Extending the ns-3 QUIC Module

Umberto Paro[1], Federico Chiariotti[2], Anay Ajit Deshpande[1],
Michele Polese[3], Andrea Zanella[1], Michele Zorzi[1]
[1]Department of Information Engineering, University of Padova, Via Gradenigo 6/B, Padova, Italy
[2]Department of Electronic Systems, Aalborg University, Aalborg, Denmark
[3]Institute for the Wireless Internet of Things, Northeastern University, Boston, MA, USA
Email: paroumbert@dei.unipd.it, fchi@es.aau.dk, deshpande@dei.unipd.it, m.polese@northeastern.edu, zanella@dei.unipd.it, zorzi@dei.unipd.it

ABSTRACT

The recently proposed QUIC protocol has been widely adopted at the transport layer of the Internet over the past few years. Its design goals are to overcome some of TCP's performance issues, while maintaining the same properties and basic application interface. Two of the main drivers of its success were the integration with the innovative Bottleneck Bandwidth and Round-trip propagation time (BBR) congestion control mechanism, and the possibility of multiplexing different application streams over the same connection. Given the strong interest in QUIC shown by the ns-3 community, we present an extension to the native QUIC module that allows researchers to fully explore the potential of these two features. In this work, we present the integration of BBR into the QUIC module and the implementation of the necessary pacing and rate sampling mechanisms, along with a novel scheduling interface, with three different scheduling flavors. The new features are tested to verify that they perform as expected, using a web traffic model from the literature.

CCS CONCEPTS

• **Networks** → **Network simulations**; **Transport protocols**; *Network protocol design*;

KEYWORDS

QUIC, ns-3, transport protocols

1 INTRODUCTION

Over the past few years, the development of new communication technologies, with new capabilities and new kinds of applications, has led to a resurgence of research on the transport layer [22]: the Quick UDP Internet Connections (QUIC) [11] protocol, which has become a strong competitor to the Transmission Control Protocol (TCP) by promising to overcome its main shortcomings, is one of the most interesting developments in the field. Congestion control has also returned to being an active topic of research, with the development of the Bottleneck Bandwidth and Round-trip propagation time (BBR) mechanism [4], which aims at fully utilizing capacity while maintaining low latency.

MSWiM '20, November 16–20, 2020, Alicante, Spain

ACM ISBN 978-1-4503-8117-8/20/11…$15.00
https://doi.org/10.1145/3416010.3423224

The QUIC protocol and the BBR congestion control mechanism were both developed by Google, and are currently used together in a number of commercial products, such as the Chrome browser. According to the estimates provided in [16], 7% of the overall Internet traffic and 30% of Google's egress traffic is currently generated by QUIC connections, although the Internet Engineering Task Force (IETF) standardization process is still ongoing [11]. The design rationale behind QUIC was to mitigate some of TCP's main issues. The first and foremost of these is Head-of-Line (HOL) blocking, which heavily affects HyperText Transfer Protocol (HTTP)/2 Web traffic. HOL occurs because of the strict requirement for in-order delivery, so that packets cannot be released to the application if an earlier one is missing, even if the packets are logically independent and belong to different application-level objects [9]. Like the pre-existing Stream Control Transmission Protocol (SCTP), QUIC solves the issue by defining different *streams* and requiring in-order delivery only for packets belonging to the same stream.

In order to maximize compatibility and avoid issues with middlebox support [20], QUIC is deployed in user space and not in the Operating System (OS) kernel, and its packets are encapsulated in standard User Datagram Protocol (UDP) packets. The protocol also includes full end-to-end encryption, as well as minor improvements in the connection establishment and selective acknowledgment handling. The performance of different versions of QUIC has been studied in a number of recent papers [5, 13, 24], with experiments in real networks based on different open source implementations of QUIC. Our native implementation of the protocol in the ns-3 simulator [7] was based on the design of the ns-3 TCP implementation [21], as the two protocols have similar functions and share several common elements.

