A QUIC Simulation Model for INET and its Application to the Acknowledgment Ratio Issue

Timo Völker*, Ekaterina Volodina†, Michael Tüxen* and Erwin P. Rathgeb†
*Department of Electrical Engineering and Computer Science
FH Münster University of Applied Sciences, Steinfurt, Germany
Email: {timo.voelker,tuexen}@fh-muenster.de
†Computer Networking Technology Group
University of Duisburg-Essen, Essen, Germany
Email: {ekaterina.volodina,erwin.rathgeb}@uni-due.de

***Abstract*—Quick UDP Internet Connections (QUIC) is a novel transport protocol introducing known features in a new protocol design. To investigate these features and the design, we developed a QUIC implementation in the INET simulation model suite.**

In this paper, we describe that implementation, its validation and a result achieved using the simulation model. The result shows the negative impact on throughput, when raising the acknowledgment ratio. We propose a solution and describe how it solves the issue.

***Index Terms*—QUIC, INET, OMNeT++, Protocol Simulation, Transport Protocol, Ack Ratio**

I. Introduction

Quick UDP Internet Connections (QUIC) is a novel transport protocol currently being standardized by the Internet Engineering Task Force (IETF). Multiple implementations exist that support the current version of the internet drafts.

In order to investigate QUIC's features and new protocol design in a controllable environment, we developed a QUIC implementation in the INET simulation model suite. The simulation model offers an agile way to evaluate the performance of the protocol and its potential optimizations. For instance, we used the simulation model to analyze QUIC connections with different ratios of received packets to sent acknowledgements (ack ratio). The result shows a negative impact on throughput, when raising the ack ratio. We propose an extension to QUIC that solves the issue, as the result further shows.

This paper starts by introducing the QUIC protocol in section II. Section III describes our QUIC implementation, followed by the description of its validation in section IV. In section V, we describe the result of the ack ratio analysis.

II. The QUIC Protocol

QUIC is a protocol invented by Google. Google presented its work in November 2013 at the IETF, which initiated the foundation of the IETF QUIC working group. The QUIC working group described the core protocol in the two internet drafts [1] and [2].

QUIC operates on top of the User Datagram Protocol (UDP) but adds features that are usually located in the transport layer, like in-order delivery, reliability, packetization, congestion control, flow control and multi-streaming. A QUIC packet consists of a QUIC packet header and one or more QUIC frames. Each frame has its own header format. QUIC always encrypts the payload and most of the header fields.

The QUIC packet header contains a packet number identifying the packet. The packet number is strictly monotonic increasing for each outgoing packet. QUIC acknowledges the receipt of a packet by sending a packet containing an ACK frame, which lists the packet numbers of received packets.

The STREAM frame is used to transfer data from the application for a specific stream. QUIC does not consider message boundaries. Each stream uses a byte-stream orientation.

QUIC protects the receiver by credit-based flow controls for the connection and for each stream. Each byte sent in a STREAM frame uses up the credit. Each byte fetched from QUIC by the application frees space for more data. A QUIC receiver sends a MAX_DATA or MAX_STREAM_DATA frame to increase the connection-level or a stream-level credit for the sender, respectively. A sender that exhausted one of the credits is not allowed to send more application data. It instead sends a (STREAM_)DATA_BLOCKED frame to inform the receiver.

A QUIC packet containing frames other than a PING frame (i.e. a frame used to check reachability to the peer) or an ACK frame is an ack-eliciting packet. When QUIC sends an ack-eliciting packet, it expects an ACK frame that acknowledges the packet. If the packet is not acknowledged after a specified time, QUIC sends a probe packet to trigger an ACK frame. A packet that was left unacknowledged upon a received ACK frame is considered lost, when the packet number is too low or the sent time too old compared to the acknowledged packet with the largest packet number. In case of a packet loss, QUIC does not retransmit the packet. It rather inspects the contained frames and checks if the information needs to be retransmitted. A STREAM frame typically contains information that needs to be retransmitted, while an ACK frame might contain outdated information. QUIC retransmits information always in a new QUIC packet with a new packet number.

III. Implementation of QUIC in INET

INET is a network simulation model suite that uses the OMNeT++ simulation library [3]. INET contains so called

ISBN 978-3-903176-28-7

simple modules for typical network protocols, e.g. TCP, UDP, Stream Control Transmission Protocol (SCTP), IP, Point-to-Point Protocol (PPP), and Ethernet. They can be combined to a compound module. One such compound module is the StandardHost. The StandardHost combines simple modules to a full TCP/IP network stack, including link layer, network layer, transport layer, and application layer.

