ETH zürich

Networked Systems
ETH Zürich — seit 2015

Continuous benchmarking of open-source QUIC implementations

Semester Thesis

Author: Kevin Marti

Tutor: Laurin Brandner, Alexander Dietmüller

Supervisor: Prof. Dr. Laurent Vanbever

April 2024 to July 2024

Abstract

QUIC is a next generation transport layer protocol with fundamental improvements compared to TCP. In this work we analyze the differences between the two QUIC implementations MsQuic and Quinn under different networking conditions and compare them to the TCP-Stack with iPerf3. We find that increasing the segment size for Quinn increases it's performance in all test scenarios up to doubling it's performance in the best cases. By adding packet loss or reordering to the network we observe that MsQuic outperforms Quinn by at least 50% in non-ideal conditions.

Contents

Chapter 1

Introduction

1.1 Motivation

Quic is a new transport layer network protocol, which has been standardized in 2021 by the IETF [2]. Many big stakeholders of the internet have deployed QUIC alongside their traditional TCP stack [9] and active research is being done around QUIC. To overcame protocol ossification QUIC was developed as a user-space transport building on UDP [3]. While QUIC has many new features like 0/1-RTT fast handshake, stream multiplexing for the removal of head-of-line blocking, and connection migration, it's user-space design gives new challenges (see section 2.1). While QUIC is usually faster in smaller file transfer were the connection setup cost is higher relative to the transfer time [9]. Research has found that for fast internet connection TCP is generally faster than QUIC due to processing overhead [10]. In this processing overhead is the factor that QUIC is always encrypted, which makes a big part of it's processing time [8].

But all these researches are done at a certain point in time in a certain environment with certain implementations. While these implementations aren't as mature as linux's TCP implementation, they are constantly developed further (for better or worse) and thus can these results give no guarantee of the actual state of the many implementations of QUIC. Therefore a continuous benchmarking is needed over the implementations that guarantees comparable results.

1.2 Task and Goals

On the way to achieve the continuous benchmarking we have to to build a framework which shows differences in implementations. The framework should be easily extendable to new implementations and be easy to automate. Finally the framework should output analytical numbers which can be interpreted by humans or machine. To test the basic framework we build our own basic benchmarking tools using various implementations. In the end we tried to find specific programming differences in the implementations that made them faster or slower in specific cases, which could be adopted by other implementations to improve their performance.

1.3 Overview

In Section 2 we talk about QUIC and it's fundamentals. We then continue to explain the structure and setup of the framework in section 3. After this in section 4 we look at the results we found using the framework. Finally in section 5 we look back and discuss the current state of benchmarking QUIC, our results and what future work can improve on this work.

Chapter 2

Background and Related Work

2.1 Background

QUIC is a user-space transport that comes with enforced encryption. Originally developed by Google and later adopted by the IETF [2]. QUIC was developed on top of UDP to overcome middleboxes which can interrupt the flow of unknown protocols. Being developed in user-space means that it can get updated independently of the kernel and receive updates faster. QUIC has been developed to improve on TCP with various features that would make it faster, more reliable and more secure. It can setup a connection faster than TCP because it uses a 0/1-RTT fast handshake principle, where it combines the mandatory cryptographic (since QUIC must always be encrypted) and transport handshake in one round trip and on later connection to a cashed origin send encrypted data wit no additional round trip. QUICs Stream multiplexing makes it so that on packet loss only the stream of the loss packet is affected while the other streams can continue to be reassembled in contrast to TCPs single bytestream. QUIC supports connection migration by attaching a Connection ID to each connection to allow identifying a connection despise changes to the clients IP and port. But living in user-space also comes with downsides as data has to be copied between user-space and kernel space which requires expensive syscalls. The encryption, the exchange between kernel and user-space and implementation specific code lead to more processing power usage compared to TCP [7] and this reduces the throughput of QUIC. The kernel has added support for UDP/QUIC in form of Generic Segmentation Offload (GSO) and Pacing to mitigate this issue. GSO allows the application to send segments bigger than the MTU to the socket which will then be broken down to MTU sized packets. Pacing is used to mitigate congestion from bursty traffic especially when used in combination with GSO as the application can not control the exact time a packet leaves the interface.

