference, Google presented statistics showing that 9.4% of their server-to-client connections and 5.4% of their client-to-server connections are facing reordering [9]. Previous evaluations have also shown that reordering is a common issue in modern networks [10] [11] [12]. Our scenario represents a common client-to-server model in which the client transfers data to the server. To analyze the benefits and downsides of RACK and our modifications to the algorithm, we focused on the application-to-application delay, the amount of time until a missing packet is retransmitted, the ratio of spurious to valid retransmissions and the goodput. If loss occurs within the network, the application-to-application delay indicates how quickly lost packets are detected and recovered by the sender's loss recovery algorithm, since data is delivered to the application reliably and in order. The ratio of spurious retransmissions to valid retransmissions indicates how well the algorithm distinguishes between loss and reordering. A falsely detected loss has a negative impact on the congestion control and will lead to a decreased goodput rate. A good recovery algorithm retransmits lost packets very quickly, while keeping the ratio of spurious retransmissions as low as possible. From the perspective of an application, the application-to-application delay and the goodput are the most relevant indications. Especially request-response protocols like HTTP/1 and HTTP/2 suffer from head of line blocking (HOL) and gain a large benefit from a quick error recovery. Our simulation scenario consists of four hosts and two routers. The routers are connected via a bottleneck link, having a capacity of 10 Mbit/s and a RTT of 50 milliseconds. The hosts are connected to the routers via lossless one Gbit/s links without a delay. To get a reasonable amount of randomness, the UDP client sends a small amount of random background traffic to the UDP server, competing with the SCTP traffic on the bottleneck link. The background traffic leads to a small amount of jitter and, if congesting the bottleneck queue, to a small amount of packet loss. The UDP client sends packets having a payload between 100 and 1000 bytes every 25 - 100 milliseconds.

A. Signaling Traffic

We started our evaluation by simulating a signaling connection. The client sends 100,000 SCTP messages at a constant rate to the server, each message has a payload of 50 bytes. The bottleneck link is configured to drop one percent of all packets. We measured the time span between the initial transmission and the retransmission of every lost packet, hereinafter referred to as the recovery time. Each experiment has been repeated ten times. Figure 2 shows the relation between the recovery time and the send interval. A recovery time of 50 milliseconds is near the theoretical lower limit of a single RTT. The lower the sending rate becomes, the better the RACK algorithm performs in comparison to the dupthresh mechanism. At a sending interval of 50 milliseconds, the dupthresh algorithm takes more than twice as long to trigger a retransmission, compared to the RACK algorithm. Especially for signaling connections with lower sending rates, RACK provides a huge benefit. In order

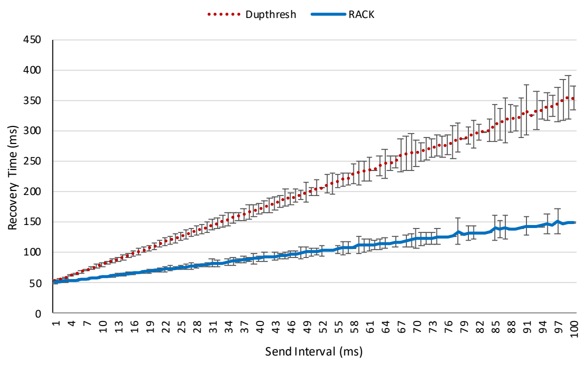

Fig. 2. Recovery time in relation to the sending rate at 1% packet loss.

to analyze RACK's robustness against packet reordering, we configured the bottleneck link as a lossless connection only facing packet reordering. We set the reordering probability to five percent. The degree of reordering is represented by an exponential distribution at ten milliseconds mean. This means that five percent of all packets are delayed by ten milliseconds mean. As shown in Figure 3, RACK provides a much better robustness against spurious retransmissions. Even for high degrees of reordering, RACK performs quite well while the dupthresh algorithm spuriously retransmits every second packet. At a send interval of 10 milliseconds and 100.000 transmitted packets, RACK spuriously retransmits 40 packets while the dupthresh algorithm retransmits 236. This ratio increases for lower send intervals where RACK keeps it at a low level while the dupthresh algorithm retransmits about 3,300 packets. Since RACK performs better than the