We extended INET by a simple module for QUIC (in the following referred to as QUIC module) and simple modules for applications that use QUIC to send and receive data. We included the QUIC module into the StandardHost, placed between an application module and the UDP simple module.

The QUIC implementation is based on the IETF internet drafts [1] and [2] with a focus on the transport layer features of QUIC. It is planned to contribute the implementation as open source to the INET model suite.

A. Realized and Missing Features

For the implementation of the QUIC module, we focused on the transport layer features required to simulate data transfer. Therefore, we implemented the in-order delivery, reliability, packetization, congestion control (CC), flow control (FC), and stream multiplexing.

Currently, the QUIC module models the established state only. It does not perform a connection setup and therefore omits the exchange of parameters between the endpoints[1]. It also does not encrypt packets. The overhead by encryption is not modeled. As such, the space available in a packet for QUIC frames is larger than in a real implementation.

B. Data Sender

When an application signals the need to connect to another host via QUIC, the QUIC module immediately indicates that it is ready to send data (it skips the connection setup process). After that, the application is able to hand over data it needs to transmit. For that, the application has to specify the number of the stream it intends to use to transmit the data. The QUIC module manages a send queue for each stream. It stores the provided data in the send queue of the specified stream and starts the sending process. If the CC allows to send a full-sized packet (i.e. a packet that fully utilizes the Maximum Transmission Unit (MTU) of the network path), the QUIC module builds a packet. Building a packet starts with adding the QUIC short packet header. Then, frames other than STREAM frames are added, if available. After that, if the FC allows to send data, one or more STREAM frames are added for the data in the stream send queues.

A scheduler decides which STREAM frames are added to the packet. We implemented a generic scheduler interface. With that, it is simple to change the scheduler mechanism. Currently, merely a Round Robin scheduler exists. This scheduler adds one STREAM frame, if the send queue of the current stream can provide enough data to fill the packet. If not, it adds STREAM frames for the next streams until the packet is full or no stream is able to provide data to send.

[1]Parameters in a configuration file compensates the lack of exchange.

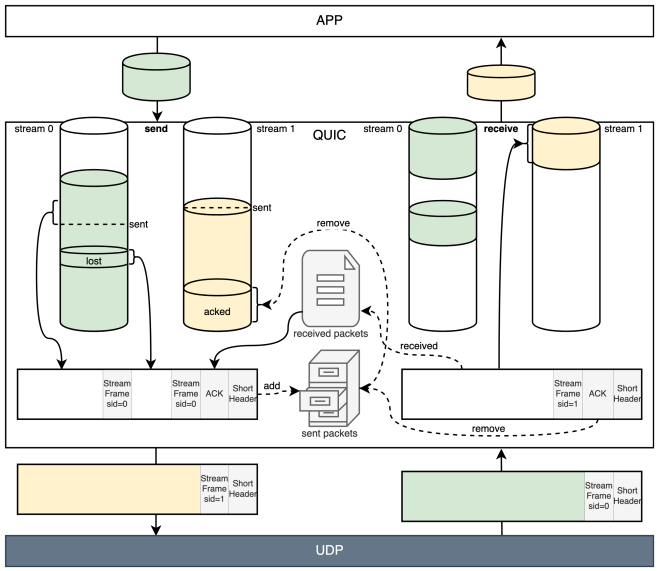

Fig. 1: Send (left) and receive (right) architecture of the QUIC module.

Before a packet will be forwarded to UDP, the packet is added to a dictionary of sent packets and the application data from the send queue is copied into the STREAM frames. The data remain in the stream's send queue until the peer host acknowledges a packet containing these data. The total size of all stream send queues is limited. If this limit is reached, the QUIC module signals to the application that it will not accept more data. Once the total size reduces to a low water mark, the QUIC module signals to the application that it is ready to accept new data. The limit and the low water mark are specifiable by parameters.

The left side of Figure 1 depicts the send architecture of the QUIC module for a connection with two streams. The cylinders represent the stream send queues. It shows how a packet is built with an ACK frame, and two STREAM frames for stream 0 (one with lost data and one with new data). A copy of the packet is stored in the dictionary of sent packets, before the application data is added and the packet is forwarded to the UDP module.