In this work, we present a further extension of the QUIC module[1] for ns-3, which significantly extends its functionality and the possibilities for research on the protocol. In particular, we added three features to the implementation, which should cover some of the most interesting research topics on QUIC:

- We implemented pacing for QUIC and integrated the BBR congestion control mechanism from [12] with the protocol, allowing researchers to test the combination of QUIC and BBR, which is becoming commonplace on the Internet.
- We implemented a realistic HTTP traffic model from the literature [23], which is more recent than that used in the current ns-3 implementation and should be a better model of contemporary web browsing.

[1]The source code is available at https://github.com/signetlabdei/quic and on the ns-3 app store.

- We generalized the scheduling among streams, allowing users to choose between the standard First In First Out (FIFO) packet queue, a Prioritized First In First Out (PFIFO) implementation that always prioritizes streams with a lower identification number when sending data, and an Earliest Deadline First (EDF) implementation that allows applications to set per-stream latency bounds and delivers first the packets whose deadline is the closest. The classes implementing these packet schedulers inherit a standard interface, which can be further extended to consider different scheduling policies.

In the remainder of the paper, we first provide an overview of the main features of the QUIC protocol and its ns-3 implementation in Sec. 2. We then describe BBR and its implementation in Sec. 3. Sec. 4 discusses the issues of modeling HTTP/3 traffic and possible innovation, presenting our implementation of a well-known traffic model and several packet scheduling policies and their implementation in the QUIC module. Finally, Sec. 5 concludes the paper.

2 THE QUIC PROTOCOL

The QUIC protocol implements additional transport layer and security functionalities as an overlay of UDP. It was initially designed by Google, in 2013 [16], and later the IETF has promoted an effort to make it a standardized protocol. So far, different Internet Drafts have been under discussion to fully develop the core transport layer features [11], end-to-end security [25], congestion control and recovery procedures [10], and the bindings to an extension of HTTP (named HTTP/3.0) [2]. In the following, we define a *connection* as a transport layer flow with two fixed endpoints, and a *stream* as an application-layer data object or flow with self-contained and independent content. The same application can generate multiple streams, e.g., a webpage with multiple HTTP objects, but should only open one connection to avoid connection setup overheads and congestion control issues. QUIC allows streams to be multiplexed on the same connection while maintaining separate receiver buffers, reducing the delivery delay in case of errors.

QUIC provides in a single layer some of the functionalities that are otherwise split into TCP and the Transport Layer Security (TLS) stack. Besides transport layer features, it implements authentication and encryption of the packet with TLS 1.3 [25]. This prevents common networking attacks, such as packet injection and eavesdropping. Moreover, it makes it harder to perform traffic analysis in the network.

From a transport layer point of view, QUIC inherits and improves several TCP design choices [22]:

- the integration with TLS allows a reduction in the connection establishment latency, as the *protocol* and the *cryptographic* handshakes can be performed in the same exchange (contrary to TCP+TLS, where the connection is first established and then encrypted). The connection setup can thus be completed in a single Round Trip Time (RTT), if the endpoints have never exchanged cryptographic information in previous connections, or even with a single transmission (0-RTT handshake), already encrypted;
- QUIC still infers the conditions of the end-to-end connection using acknowledgments (duplicate or not), and adapts using a congestion-window-based mechanism. However, it improves TCP's retransmission and loss recovery process, as the sender is able to distinguish out-of-order and lost packets, and the Selective Acknowledgment (SACK) signaling can contain more blocks than in TCP [11]. Furthermore, as will be discussed later in this paper, QUIC can feed the congestion control mechanism with additional information, if compared to TCP, thus enabling a more prompt and precise estimation of the connection status. Notably, the receiver explicitly signals the delay between the reception of a packet and the transmission of the associated acknowledgment, for a better estimation of the RTT;
- with QUIC, several streams (with data and control) can be multiplexed on a single connection. The multiplexing exploits a new packet structure, which combines frames of different streams into a single packet. Control frames can also be appended, to carry, for example, acknowledgments or flow control information, at the stream or connection level. The stream concept seamlessly supports HTTP streaming and objects, with the possibility of a one-to-one mapping between application and transport layer streams. This benefits dynamic applications with different reliability requirements on different portions of the data to be transmitted, as different streams can be configured to deliver frames reliably, in-order, or not. As discussed in [22], HTTP/3.0 streams on QUIC do not suffer from the HOL blocking issue that impacts the performance of HTTP/2.0 on TCP, where a single packet loss prevents TCP from delivering the packets of any other stream to the application.