2.2 Related Work

Measurements: In *QUIC is not Quick Enough over Fast Internet* [10] the authors compare QUIC with TCP over fast internet (>500 Mbps). They show that TCP has an advantage over QUIC due to QUICs excessive data packet and QUIC's distinctive user-space ACKs but don't use features like ACK-Frequency and GRO. In *Dissecting Performance of Production QUIC* [9] they compare the performance of different production deployed QUIC servers, they find that the performance is influenced by servers choice of congestion control algorithm and configuration tuning of the clients. *A High-Speed QUIC Implementation* [7] bypasses the Linux kernel networking stack by using the DPDK library and compares it with against other QUIC implementations and the

TCP+TLS stack. They find that it outperforms all tested QUIC stacks and matches TCP+TLS with common optimizations but their implementation and measurements are limited to only one core.

Features: *Optimizing UDP for content delivery: GSO, pacing and zerocopy* [1] talks about the optimizations to the UDP stack like GSO/GRO, zerocopy and pacing and how these features can significantly reduce cycle cost. In *Bandwidth-Delay-Product-Based ACK Optimization Strategy for QUIC in Wi-Fi Networks* [4] designs an ACK frequency optimization scheme to improve performance and reduce operation cost with a focus on Wi-Fi-enabled IoT communications. In *Making QUIC Quicker With NIC Offload* [8] analyze different QUIC implementations and cost of it's component and upon this propose a design for Network Interface Cards but doesn't regard features like GSO and Pacing. Finally in *ECN with QUIC: Challenges in the Wild* [6] finds that only less than 2% of the QUIC hosts passes their ECN validation test.

For a overview of QUIC and it's deployment the paper *The QUIC Transport Protocol: Design and Internet-Scale Deployment* [3] was consulted.

Chapter 3

Testing Framework

This sections gives an overview of the testing environment, what problems arose and what the limitations are. This should help to put the results better into perspective of what they mean and how they are to be taken for future works.

3.1 Testbed

The entire test setup is run on a single VM using Ubuntu 24.04 and Kernel 6.8.0. To simulate a network between the server application and the client application a network namespace is created using `ip netns` in which `tc qdisc netem` is used for traffic conditioning like delay, packet loss and reordering rate. Furthermore the VM has 2 pinned CPUs which were shielded from the OSs task scheduler. This allows us to execute the client and the server each on their own CPU without getting interrupted by the OS and give more consistent results. We intend to replicate a realistic scenario, which means setting of the loopback MTU size to 1500, but due to a bug in MsQuic Custom (see 3.2) the MTU size for the loopback interface has to be set to 1600. To capture the traffic we use `tcpdump` and analyze the resulting pcap with pyshark in python. Furthermore `strace/ltrace` and `perf` are used to collect system/library calls and produce flamegraphs.

3.2 Implementations

The implementations under testing are MsQuic developed by Microsoft and Quinn founded by Dirkjan Ochtman and Benjamin Saunders. MsQuic is developed in C and Quinn was developed in Rust. Both repositories provide a benchmarking tool developed by themselves. But to have more control over what is happening we decided to develop our own minimalistic benchmarking tools for each implementation. These will be called "MsQuic Custom" and "Quinn Custom". To get verification for our benchmarking tools the individual implementations benchmarking tool are also integrated in the measurements, which will be called "MsQuic Official" and "Quinn Official" respectively. These have no settings tuned and are used out of the box. During the project a potential inefficiency in Quinn was discovered. The experimental patch is only applied to Quinn's official benchmarking tool called "Quinn Official patched".