C. Data Receiver

When the QUIC module receives a packet for an open connection, it starts by processing each frame on its own. For a STREAM frame, the containing application data are stored in the receive queue of the corresponding stream. Each stream manages a separate receive queue. The size of each stream receive queue is limited, as well as the total size of all stream receive queues is limited by configurable parameters.

When data were received in order, the QUIC module will provide them directly to the application. In this case, the QUIC module signals to the application that it is able to provide newly received data for the specific stream. If the application requests data for that stream, the QUIC module sends them to the application and removes them from the stream receive

queue. When the data were received out of order, the QUIC module waits for the data that fit into the gap between the previously received and the newly received data, before it will provide them to the application.

Once all frames were processed, the packet is considered fully received and added to a list of received packets. How the QUIC module responds depends on the received packet. When the packet received is not ack-eliciting, the QUIC module does not send an ACK frame. If no ack-eliciting packets were received, it may include an ACK frame in the next ack-eliciting packet it has to send. Whether it should bundle an ACK frame for non ack-eliciting packets is configurable by a parameter. When the packet received is ack-eliciting, the QUIC module does a delayed acknowledgment. Upon the first packet received since the last ACK frame was sent, a timer is started. An ACK frame is sent bundled in an ack-eliciting packet that needs to be sent because of another reason, or in an own packet if either the timer expires or the packet received was the n^{th} one since the last ACK was sent. n is a configurable parameter that specifies the ack ratio to $1{:}n$. The parameter is 2 by default.

The right side of Figure 1 shows the receive architecture of the QUIC module for a connection with two streams. The cylinders represent the stream receive queues. It shows how a packet is processed that contains a STREAM frame for stream 1 and an ACK frame. The application data in the STREAM frame is stored into the stream receive queue. The ACK frame acknowledges a previously sent packet with a STREAM frame for stream 1. The acknowledged packet is removed from the dictionary of sent packets and its contained application data is removed from the send queue. After the received packet is fully processed, it will be added on the list of received packets.

D. Congestion Control

QUIC is designed to work with different CC algorithms. To make it simple to implement and integrate different algorithms, we implemented a generic CC interface. A parameter decides which CC algorithm to use. We implemented New Reno, as this is the CC specified for QUIC. The New Reno implementation increases the congestion window in congestion avoidance either in an approximate way as shown in [2] or in a more exact way by counting the acknowledged bytes as recommended in [4]. A parameter specifies which way to use. We also implemented a CC that does not limit the number of outgoing packets. It basically disables the CC.

E. Flow Control

For the QUIC module, we specify the stream receive queue size and the total size of all stream receive queues by parameters. Based on that, the initial FC credits are determined, which a QUIC host communicates during connection setup. Since the QUIC module skips the connection setup, it interprets these parameter values also as the values of the peer.

The MAX_DATA and MAX_STREAM_DATA frames are used to increase FC credit. In order to keep the overhead by these frames low, the receiver generates them only if the remaining credit falls below a specific threshold. The threshold is configurable by a parameter, which is the half of the initial credit by default.

Fig. 2: Generic network for the case studies.

F. QUIC Applications

To be able to simulate the behavior of the QUIC module, we created two QUIC applications. One named Traffic Generator simply generates data messages in a configurable interval and size to forward it to QUIC for transmission. The other named Discard Server simply fetches data from QUIC as soon as possible.

IV. Validating the Simulation Model

We used two approaches to validate the simulation model. In order to check the behavior on a packet level, we used *packetdrill* [5]. Packetdrill allows to write simple scripts that help testing a network protocol implementation. Rüngeler [6] ported packetdrill into the INET simulation model suite. We extended packetdrill in INET to support the QUIC module and a syntax to describe QUIC packets. Our packetdrill scripts check in specific scenarios whether the QUIC module reacts on API calls or received packets correctly and whether it sends the right packet at the right time. In order to check the behavior when interacting with another host, we ran the simulation in specific cases and statistically analyzed the result, where the correct result is known. Four of these case studies are described in the following subsections.

For the case studies, we used a network for the simulation as shown in Figure 2. The links between hosts and routers are ideal links with an unlimited bandwidth and zero delay. The Discard Server application runs on the receiver. The Traffic Generator application runs on the sender. Since each case needs a saturated sender the Traffic Generator application is configured to generate data messages as fast as possible with a size of at least the initial CC window. For the same reason the QUIC send queue limit is set to a value of at least the maximum of the amount of data the sender is allowed to have outstanding at any given time.