Finally, besides transport and security features, QUIC provides a mechanism to increase the stability of end-to-end connections with respect to changes in the topology or the configuration of the lower layers of the protocol stack. This is done through a connection-level identifier, which makes it possible to maintain the same established connection even in case of—for example—mobility in cellular and Wi-Fi networks that updates the IP address [11].

2.1 The QUIC ns-3 module

`QuicSocketBase` (which extends `QuicSocket`), the main class of the QUIC implementation, models the basic functionalities of a QUIC socket, like in the TCP implementation. Each client will instantiate a single `QuicSocketBase` object, while the server will fork a new socket for each incoming connection. The QUIC socket is bound to an underlying UDP socket through a `QuicL4Protocol` object, which handles the initial creation of the `QuicSocketBase` object, triggers the UDP socket to bind and connect, and handles the delivery of packets between the UDP socket and `QuicSocketBase`.

A `QuicSocketBase` object receives and transmits QUIC packets and acknowledgments, accounts for retransmissions, performs flow and congestion control at the connection level, takes care of the initial handshake and exchange of transport parameters, and handles the life cycle and the state machine of a QUIC connection. An instance of the `QuicSocketBase` class holds pointers to multiple other relevant items, including:

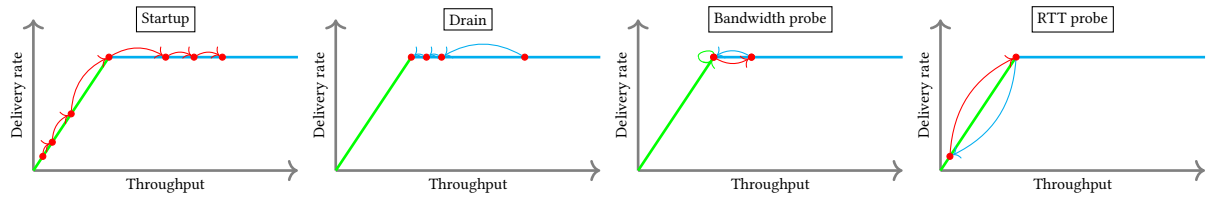


Figure 1: Operation of the BBR algorithm [22]

- the socket transmission and reception buffers, implemented by `QuicSocketTxBuffer` and `QuicSocketRxBuffer`, respectively;
- a `QuicSocketState` object, extending `TcpSocketState` [6] with additional variables that are used by the QUIC state machine and congestion control;
- an object extending the `TcpCongestionOps` class, which performs the congestion control operations and provides a basic compatibility with the TCP congestion control implementations, as we will discuss later.

QUIC streams are modeled by the `QuicStreamBase` class, which extends the basic `QuicStream` class. It buffers application data, performs stream-level flow control, and delivers the received data to the application. Similarly to the full socket, a `QuicStreamBase` object also has pointers to transmission and reception buffers, which are implemented by the `QuicStreamTxBuffer` and `QuicStreamRxBuffer` classes.

Multiple `QuicStreamBase` objects are connected to a single `QuicSocketBase` through an object of the `QuicL5Protocol` class. A `QuicSocketBase` holds a pointer to a `QuicL5Protocol` object, and the latter contains a vector of pointers to multiple `QuicStreamBase` instances. The `QuicL5Protocol` class creates and configures the streams, and takes care of delivering packets or frames to be transmitted and received across the streams and the socket.

The instantiation of a QUIC socket is simplified by the presence of the `QuicHelper` class, which extends `InternetStackHelper` to add QUIC to the supported protocols.