3.2.1 Settings

We try to set the settings as close to each other as possible to get the best possible comparison of implementation details and not compare differences in settings. Therefore all tools are set to use cubic, since we don't want to measure differences in congestion control algorithm. Furthermore all tools had encryption disabled since it is known encryption uses a lot of CPU [8] and we don't want to compare the encryption algorithm. MsQuic Custom and Quinn Custom have their settings synced but the changes were minimal, see table 3.1 ("-" means no equivalent setting available) and results would be very close to equal had the defaults of each application been used. The window and buffer sizes of each custom implementation are set individually as e.g. increasing their size increased throughput in Quinn while showing no performance difference in MsQuic. The official benchmarking tools have their windows set to the default values.

Setting	MsQuic	Quinn
MinimumMtu	1248	1248
MaximumMtu	1500	1500
MaxAckDelayMs in ms	200	-
IdleTimeoutMs in ms	2000	2000
StreamRecvWindowDefault in Bytes	65536	30'000'000
receive_window in Bytes	-	4'294'967'295
send_window in Bytes	-	10'000'000
datagram_receive_buffer_size in Bytes	-	30'000'000
datagram_send_buffer_size in Bytes	-	30'000'000

Table 3.1: Settings and window sizes

3.2.2 Patch

Both implementations support GSO and therefor send segments that are bigger than the MTU to the socket. During the analysis of Quinn's packet stream we noticed that the biggest segments were only around 15KB. Looking through the code of Quinn, we find that they hardcode a limit of 10 times the estimated MTU size, which explains the 15KB. The patch only changes this hardcoded limit to 40 times estimated MTU size to get a segment size of 60KB. The authors of Quinn noted that the 10 times limit was benchmarked to be optimal, but it goes against our intuition since bigger segments would mean less packet processing overhead like system calls and copying. MsQuic reaches a maximum segment size of 65KB.

3.3 Limitations

We limit ourselves to a very basic setup. So bugs which reside in the implementations or tools used were not hunted down. Also didn't we verify that the implementations were following the QUIC protocol correctly but just trusted the developers. Also did we try to avoid the influence of Congestion Control Algorithm (CCA) as good as possible, wile they are an important part, they are regarded as a separate entity of the implementations and is not in the scope of our work. What we could do, but didn't do, was counting CPU cycles or relating performance to CPU usage, more on this in 5.2. Further did we limit ourselves to the default settings with exceptions, since these settings can take on many mutations with varying impact on our results. Tuning these settings to be the "best" in our case would take an excessive amount of time with no guarantee that it would not render the implementation unusable in another scenario.

Chapter 4

Evaluation

This section contains the measurements of a single download of varying sizes performed under different environment conditions like delay, loss and reordering.

4.1 Base Performance

First we look at the case where we download 1GB under no bandwidth restriction and perfect environment variables, meaning no more delay than the loopback interface has internally and no artificially added loss or reordering.

Figure 4.1 shows that the traditional tcp stack is able to reach 3 times higher throughput than it's best QUIC counterpart. Then there is another big gap between the patched version of Quinn and all other implementations. This shows that the patch for Quinn benefits Quinn heavily in this scenario, since both official Quinn implementations use the same settings. The benefit of the patch is that the implementation has to send roughly 4 times less segments to the socket which cost a syscall each, as one can see in figure 4.2. Further one can see that the both MsQuic implementations and the Quinn Official patched implementation send about the same amount of segments, which is to be expected since all three try to send segments around the size of 65KB to the socket. iPerf3 also sends segments of around 64KB in size, but has many more smaller packages in between the bigger ones, which makes the noticeable difference. The relative difference of segments send and received between tools stay more or less the same across all measurement and will therefore not be looked at anymore.