A. Link Bandwidth Case

The Link Bandwidth case shows that the QUIC simulation model is able to fully utilize the bandwidth (BW) of the bottleneck link. In this case, the saturated sender fills the drop tail queue of router R1 until the router has to drop packets. When the sender detects packet loss, its New Reno CC reduces the congestion window, which in turn reduces the number of outgoing packets.

Since, in the simulation, there are no side effects and the time hosts need to process packets are not accounted,

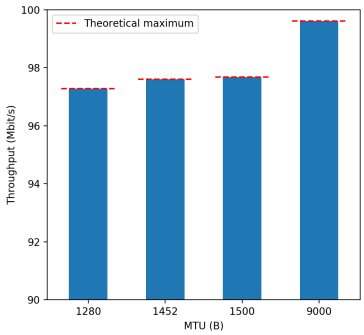

Fig. 3: Throughput using a $100\,\mathrm{Mbit/s}$ link.

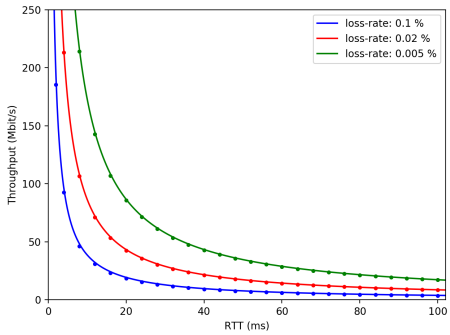

Fig. 4: Throughput using a link with packet loss.

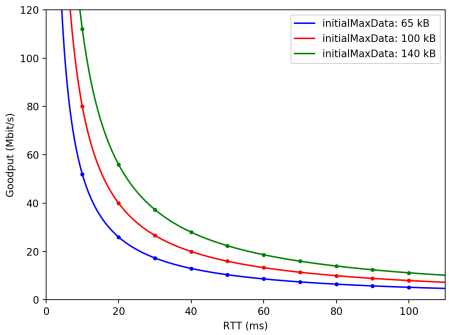

Fig. 5: Goodput in a FC limited connection.

the measured throughput should be equal to the theoretical maximum. For the QUIC layer, this is

$$TP_{max} = \frac{maxQuicPacketSize}{maxPacketSize} \cdot BW \tag{1}$$

where $maxPacketSize$ is the total size of the packet and $maxQuicPacketSize$ is the MTU reduced by the IP and UDP header size.

For the simulation, we configured the bottleneck link with a bandwidth of $100\,\mathrm{Mbit/s}$ and a delay of $1\,\mathrm{ms}$. We set the queue size of router R1 to 23 packets. This value was derived from the Bandwidth-Delay-Product, which is the ideal queue size as described in [7]. We repeated the simulation for different values of $MTU = 1280, 1452, 1500, 9000\,\mathrm{B}$.

We simulated a QUIC connection with over eight seconds of data transfer. To calculate the throughput, we sum up the bytes received by QUIC on the receiver for five seconds (omitting the first three seconds to avoid measuring effects from CC in slow start). As Figure 3 shows, the measured throughput for each MTU equals the theoretical maximum calculated by (1).

B. Lossy Link Case

In contrast to the last case, a fixed packet loss rate is used. The packet losses regulate the New Reno CC at the sender, which limits the throughput. Mathis et al. presented in [8] an equation for throughput, when using TCP on a link with a fixed packet loss rate.

As basically the New Reno CC algorithm using a stream-oriented transport protocol was analyzed, the equation is also applicable for QUIC, when using New Reno as specified by [2]. Adapting the throughput (TP) equation for QUIC gives us

$$TP = \frac{maxQuicPacketSize}{RTT}\sqrt{\frac{3}{2p}} \tag{2}$$

For the simulation, we chose a configuration that fulfills the assumptions used for the equation. These are the following.

(a) New Reno CC increases its window in congestion avoidance exactly by $maxQuicPacketSize$ bytes per RTT.
(b) The packet loss is periodic, i.e. exactly one packet is dropped after $1/p$ packets were transmitted successfully.
(c) Only the packets from the sender are dropped, not the packets from the receiver.
(d) The receiver immediately sends an ACK frame for each received packet.
(e) The bandwidth of the bottleneck link is high enough to not be the limiting factor.
(f) The FC gives enough credit to not be the limiting factor.