The congestion control for a `QuicSocketBase` instance can (i) be in legacy mode, i.e., the congestion control algorithms implemented for TCP drive the congestion window of the QUIC connection as well; or (ii) use more refined algorithms, which exploit the additional information that the QUIC socket provides, and develop congestion control methods for QUIC only. The QUIC implementation currently includes one such algorithm, namely, the basic NewReno-like congestion control scheme specified in the Draft [10].

The compatibility with legacy TCP congestion control algorithms is achieved by having a `TcpCongestionOps` instance as the basic congestion control object in `QuicSocketBase`. Then, we introduced a new class `QuicCongestionControl`, which extends `TcpNewReno`.[2] `QuicCongestionControl` features additional methods that inject additional information specified by the QUIC Internet Draft in the congestion control algorithm (e.g., the transmission of a packet, more refined information on the RTT) if needed. When the congestion control algorithm is set in `QuicSocketBase` (i.e., after the socket is created by `QuicL4Protocol`), the `SetCongestionControlAlgorithm` checks if the specific algorithm extends `QuicCongestionControl`, and in this case sets the `m_quicCongestionControlLegacy` flag to false. Otherwise, the latter is set to true and the legacy mode is activated. Then, every time the socket needs to trigger the methods of the congestion control algorithm, e.g., when an acknowledgment is received, or a Retransmission Timeout (RTO) expires, it will check whether the operations are in legacy mode to call the relevant methods.

We refer the reader to [7] for additional details on the ns-3 QUIC implementation.

3 BBR CONGESTION CONTROL IN QUIC

Google's BBR [4] is a new congestion control mechanism, which tries to explicitly estimate the Bandwidth-Delay Product (BDP) and stay close to the connection's optimal operating point, defined as full capacity exploitation with minimum RTT [15]. In other words, the objective of BBR is to transmit data at the highest possible rate without creating a queue. BBR uses a capacity-based philosophy, measuring capacity directly like the older Westwood mechanism [18], and uses pacing as its main limit to the sending rate, keeping a larger congestion window to allow for capacity variations. The protocol has four phases, which are represented in Fig. 1:

- In the Startup phase, BBR uses a gain of $2/\ln(2)$ to quickly ramp up the sending rate until the actual bandwidth is discovered. This can create a queue of up to twice the BDP, resulting in an RTT increase of twice the minimum RTT of the connection.
- In the Drain phase, BBR uses the inverse of the startup gain to reduce the queue before starting normal operation.
- The bandwidth probe phase is BBR's normal mode of operation: in this phase, BBR is driven by its capacity estimates. The estimates of the capacity are passed to a max filter, whose output is the protocol's bandwidth estimate. This optimistic estimation mechanism is not without issues, as we will discuss later, but it allows BBR to fully exploit the capacity in stable channels. The protocol then sets the pacing rate to match the estimated bandwidth and the congestion window to twice the BDP, ensuring that the reaction to capacity drops will not be too slow. Since capacity estimates are limited by the pacing rate, BBR periodically probes the bandwidth by temporarily setting the pacing rate to 1.25 times the measured bandwidth. In this way, it builds up a queue and gets more accurate estimates of the capacity, identifying large upswings in the capacity. After one RTT, the protocol spends

[2] As expected, `TcpNewReno` extends `TcpCongestionOps` itself.

another RTT with a reduced pace of 0.75 times the bandwidth in order to reduce the standing queue.
- The RTT probe phase is repeated periodically, with a default period of 10 s. During this phase, BBR updates its estimate of the connection's minimum RTT. To do so, it reduces the congestion window to 4 packets for a short period, flushing the queue at the bottleneck and ensuring that the estimate of the minimum RTT is unaffected by self-queuing delay. Naturally, other flows sharing the bottleneck buffer might still bias the estimate. After an RTT probe, operation resumes normally.

Since it does not interpret loss as a signal of congestion, relying on the BDP estimate to avoid buffer overflows, BBR does not reduce its sending rate as a consequence of packet loss. This gives it an advantage over loss-based mechanisms in naturally lossy connections such as wireless and mobile systems. BBR can also make full use of Explicit Congestion Notification (ECN), as well as a seamless integration with QUIC's SACK and delay estimation mechanisms.