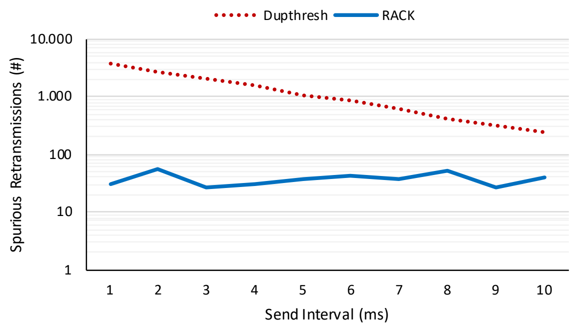

Fig. 3. Amount of spurious retransmissions in relation to the send interval.

dupthresh algorithm in both experiments, we evaluated a combination of loss and reordering. We have configured the bottleneck link at one percent packet loss and a probability for packet reordering of one percent, having a reordering degree of ten milliseconds mean. The reordering degree is still represented by an exponential distribution. As shown in Figure 4, a small degree of reordering makes no difference for the dupthresh algorithm. Since the connection faces a constant level of reordering, the RACK algorithm calculates a reordering window of a single RTT and does not use its dupthresh algorithm. This results in a recovery time of at least two RTTs. We can summarize that in case of a low sending

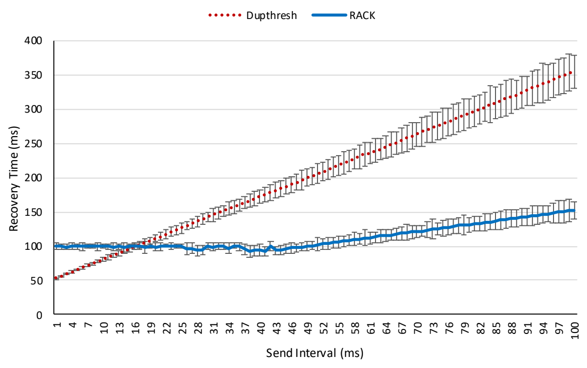

Fig. 4. Recovery time in relation to the send interval at one percent packet loss and reordering.

rate or a high degree of reordering, RACK shows a clear advantage over the dupthresh algorithm. If the sending rate exceeds twelve packets per RTT and the connection does not face any reordering, RACK shows no benefit since it also uses a three strikes dupthresh algorithm.

B. Saturated connection

Our second scenario analyzes a bulk data transfer from the client to the server, where the client sends full sized frames at wire speed for 60 seconds. In case the connection does not face any packet reordering, both algorithms utilize the link and do not trigger any spurious retransmissions. We increased the link's rate of packet loss and observed a reduced goodput for both algorithms, the results shown in Figure 5. The numbers match our expectations of a similar performance for both algorithms. Figure 6 shows the impact of packet

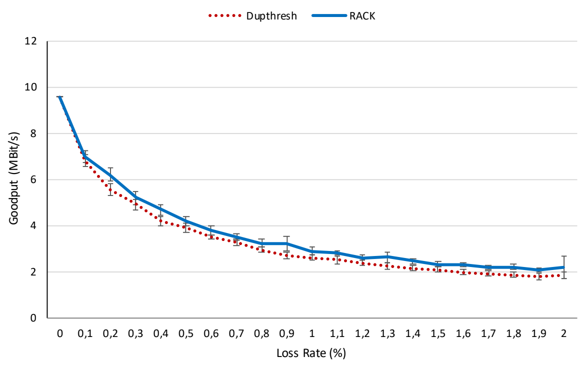

Fig. 5. Goodput in relation to packet loss for a bulk transfer.

reordering on the goodput for a bulk transfer. The connection does not face any loss but a fixed one percent probability of reordering. We increased the mean value of the distribution and measured the goodput. The graph shows the negative impact of spurious retransmissions to the goodput for bandwidth oriented transmissions. Due to RACK's reordering window mechanism, the goodput is up to two times higher compared to the dupthresh mechanism. Figure 7 shows the impact to the

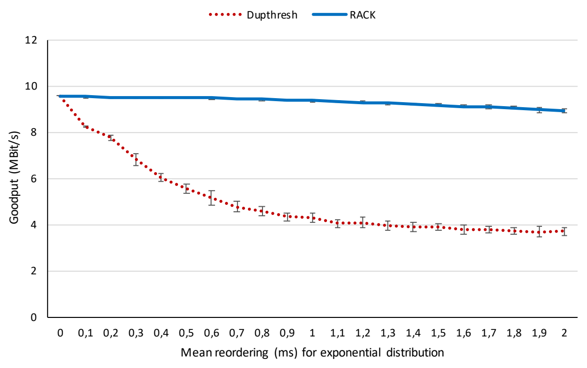

Fig. 6. Goodput in relation to packet reordering degree at 1% probability.