For (a), we configured the QUIC module to use the more exact way of counting the acknowledged bytes as recommended in [4]. For (d), we set the ack ratio to 1:1. For (e), we set the bandwidth of the bottleneck link[2] to $100\,\mathrm{Gbit/s}$. We further set the MTU to have $maxQuicPacketSize = 1252\,\mathrm{B}$. We repeated the simulation for different values of $RTT = 2, 4, 8, 12, ..., 100\,\mathrm{ms}$ and $p = 0.1, 0.02, 0.005\,\%$.

To measure the throughput, we sum up the number of bytes received by QUIC on the receiver starting after the New Reno CC on the sender changed from slow start to congestion avoidance. Figure 4 shows the result. The curves describe the theoretical throughput using (2). The small dots describe the measured throughput from the simulation. The fact that these dots are located on the curves shows that the result from the simulation conforms to the expected result.

C. Flow Control Limited Case

This case validates the function of the stream FC in the QUIC simulation model (validating the connection FC has been done in the same way). For that, the CC is disabled. The saturated sender sends packets as long as the FC allows to send application data. Once the FC credit is exhausted, the sender has to wait for a credit update from the receiver.

The sender starts with using the full initial FC credit ($initialMaxData$) by sending $initialMaxData$ bytes of application data in QUIC packets to the receiver. Since the application on the receiver fetches data from the QUIC stream as soon as possible, after the packets arrived, the receiver will increase the FC credit by $initialMaxData$ and informs the sender by sending MAX_STREAM_DATA frames. The

[2]An unlimited bandwidth fits even better to the equation but would have a negative impact on the throughput of QUIC. The sender would send packets at the exact same time. This makes it unable to detect the order of sent packets by time, which is necessary when the sender detects a packet loss, because only acknowledgements for packets sent timely after the lost packet are allowed to use to increase the congestion window.

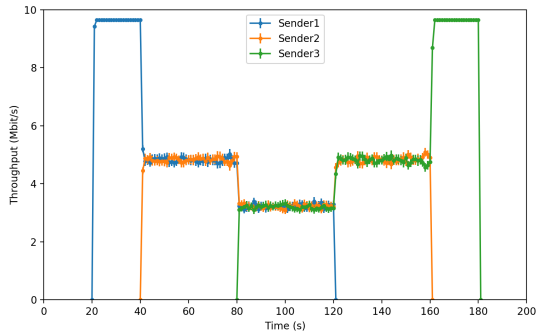

Fig. 6: Throughput using a link shared among three senders.

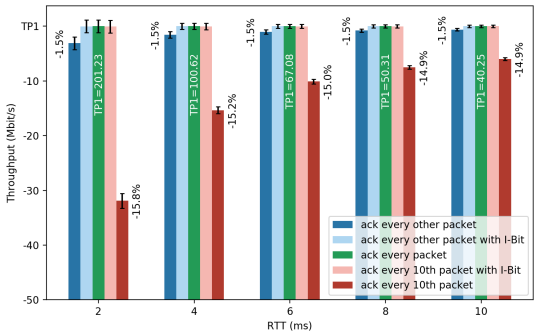

Fig. 7: Throughput for different ack ratios with $0.1\,\%$ random packet loss.

information about the new FC credit arrives at the sender after one RTT, whereon the sender again sends $initialMaxData$ bytes of application data. Therefore, the sender is able to send $initialMaxData$ bytes of application data per RTT, which gives us the equation for goodput (GP).

$$GP = \frac{initialMaxData}{RTT} \tag{3}$$

For the simulation, we configured the bottleneck link with an unlimited bandwidth and disabled the CC on the sender. The application on the sender uses solely stream 0. We repeated the simulation for different values of $RTT = 2, 4, 10, 20, ..., 100\,\mathrm{ms}$ and initial FC credit for stream 0 on the receiver $initialMaxData = 65, 100, 140\,\mathrm{kB}$.

The curves in Figure 5 show the theoretical goodput by using (3) for the three values of $initialMaxData$. To calculate the goodput from the simulation, we sum up the application data in bytes received for stream 0 on the receiver. The measured goodput is shown by the small dots in Figure 5. The fact that these dots are located on the curves shows that the result from the simulation conforms to the expected result.