3.1 Implementation in ns-3

In the following, we describe the integration of the BBR ns-3 implementation by Jain *et al.* [12] in the QUIC module. Since the QUIC module's codebase is compatible with the releases after `ns-3.29`, while the BBR is based on the older `ns-3.27` release, the original implementation had to be adapted to the changes made in newer releases. In particular, recovery operations are implemented separately from the QUIC socket class, with a more modular approach. The mechanics of congestion window updates are also slightly different. The updated BBR code was added to the QUIC module.

The implementation of BBR also involved significant changes to the `QuicSocketBase` class in order to add the features that BBR needs for delivery rate estimation and pacing. Pacing is the most relevant feature used by BBR, as adjusting the pacing rate is the main way the mechanism performs congestion control. It is used to send data at a constant rate, matching the estimated capacity and avoiding packet bursts that may cause overflow in the bottleneck link buffer. As the QUIC module is based on the existing TCP implementation design, the new QUIC pacing also follows the `TcpSocketBase` code structure, which includes pacing since the `ns-3.28` release. A `Timer` object is started whenever a packet is sent by the socket, and it is set to expire after a time given by L/B_{pace}, where B_{pace} is the pacing rate and L is the packet size. The `SendPendingData` method of `QuicSocketBase` checks the pacing timer state before sending a packet: if the timer is expired, the packet is sent straight away, and the timer is restarted, while if it is still running a call to `SendPendingData` is scheduled upon timer expiration. In order to comply with the QUIC specification [10], pacing is temporarily disabled before sending packets following Tail Loss Probe (TLP) or RTO events.

Another change to the QUIC module to support BBR is the implementation of rate measurements through the `RateSample` interface, as BBR needs capacity estimates for both its delivery rate estimation and the detection of the application-limited state. Unlike in the TCP implementation, the rate estimation is performed by the `QuicSocketTxBuffer`, which updates the rate sample whenever packets are removed from the buffer to be sent or acknowledgments are processed. A new `RateSample` is generated when new ACKs are processed and passed along to the congestion control class.

In the TCP implementation, the `TcpTxBuffer::OnApplicationWrite` method detects the BBR application-limited state upon the insertion of data in the Transmission Buffer by the application layer. The connection is then considered to be application limited if the data in the application buffer is not sufficient to create a Maximum Transmission Unit (MTU)-sized packet. However, the QUIC version is less straightforward: the application data passes through both the `QuicStreamTxBuffer` and the `QuicSocketTxBuffer` before being sent. Hence, we used a different approach: the connection is considered application-limited if the amount of available application data in the `QuicSocketTxBuffer` when sending the packet is lower then a full MTU. This check is implemented in the `QuicSocketBase::SendDataPacket` method.

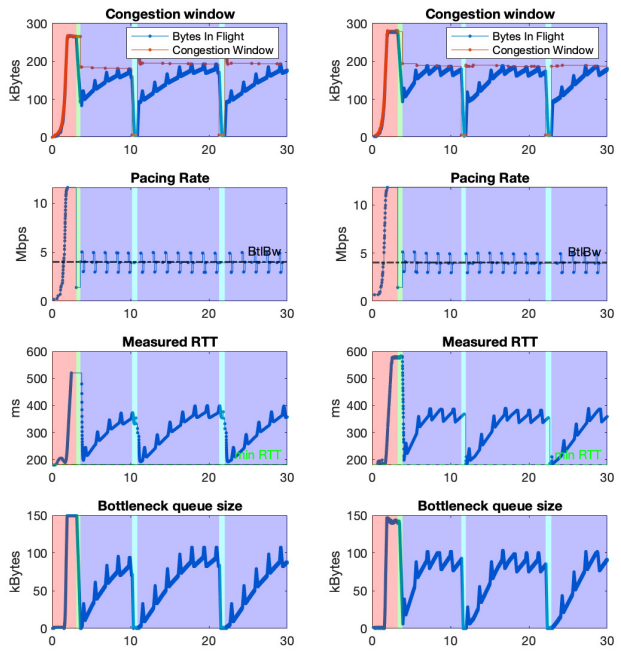
As mentioned in Sec. 2, the QUIC module supports both *legacy* `TcpCongestionOps` inherited from the TCP implementation and protocol specific `QuicCongestionOps` objects for managing congestion control. Since the BBR implementation we adapted requires the changes to the interface that we discussed above, `QuicBbr` is designed as a QUIC-specifc congestion control to preserve the self-contained nature of the module. The higher flexibility of the `QuicCongestionOps` interface, which considers a wider set of congestion events such as ACK reception, packet losses and packet sending, allowed us to implement BBR with no other changes to `QuicSocketBase`, except for the introduction of pacing and rate sampling discussed above.