goodput by applying a combination of reordering and loss. We configured a static one percent probability of reordering. The degree of reordering is modeled using an exponential distribution with five milliseconds mean. We increased the loss rate and observed a reduced goodput for both algorithms. The measurements have shown a positive impact on signaling

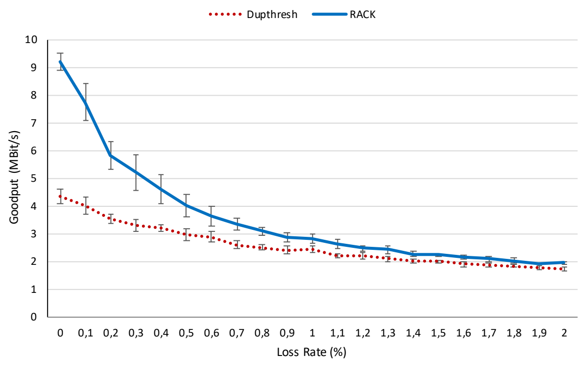

Fig. 7. Goodput in relation to packet loss and a static reordering degree.

traffic as well as on bulk data transfers for the SCTP protocol under certain conditions.

V. Modifications and Improvements

Our evaluations also revealed some flaws and requirements regarding the usage with SCTP. Therefore, we have made some modifications and improvements which provide a better performance in certain scenarios.

A. Burst Mitigation

The RACK algorithm tends to mark large blocks of payload as lost, which is scheduled for retransmission. This behavior is in particular caused by RACK's modified dupthresh algorithm and Tail Loss Probing (TLP). In case the tail of a transmission is lost, the sender triggers a loss probe as described previously. If the peer acknowledges the loss probe, RACK assumes all previously sent but unacknowledged packets as lost. Even if the loss event reduces the congestion window, the sender will push a large burst of packets into the network because no packets are assumed to be in flight anymore. Since bursts tend to cause additional packet loss by overflowing router queues, a burst mitigation mechanism is desireable. *Cheng et al.* suggest implementing Proportional Rate Reduction (PRR) in order to mitigate this issue. Since SCTP already includes a burst mitigation algorithm, we have not implemented PRR in favor of optimizing the builtin mechanism. First, we have evaluated the impact of TLP caused bursts without a burst mitigation technique. For this evaluation, we still used the same client/server scenario as described in the last section, having a ten Mbit/s bottleneck and a 50 milliseconds RTT. The bottleneck router is configured with a drop tail queue, providing a frame capacity of ten packets for the SCTP and UDP connection. The sender transmits 2,000 packets at a constant rate of one packet per millisecond, each packet carries 1,000 bytes of payload. The network drops a tail of 100 packets, affecting DATA chunk TSNs 1,901 - 2,000. Figure 8 illustrates this tail

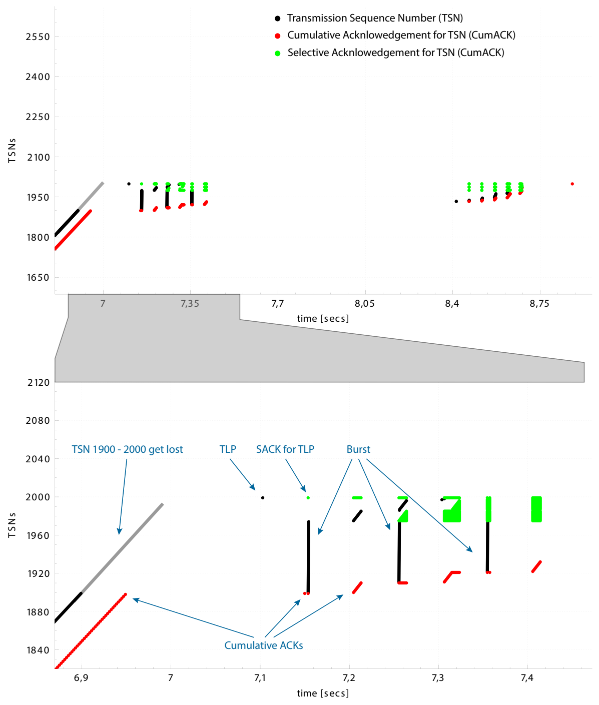

Fig. 8. Missing burst mitigation causes additional losses.