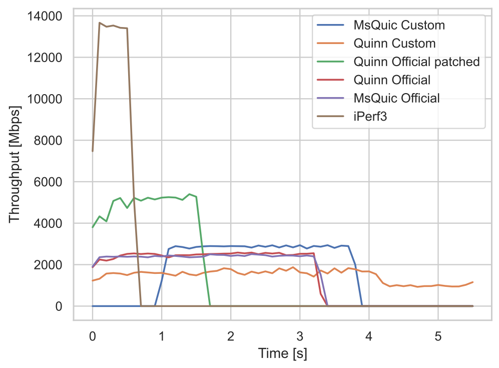

Figure 4.1: Throughput without restrictions when downloading 1GB

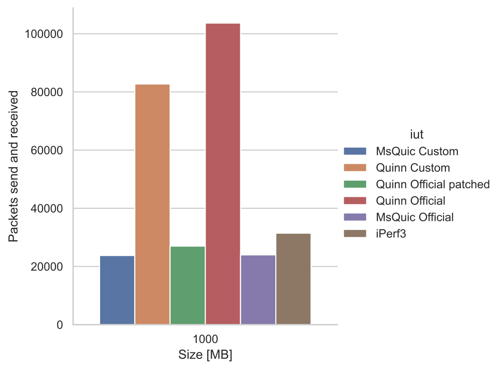

Figure 4.2: Number of segments send and received

The situation changes when we limit the maximal available bandwidth. Figure 4.3 shows that under restricted bandwidth every implementation is more or less equally fast due to reaching the limited bandwidth equally fast.

Remark: Note that in figure 4.1 and figure 4.3 MsQuic Custom has a delayed start in all measurements. This is known bug in **our** benchmarking tool, which could not be fixed in time for the report. The delay varies but normally is around 0.5 seconds, it is not subtracted from the times.

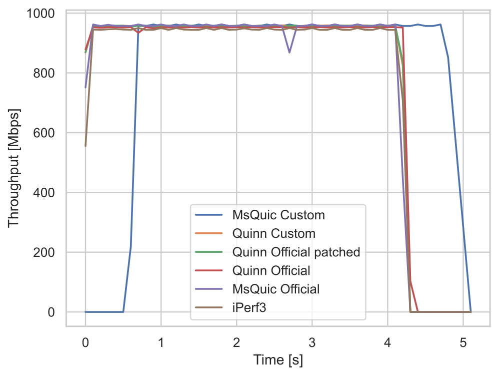

Figure 4.3: Throughput with 1000Mbps bandwidth limit

4.1.1 strace

We use `strace` to get some information about the implementations behavior. While more system calls do not necessarily mean worse performance, they could indicate a point of improvement. In table 4.1 we see the most time consuming syscalls made by the server in the unrestricted measurement in 4.1. One difference is time spent with syscalls. Comparing MsQuic in table 4.1c with Quinn in table 4.1a, MsQuic spents 3 times more time with syscalls than Quinn. An important aspect of the syscalls are the I/O syscalls (marked in bold) which are responsible for communicating with the socket and make the biggest portion of the total syscalls. Looking at them for the two different implementations shows that Quinn has double the amount of I/O calls. Which are directly related to the amount of segments Quinn sends in comparison to MsQuic (see figure 4.2), because comparing Quinn with the patched version in table 4.1b, which sends approximately the same amount of segments as MsQuic, shows a much lower amount I/O calls and also spends less time with syscalls, further indicating a benefit of sending less but bigger segments.

4.2 Influence of Delay, Loss and Reordering

Taking a step closer to the real world, this section inspects the influence of delay, loss and reordering on each implementation. Each test downloads 200MB with different network conditions and a 1000Mbit/s bandwidth. The 200MB is chosen, because it would allow for the implementations to reach their I/O-Limit and then analyze the impact of delay, loss and reordering. A further benefit of having a bigger download is to increase packet loss/reordering, since a too low packet count could lead to no loss at all.

% Time	Seconds	Calls	Syscall
95.59	0.567138	71824	**sendmsg**
3.82	0.022683	4460	**recvmmsg**
0.34	0.002042	866	epoll_wait
0.06	0.000357	1	execve
0.05	0.000276	12	setsockopt
0.03	0.000176	18	mmap
0.01	0.000057	7	**read**
		...	
99.42	0.589878	76291	total **I/O calls**
100.00	0.593317	77307	total

(a) Quinn

% Time	Seconds	Calls	Syscall
93.93	0.186980	19065	**sendmsg**
4.84	0.009633	3681	**recvmmsg**
0.77	0.001537	921	epoll_wait
0.14	0.000283	12	setsockopt
0.08	0.000154	18	mmap
0.03	0.000062	18	write
0.03	0.000056	7	**read**
		...	
98.8	0.196678	22753	total **I/O calls**
100.00	0.199067	23815	total