3.2 Performance evaluation

In the following, we propose a few examples to show that BBR behaves correctly in the QUIC implementation. All the simulation scenarios are based on the `TcpVariantsComparison` and `QuicVariantsComparison` example classes, which use a backed-up source and a dumbbell topology with one or two senders on one side of the bottleneck link and an equivalent number of receivers on the other. The local links have a 10 Mb/s capacity and 45 ms delay, while the bottleneck link has a negligible 0.01 ms delay and its capacity is a variable parameter. The total minimum RTT is then approximately 180 ms. As BBR is designed to converge to Kleinrock's optimal operating point [4] as long as the model parameters are correctly estimated, the measured RTT is supposed to be close to the minimum RTT for the connection in the absence of cross-traffic flows.

First, we consider a constant capacity scenario, in which the bottleneck bandwidth is set to 4 Mb/s. The simulation lasts 30 seconds, and a single sender/receiver pair communicates over the bottleneck. Fig. 2 illustrates the behavior of both the TCP and QUIC implementations of BBR. The different colors in the plot backgrounds describe the different modes in BBR: STARTUP (red), DRAIN (green), PROBE_BW (blue), and PROBE_RTT (cyan).

The behavior of BBR is almost the same in TCP and QUIC, with some minor differences due to the underlying protocols. The congestion window is set to twice the BDP as soon as the drain phase ends, and a slight overestimation of the capacity leads BBR to slowly build up a queue, until it becomes limited by the congestion window (or the buffer is emptied during an RTT probe). The max filter



(a) TCP BBR (b) QUIC BBR

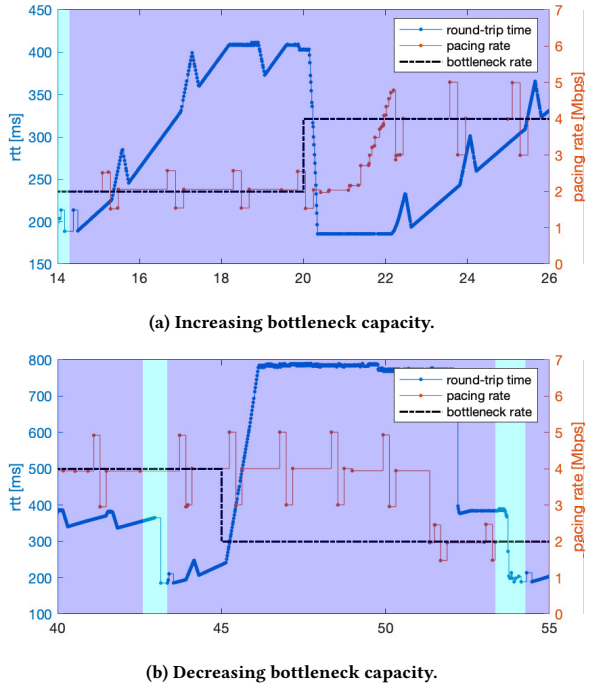
Figure 2: Time evolution of bytes in flight, *CWND*, pacing rate, RTT, BBR modes (background colors) and queue size at the bottleneck node for TCP and QUIC.

slightly overestimates the available capacity, leading BBR to becoming limited by the congestion window instead of the pacing, so the RTT is approximately twice the minimum connection delay. However, this behavior is consistent with the behavior of actual BBR implementations under even very low values of jitter.