loss, magnified in the lower part. The loss begins at [t=6.9]. The lost packets are drawn in light grey. The TLP timer has been armed at [t=7.0], right after the last segment has been transmitted by the sender. Since the RTT is 50 milliseconds, the sender has not detected any reordering, and more than a single packet is in flight, the probing timer is armed to fire after 102 milliseconds (2 * SRTT + 2 milliseconds artificial delay). Since no new transmissions have been triggered and no SACKs arrived, the timer fires at [t=7.1] and sends TSN 2,000 as a tail loss probe. 50 milliseconds later, the sender receives a SACK from the receiver which acknowledges the TLP. The sender now marks all outstanding packets in flight as lost and schedules their retransmission. Since no burst mitigation is applied, this leads to a burst of more than 70 packets only limited by the congestion window. The burst is too large for the bottleneck router queue, resulting in a large number of dropped packets. As shown at [t=7.2], the first block of the burst gets cumulatively acknowledged, but most of the subsequent packets get lost. The RACK algorithm detects those losses which leads to additional losses of the same pattern, see [t=7,25] and [t=7.35]. In a next step, we applied SCTP's default burst limitation algorithm to the TLP mechanism. By default, the algorithm limits the size of each burst to four times the maximum transmission unit (MTU). As shown in Figure 9 at [t=7.15], the sender bursts only four packets to the network upon receiving the acknowledgement for the tail loss probe. Since the receiver still has gaps in its TSN sequence number space, every received packet gets acknowledged by a separate SACK. Each of these received SACKs trigger the sender to transmit four additional data packets, resulting in 16 subsequent payload packets, as shown at [t=7.2]. As shown in the magnified part, the 16 packets are not sent at once but at the arrival speed of the SACKs. This behavior repeats after another round trip at [t=7.25]. SCTP's default burst mitigation strategy still results in subsequent packet loss, albeit at a lower level. While SCTP's burst mitigation works well

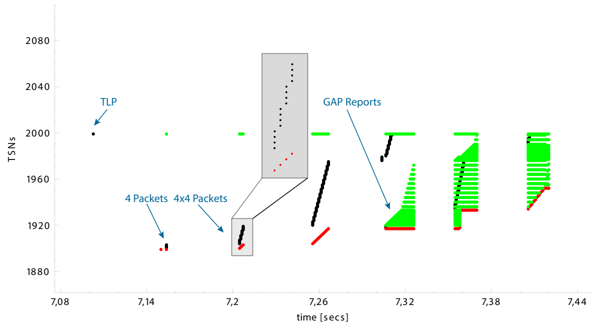

Fig. 9. Mitigated TLP burst due to SCTP burst mitigation, limit set to 4.

during normal operation, it behaves much more aggressive during recovery. In normal operation, i.e. no gaps have been detected, the receiver tends to acknowledge only every second packet. A burst size of four then results in four new packets for every two acknowledged packets. If the receiver detects a gap in the sequence number space, every received packet is acknowledged by a SACK chunk until the gap is filled. This results in four new packets for every acknowledged packet. We have evaluated multiple networking scenarios to optimize SCTP's burst mitigation during tail loss recovery. To illustrate the issue, we are using the same bottleneck scenario, having a tail loss of 100 packets. Considering full sized frames, this is in the magnitude of a bandwidth-delay product of a 100 Mbit/s connection having a 10 milliseconds RTT. Figure 10 shows the required amount of retransmissions in relation to the bottleneck queue capacity for multiple max burst values. Even for larger queueing capacities, SCTP's default burst limit of four packets is too high. Due to the additional packet loss, reducing the burst limit from four to three reduces the recovery time by more than a half in case of a shallow buffer. Considering these results, an optimal burst limit depends on the network conditions, which make a static value unsuitable. Therefore, we have developed an adaptive burst limit algorithm to overcome this issue. Initially, SCTP's default burst limit of four MTUs is applied and kept during the first RTT. For all subsequent transmissions during the tail loss recovery, the limit is set to two MTUs per burst. This behavior is similar to the slow start mechanism of congestion controls, having an initial window of four and doubling the amount of in-flight data every RTT. In case the sender detects subsequent packet loss during