(b) Quinn Patched

% Time	Seconds	Calls	Syscall
44.93	0.869767	103	futex
37.57	0.727442	8516	epoll_wait
11.22	0.217295	15482	**sendmsg**
3.86	0.074775	17024	**recvmmsg**
1.35	0.026145	30539	gettid
0.35	0.006829	1987	**read**
0.17	0.003232	1744	lseek
0.10	0.001900	731	rt_sigprocmask
0.07	0.001270	125	mmap
		...	
15.43	0.298723	34493	total **I/O calls**
100.00	1.935990	77636	total

(c) MsQuic

Table 4.1: Syscalls by each server

4.2.1 Delay

Starting off with only measuring the impact on implementations when we introduce 10ms of delay into the network and download a 200MB File. In figure 4.5 one can see that Quinn Custom still performs at the bandwidth limit while all other implementations start to fall of and other measurements showed that they would fall off even further with more delay. Quinn Custom can still perform because it has it's window sizes set to 30MB each. Which is not a realistic scenario, but works in our testing environment. MsQuic does not allow to configure it's send window and increasing the receiving window did not affect this benchmark. On the other hand Official Quinn patched did not perform better than the base version. During this measurement we experimentally increased the window size of iPerf3 to 60MB which would allow it to reach the bandwidth limit, but following tests with packet loss were no longer possible, which resulted in using the default for all measurements.

We think this difference in performance does come from two sources. For one, the different maximal achievable window sizes do seam to play a role in the peak throughput, because Quinn

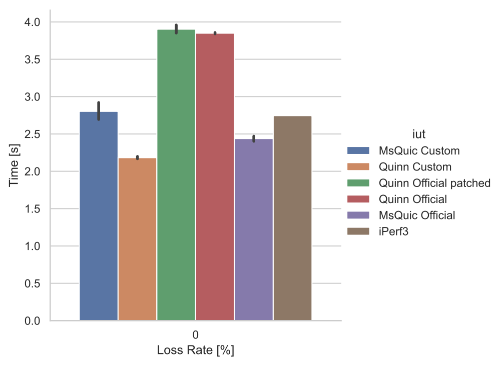

Figure 4.4: Time to download 200MB with 10ms delay

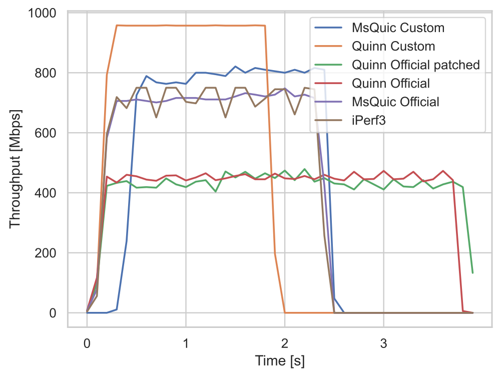

Figure 4.5: Throughput during download with 10ms delay

Custom has the window sizes (sending and receiving) set the highest and also performs the best here. MsQuic seems to use a dynamic sending window [5], which even outperforms iPerf3. On the other hand we don't believe that Quinn is unable to transfer more than 500Mbit/s over a fairly quick link. Therefore we expect that there is a unexpected behavior going on with our setup of Network-Namespace, Network-Emulator and Quinn, which we could not identify, more in 5.2.

4.2.2 Loss

We next add packet loss to our network emulator. Since `netem` requires delay to be present to emulate packet loss and reordering, the following measurements are done with 10ms delay and varying packet loss.