Since QUIC has a slightly different delay measurement, its capacity estimate is slightly higher, which leads the protocol to sending data slightly faster than TCP, but the difference is minor.

We then consider another scenario, in which 20 seconds after the simulation begins the bottleneck capacity is increased from 2 to 4 Mb/s, and then decreased back to 2 Mb/s 25 seconds later. Fig. 3a shows how BBR quickly adapts to the available capacity thanks to the pacing gain cycle. As expected, BBR can quickly fill the capacity when it increases. Conversely, Fig. 3b shows the behavior when capacity decreases: it takes a while for the BBR max filter to discard the previous maximum capacity, but the protocol eventually adapts to the change. As for the previous results, this behavior is consistent with the TCP implementation.

Finally, we evaluate the fairness of the protocol by using a constant 4 Mb/s bottleneck capacity, with 2 flows sharing the bottleneck. The first sender immediately starts the connection, while the second waits 15 seconds. The behavior of the flows in terms of pacing rate and RTT is shown in Fig. 4: as expected, the two flows converge to an equal share of the available capacity after an initial transition.



(a) Increasing bottleneck capacity.

(b) Decreasing bottleneck capacity.

Figure 3: Effects of variation in the bottleneck capacity on QUIC BBR.

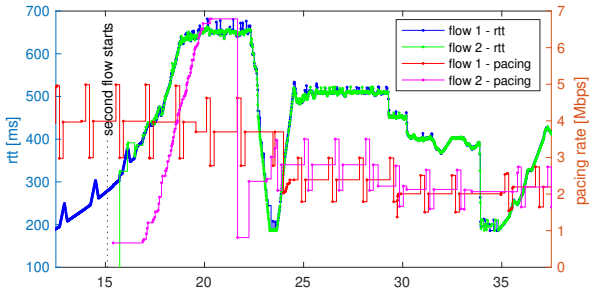


Figure 4: Behavior of two QUIC BBR flows sharing the same bottleneck.

4 HANDLING HTTP/3.0 TRAFFIC

The second contribution of this paper is an improved interface for the handling of HTTP/3.0 traffic. HTTP is the most dominant application-level protocol for web browsing, and has recently expanded to other kinds of traffic (e.g., video and audio streaming). It works on a request and response pattern: in a simple scenario, the HTTP client requests a web page or object and the HTTP server responds with the requested data. HTTP web pages are composed of two kinds of objects: the *main object*, which contains the basic HyperText Markup Language (HTML) code of the page, and the *embedded objects*, which may be images, videos or other contents present in the page. Pages might contain multiple main objects, as they can contain third-party scripts (e.g., social media sharing

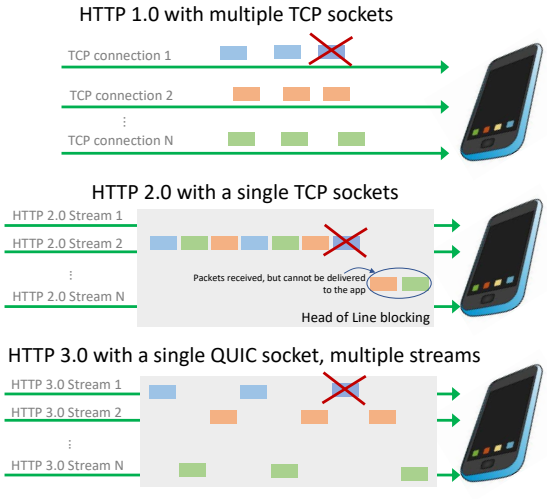


Figure 5: The HOL problem with TCP and HTTP/2, solved by QUIC thanks to the support of multiple independent streams.

buttons). Each main object specifies its embedded objects, which will be received by the client.