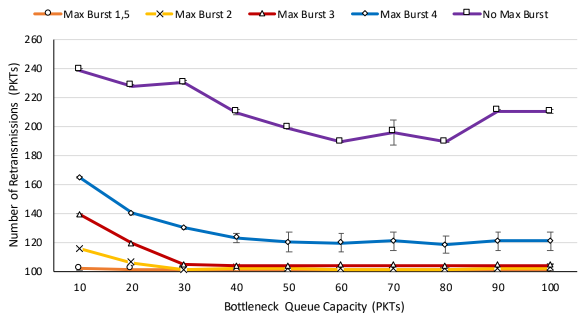

Fig. 10. Number of retransmissions to recover a tail loss of 100 packets in relation to bottleneck capacity and max burst limit.

the recovery operation, the burst limit is reduced. If the loss affects packets which have been retransmitted during the first recovery round trip, the initial burst size is assumed as too high and will be reduced by a single MTU until a minimum of two MTUs. If the lost packets have been transmitted during a subsequent round trip period, the increasing rate is too high and will be reduced by 0.25 until a minimum of 1.25. As soon as the sender has performed 16 tail loss recovery operations without additional packet loss, the burst limit is reset to its initial values [4/2]. This mechanism lets the sender dynamically react to changing network conditions, especially shallow bottleneck buffers.

B. Active RTT Measurement

Since RACK heavily relies on RTT values, precise and actual RTT estimations are indispensable. The first RTT estimation is available right after finishing the four-way SCTP handshake, but this estimation may not be very precise. Especially signaling connections following a request/response style traffic pattern may face long idle periods without transmissions. If the network conditions change in the meantime, the RTT measurements can become invalid. If a connection is idle, SCTP sends heartbeats periodically to monitor the reachability of the peer. These heartbeats do not only ensure a vital connection, they also provide a handy RTT measurement mechanism. Additionally to the periodic heartbeats, an SCTP peer can actively trigger RTT measurements by sending heartbeat chunks. The receiver will instantly reflect the heartbeat chunk, allowing an RTT measurement without transmitting any payload. By default, an SCTP peer sends a heartbeat per path every 30 seconds of being idle.

VI. Conclusion and Outlook

We have successfully adopted Google's alternative loss recovery mechanism for the SCTP protocol and improved its mechanisms by utilizing the SCTP specific feature set. To verify our adaptation and to evaluate our SCTP specific improvements to the RACK algorithm, we implemented the algorithm in the SCTP model of the OMNeT++/INET simulation framework. The RACK algorithm shows a significant improvement over the dupthresh algorithm, especially for non-saturated connections and in the event of tail loss. The benefits are not limited to loss. RACK is also much more resilient against packet reordering in the network. By utilizing SCTP specific features, we were able to further improve the RACK algorithm in common cases. Our future work will focus on implementing RACK in the FreeBSD kernel stack and the widely used user space SCTP library. Additionally, we are considering if and how our improvements can be applied for other protocols such as TCP.

References

[1] R. Stewart (Ed.), "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–152, Sep. 2007, updated by RFCs 6096, 6335, 7053. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4960.txt

[2] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681 (Draft Standard), RFC Editor, Fremont, CA, USA, pp. 1–18, Sep. 2009. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5681.txt

[3] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "Rack: a time-based fast loss detection algorithm for tcp," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tcpm-rack-05, April 2019. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-tcpm-rack-05.txt

[4] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–12, Oct. 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2018.txt

[5] 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 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–15, Aug. 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6675.txt

[6] "RACK: a time-based fast loss recovery Draft-ietf-tcpm-rack-03 updates," 2018. [Online]. Available: https://datatracker.ietf.org/meeting/101/materials/slides-101-tcpm-update-on-tcp-rack-00

[7] "INET Framework for OMNEST/OMNeT++," 2018. [Online]. Available: https://github.com/inet-framework/inet

[8] "OMNeT++ Simulation Environment," 2018. [Online]. Available: https://www.omnetpp.org/

[9] I. Swett, "Ietf maprg - udp packet reordering," September 2019. [Online]. Available: https://datatracker.ietf.org/meeting/102/materials/slides-102-maprg-udp-packet-reordering-ian-swett-00

[10] J. Bellardo and S. Savage, "Measuring packet reordering," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, ser. IMW '02. New York, NY, USA: ACM, 2002, pp. 97–105. [Online]. Available: http://doi.acm.org/10.1145/637201.637216

[11] J. C. R. Bennett, C. Partridge, and N. Shectman, "Packet reordering is not pathological network behavior," *IEEE/ACM Trans. Netw.*, vol. 7, no. 6, pp. 789–798, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1109/90.811445

[12] V. Paxson, "End-to-end internet packet dynamics," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 139–152, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/263109.263155