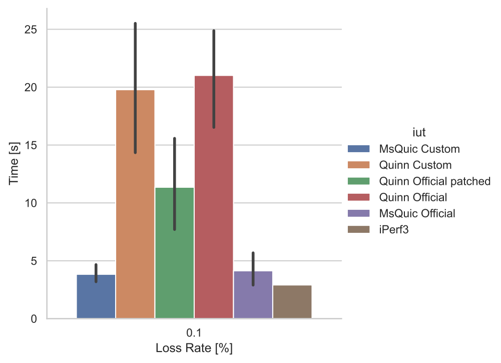

Figure 4.6: Time to download 200MB with 0.1% packet loss

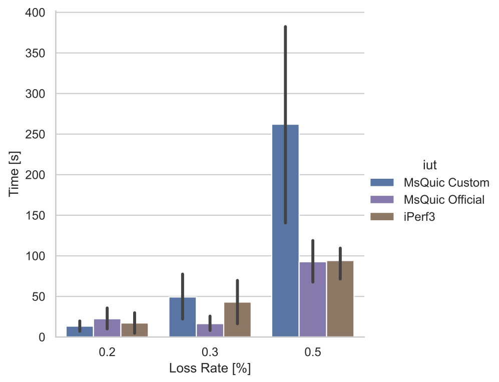

Figure 4.7: Time to download 200MB with packet loss >0.1%

Comparing the times with loss rate of 0 and 0.1 in figure 4.4 figure 4.6, it shows that Quinn handles loss very badly. Official Quinn patched, the fastest version of Quinn in this case, is 3 times slower than MsQuic. While the unpatched versions can go up to 5 times slower than MsQuic. It also shows that Quinn's time has a much higher deviation. We continue the measurements with higher loss rates in figure 4.7 but without the Quinn benchmarks since they would not be able to complete the download in a timely manner. We see that MsQuic Custom download time explodes with 0.5% loss rate, but this appears to be a bug or miss configuration, since MsQuic Official handles the packet loss as good as iPerf3.

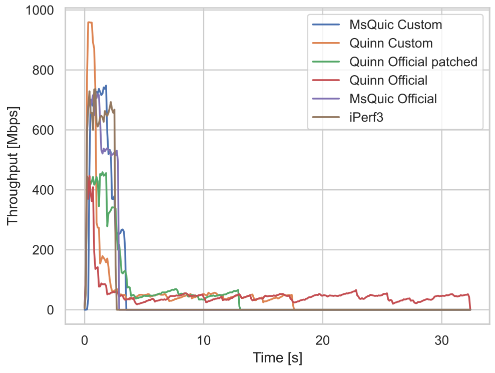

Figure 4.8: Throughput during download with 0.1% packet loss

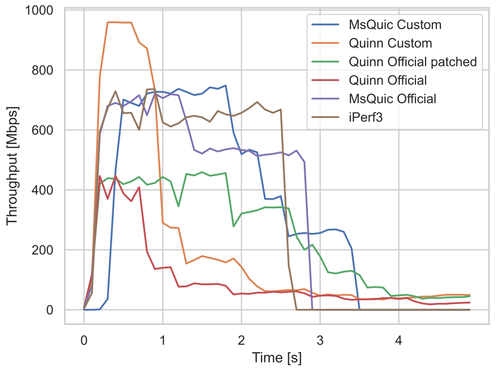

Figure 4.9: Throughput during download with 0.1% packet loss zoomed in

Looking into the throughput in figure 4.8 (or figure 4.9 for a zoomed in version) shows that after an initial rise to peak throughput the loss causes the throughput to fall off and then never recover above a stable point. While MsQuic also has this behavior, it doesn't fall off that much. iPerf3 shows to be the most resilient with keeping it's throughput almost the same as with no loss.

As with delay, we don't think this behavior can be normal for Quinn. At the same time MsQuic is showing that it can handle the network conditions quiet as good as iPerf3. What could be a reason is that the default window sizes of Quinn are too big. Which would then mean that the response for the packet loss could take too long, which would result in packets arriving but considered loss since they arrived too late, which would then explain the downwards spiral of Quinn.

4.2.3 Reordering

This measurement follows the same setup as the previous one for the same reason. Adding 0.1% of packet reordering to the stream instead of loss should lead to an increased performance. Figure 4.10 shows that this is the case for MsQuic and iPerf3. MsQuic gained close to 20% performance boost while iPerf3 performed only 5% better. Here we also continue the benchmark with higher reordering rate without the Quinn benchmarks for the same reason as previously. Figure 4.11 shows that MsQuic Custom and Official are able to deal better with packet reordering than iPerf3.