D. Shared Link Case

The Shared Link case aims to show that QUIC senders are able to share a bottleneck link in a fair manner. We extended the network shown in Figure 2 by three additional senders and three additional receivers. Each sender only sends packets to its corresponding receiver. Sender1, Sender2 and Sender3 are saturated QUIC senders. Each of these senders starts sending at a different time and has to regulate its send rate for fair sharing. Sender4 sends UDP packets in a random fashion to reduce traffic phase effects [9].

When only one sender is active, it should achieve the maximum throughput possible by the link bandwidth, as given by (1). Two or three active senders should achieve a half or a third of the maximum throughput, respectively.

For the simulation we configured the bottleneck link with a bandwidth of $10\,\mathrm{Mbit/s}$ and a delay of $10\,\mathrm{ms}$. We further configured Sender1 to send from $20\,\mathrm{s}$ to $120\,\mathrm{s}$, Sender2 to send from $40\,\mathrm{s}$ to $160\,\mathrm{s}$ and Sender3 to send from $80\,\mathrm{s}$ to $180\,\mathrm{s}$, as well as Sender4 to send UDP packets all the time with a random size that is uniformly distributed from $100\,\mathrm{B}$ to $1000\,\mathrm{B}$ in random intervals that are uniformly distributed from $25\,\mathrm{ms}$ to $100\,\mathrm{ms}$. To compensate random effects, we repeated the simulation 100 times.

To visualize the throughput in a time curve, we measured one value per second. We did that by summing up the bytes received by QUIC on each of the three receivers for each second. We repeated that for all 100 runs to calculate the average throughput and the confidence interval. As Figure 6 shows, the senders are able to share the bottleneck link in a fair manner.

V. QUIC Throughput for different Ack Ratios

The IETF QUIC working group specified in [1] that a QUIC endpoint should send an ACK frame for at least every second received ack-eliciting packet. Whether this ack ratio of 1:2 is a good choice is an ongoing discussion in the working group. Choosing an ack ratio is a tradeoff. Reducing the frequency of ACK frames reduces the workload for the sender and the receiver of the frame, as well as for the network. Increasing the frequency of ACK frames gives the data sender a more up to date view of transmitted and lost packets. Fairhurst et al. suggest in [10] to change the ack ratio to 1:10. Iyengar and Swett propose in [11] an extension that allows to change the ack ratio dynamically during a QUIC connection.

A. Simulation

One drawback when sending ACK frames less frequently is a reduced throughput if the sender is limited by the New Reno CC in a way, where it cannot fully utilize the bandwidth of the bottleneck link (e.g. because the link is shared with other senders). For example, the Lossy Link case has this characteristic. We further examined the case to get a better understanding of the issue. As our intention now is to simulate in a more real-world environment, we changed the periodic packet loss to a random packet loss. Due to the random influence, we repeated the simulation 100 times. We also changed the way how the sender increases its congestion window in congestion avoidance to the one shown in the pseudo code in [2], as this is more likely how real implementations do it.

B. Description of the Result

To calculate the throughput, we sum up the bytes received by QUIC on the receiver. It is important to note that the

time the endpoints need to process a packet is not considered. Figure 7 shows the average throughput and the confidence interval for the ack ratios 1:1 (ack every packet), 1:2 (ack every other packet) and 1:10 (ack every 10$^{\text{th}}$ packet) for small values of RTT using a packet loss rate of $p = 0.1\,\%$. The randomness in packet loss causes an increase in throughput as visible when comparing the result for the ack ratio 1:1 with the result from the Lossy Link case shown in Figure 4. The changed way in increasing the congestion window has only marginal impact on the throughput.

For lower packet loss rates using the values of RTT as in Figure 7, the absolute difference in throughput for higher ack ratios keeps almost unchanged. Because of the higher throughput values, the relative difference shrinks. The relative difference between an ack ratio of 1:1 and 1:2 or 1:10 is $-0.7\,\%$ or $-6.2\,\%$ for $p = 0.02\,\%$ and $-0.4\,\%$ or $-3.2\,\%$ for $p = 0.005\,\%$, respectively. As shown in Figure 7, the opposite is true as the RTT increases for a fixed packet loss rate. The absolute difference shrinks, while the relative difference keeps almost unchanged. This changes when the RTT exceeds the ack delay timer of the receiver. Then, both, absolute and relative difference shrink as the RTT further increases.