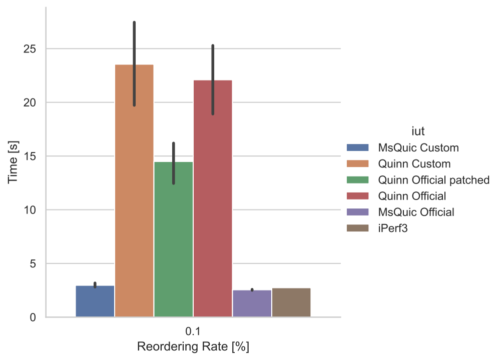

Figure 4.10: Time to download 200MB with 0.1% packet reordering

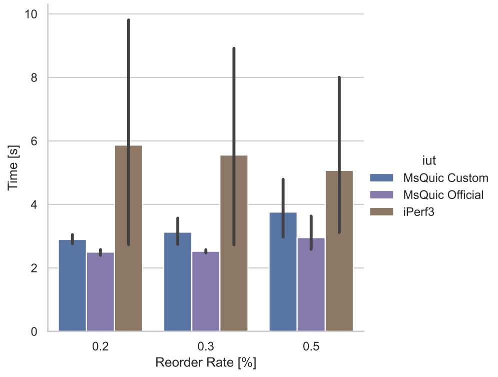

Figure 4.11: Time to download 200MB with >0.1% packet reordering

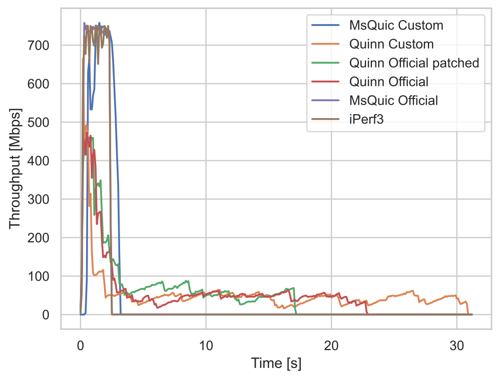

Figure 4.12: Throughput during download with 0.1% packet reordering

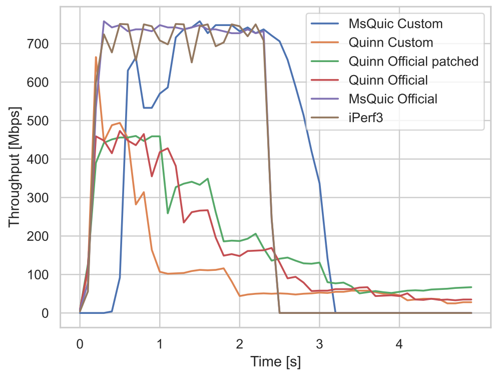

Figure 4.13: Throughput during download with 0.1% packet reordering zoomed in

Looking at the throughput in figure 4.12 (zoomed in figure 4.13) it shows the same behavior for Quinn as when it handles loss, the throughput rises and then starts to fall off and doesn't recover. In comparison MsQuic manages to hold it's throughput high. iPerf3 which was marginally affected by lower loss rates, is even less affected by reordering at reordering rates. The reasoning follows the same as with loss.

4.2.4 Conclusion

Concluding all the test results show that there are big differences between all implementations including iPerf3. While iPerf3 with it's TCP stack still dominates in unrestricted pure performance, our tests show that in Quinn's case a simple patch to increase the segment size can not only increase the throughput massively but also reduce syscalls and related costs. We also learn that in Quinn's case tuning window/buffer sizes can lead to better results when working with delay. The most notable difference in performance between the implementations is when packet loss or reordering occurs. MsQuic handles packet loss much better than Quinn resulting in over 400% faster download in the best case and still over 50% faster download in the worst case. Performance with packet reordering show similar results. Finally, throughout all tests the patch for Quinn performed equally or better than the non-patched version.