C. Analysis of the Result

A closer look into the output of the simulation reveals why the throughput decreases for higher ack ratios. The sender is limited by the CC. The congestion window is lower than the bottleneck link bandwidth would have allowed, due to the configured packet loss rate. Therefore, there is no continuous packet flow. Rather, the sender sends the number of packets allowed by the congestion window and waits for the acknowledgements from the receiver. Whenever the number of arrived packets at the receiver, that need to be acknowledged, is not a multiple of n, the receiver with an ack ratio of $1{:}n$ delays the acknowledgement for the last packets received. Clearly, that happens more often with higher n. The receiver delays the acknowledgement until more packets arrive after one RTT or until the ack delay timer expires, whichever occurs earlier. Since each newly acknowledged packet increases the congestion window, this reduces the growth of the congestion window at the sender, which leads to the decreased throughput for higher ack ratios.

The decrease in throughput shrinks relatively for a lower packet loss rate, because the number of packets the sender is allowed to send by the congestion window is larger due to fewer packet losses, while the number of packets that are acknowledged with a delay keeps unchanged. On the other hand, the decrease in throughput remains steady for a higher RTT that still is below the ack delay timer on the receiver, because the reduced number of RTTs (where a delay could occur) for a fixed period of time is even out by the longer delay (which equals one RTT).

D. Proposed Solution

The receiver cannot know when more packets are expected. Only the sender has this information. We propose to specify one of the reserved bits in the QUIC short packet header as *Immediate Bit* (I-Bit) similar to the one specified for SCTP in [12]. With that, a sender is able inform the receiver, before it is blocked in waiting for acknowledgments. A sender should set the I-Bit with the last packet sent before it is blocked. A receiver should send an acknowledgment immediately after receiving a packet with the I-Bit set.

We implemented the I-Bit extension in order to examine the effect. As shown in Figure 7 by the light blue and light red bars, the extension solves this issue for higher ack ratios.

Alternatively, it is feasible to solve this issue with the extension that allows to change the ack ratio dynamically [11]. A sender could set the ack ratio to 1:1 before it is blocked and reset the ack ratio afterwards. However, the I-Bit extension is simpler and does not add overhead.

VI. Conclusion and Outlook

It is to be expected that the QUIC protocol will play a significant role for data transfer in the future internet. For in-depth analysis of the protocol's behavior, we created a simulation model for the INET simulation model suite. In this paper, we described the architecture of the model and its validation. We further used the simulation to analyse the influence of higher ack ratios. The result described in this paper showed the negative impact for throughput and how the I-Bit extension can compensate it.

We will use the simulation model to do further analysis on the protocol's behavior. In order to simulate more aspects of the protocol, we are going to extend the QUIC module step by step. Subsequently, we will contribute our implementation as open source to the INET model suite.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," IETF, Internet-Draft draft-ietf-quic-transport-27, February 2020.

[2] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," IETF, Internet-Draft draft-ietf-quic-recovery-27, March 2020.

[3] INET Network Simulation Model Suite for OMNeT++. [Online]. Available: https://inet.omnetpp.org/

[4] E. Blanton, Dr. V. Paxson, and M. Allman, "TCP Congestion Control," IETF, RFC 5681, September 2009.

[5] N. Cardwell and B. Raghavan, "Drilling network stacks with packetdrill," *USENIX ;login:*, vol. 38, no. 5, pp. 48–52, October 2013.

[6] I. Rüngeler and M. Tüxen, "Integration of the packetdrill testing tool in inet," *CoRR*, vol. abs/1509.03127, September 2015.

[7] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '04. ACM, August 2004.

[8] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the tcp congestion avoidance algorithm," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 3, p. 67–82, July 1997.

[9] S. Floyd and V. Jacobson, "Traffic phase effects in packet-switched gateways," *SIGCOMM Comput. Commun. Rev.*, vol. 21, no. 2, p. 26–42, April 1991.

[10] G. Fairhurst, A. Custura, and T. Jones, "Changing the Default QUIC ACK Policy," IETF, Internet-Draft draft-fairhurst-quic-ack-scaling-01, March 2020.

[11] J. Iyengar and I. Swett, "Sender Control of Acknowledgement Delays in QUIC," IETF, Internet-Draft draft-iyengar-quic-delayed-ack-00, January 2020.

[12] M. Tüxen, I. Ruengeler, and R. R. Stewart, "SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol," Nov. 2013.