Chapter 5

Outlook

In this Chapter we conclude what we learned throughout the work and look into what would be the next steps to dig deeper into the problem of what makes implementations better or worse in certain situations.

5.1 Lessons Learned

Here we recap in short form our experience we gathered from this work and the findings we made.

Lesson #1: Working with QUIC still requires a somewhat high expertise. While documentations for each implementations exist, they all are very programming specific and can miss specific information which then requires one to look into the source code or benchmarking. The absence of bigger reference projects don't support this point.

Lesson #2: Emulating the network works quite well in our setup, but it always leaves a amount of uncertainty about the results. For a final and conclusive setup a hardware setup is advised.

Lesson #3: From our results, we saw that the window sizes have a great influence on the throughput performance. Also can they influence the recoverability of packet reordering and loss. MsQuic doesn't allow to set the max window size while Quinn allows it, which in return is one parameter more the developer has to tune. What approach is preferable in the long run time will show.

Lesson #4: Throughout all measurements sending less but bigger segments to the socket has a positive impact, from higher throughput, to faster downloads in unstable networking conditions and less syscalls. This is something developers should always look into when implementing QUIC.

5.2 Future Work

In this last section we discuss the various ways this work can be extended or be improved on. The framework can be extended in all ways imaginable, but it is laid near to improve on the current measurements we have. While current measurements show which implementation is faster, it doesn't relate it to CPU performance. Measuring CPU performance in relation to throughput would allow one to scale the results and perhaps uncover wasted cycles in the implementation. In the same line of thought is to implement multiple connection into the benchmarking tools to bring the server to the limits. Both official benchmarking tools of MsQuic und Quinn offer this, but both may need some tweaking to fit our framework. And finally, to remove our uncertainty if our network setup is influencing the results in unexpected manner, we advice to fit this setup to two hardware computers.

Bibliography

[1] de Bruijn, W., and Dumazet, E. Optimizing udp for content delivery: Gso, pacing and zerocopy. In *lpc net2018 talks* (2018).

[2] Iyengar, J., and Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[3] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., and Shi, Z. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, Association for Computing Machinery, p. 183–196.

[4] Liu, Y., Yang, Z., Peng, Y., Bi, T., and Jiang, T. Bandwidth-delay-product-based ack optimization strategy for quic in wi-fi networks. *IEEE Internet of Things Journal 10*, 20 (2023), 17635–17646.

[5] Microsoft. Msquic sendbuffer/window (2023.07.12). `https://github.com/microsoft/msquic/blob/f8d7248b90b2572577d80a88391042091e381815/src/core/send_buffer.c#L29`.

[6] Sander, C., Kunze, I., Blöcher, L., Kosek, M., and Wehrle, K. Ecn with quic: Challenges in the wild. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (New York, NY, USA, 2023), IMC '23, Association for Computing Machinery, p. 540–553.

[7] Tyunyayev, N., Piraux, M., Bonaventure, O., and Barbette, T. A high-speed quic implementation. In *Proceedings of the 3rd International CoNEXT Student Workshop* (New York, NY, USA, 2022), CoNEXT-SW '22, Association for Computing Machinery, p. 20–22.

[8] Yang, X., Eggert, L., Ott, J., Uhlig, S., Sun, Z., and Antichi, G. Making quic quicker with nic offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (New York, NY, USA, 2020), EPIQ '20, Association for Computing Machinery, p. 21–27.

[9] Yu, A., and Benson, T. A. Dissecting performance of production quic. In *Proceedings of the Web Conference 2021* (New York, NY, USA, 2021), WWW '21, Association for Computing Machinery, p. 1157–1168.

[10] Zhang, X., Jin, S., He, Y., Hassan, A., Mao, Z. M., Qian, F., and Zhang, Z.-L. Quic is not quick enough over fast internet. In *Proceedings of the ACM on Web Conference 2024* (New York, NY, USA, 2024), WWW '24, Association for Computing Machinery, p. 2713–2722.