When a client requests a web page, all the objects discussed above are downloaded after a single HTTP `GET` request, but they might not need to be displayed at the same time. HTTP/1.1, the first version of the protocol to be widely adopted, did not allow multiplexing: a new TCP connection was opened for each object, each with a separate congestion control state. The competition between multiple flows with the same endpoints was inefficient, and version 2 of the protocol, introduced in 2015 as RFC 7540 [1], uses a single TCP connection to transmit all the objects in a page. This solves the problem in lossless connections, but introduces HOL in lossy ones: since all the packets for multiple objects are multiplexed in the same TCP connection, the in-order delivery requirement means that the loss of a single packet would stop all the other objects from being released to the application. This can have a significant impact in mobile and wireless connections, where measurements show that the page load times of HTTP/2.0 are similar to those of HTTP/1.1 [9].

For this reason, HTTP/3.0 has been proposed as a standard draft [3]: the multiplexing features of HTTP/2.0 are maintained, but the protocol runs over QUIC, sending objects on different streams. Since in-order delivery is only guaranteed between frames belonging to the same streams, the HOL issue is solved. Fig. 5 shows an example of this: while HTTP/1.1 introduces competition between flows and HTTP/2.0 suffers from HOL, HTTP/3.0 can multiplex multiple objects and handle them separately by using different streams.

The ns-3 simulator currently lacks a source model that mimics HTTP/2.0 or HTTP/3.0 streams, and this has prevented, so far, studies of QUIC performance with realistic applications. Therefore, we consider in this paper the implementation of an HTTP model from [23]. The model specifies the traffic parameters, listed in Table 1, for the number and size of the main and embedded objects in a webpage, as well as user based parameters such as reading time. The model is implemented in the `GenericHTTPVariables` class, and the parameters of the model are implemented as attributes. These attributes are initialized to the value given in the paper, but can be re-configured by the user.

4.1 Scheduling policies for QUIC

In a QUIC flow, data from different streams are multiplexed into the same connection, with a common congestion control. Since sender-side buffering may occur, the protocol needs a scheduling policy to decide which frame to send first: the QUIC protocol draft [11] does not specify a stream prioritization technique, aside from mandating that control frames in stream 0 must be sent first, but states that implementations should provide ways to perform it. If we examine an HTTP/3.0 page load, we can see that scheduling is of the utmost importance: the main object and the basic elements of the page should be loaded first to reduce page load times, while complex scripts and large media objects can wait until the rest of the page is already visible. Dependencies should also be considered [19], as objects can not be loaded before their dependencies. Perceived load time is a key Quality of Experience (QoE) metric, and techniques such as the prioritization of areas of the page that a user is likely to look at first can significantly improve it [14]. Using the appropriate scheduler can significantly increase the delay performance [17], but the topic is still relatively new and unexplored.

The previous version of the QUIC module only considered a simple FIFO scheduler, in which frames from all streams, except for stream 0, were put in the same queue. The scheduler was implemented directly in the `QuicSocketTxBuffer` class, which made it hard to modify without affecting the buffer. In the new version of the module, we introduced a `QuicSocketTxScheduler` interface, decoupling the scheduling from the buffer implementation and allowing users to implement their own schedulers transparently. In fact, the actual list of `QuicSocketTxItem` objects to transmit is now part of the scheduler class: the `QuicSocketTxBuffer::Add` method is now a wrapper that creates a `QuicSocketTxItem` from the frame and calls `QuicSocketTxScheduler::Add`. The scheduler object can then use any priority structure to decide which frame to send when the `QuicSocketTxScheduler::GetNewSegment` method is called. Frames from stream 0 are maintained in a separate list in `QuicSocketTxBuffer`, as the QUIC specification states that they should always be sent first independently of the scheduling policy. Another common feature of all scheduling algorithms should be the use of a FIFO policy between frames from the same stream, as the in-order delivery requirements makes other choices inherently less efficient.

The type of scheduler each QUIC socket uses can be chosen by setting the `SchedulingPolicy` attribute of the `QuicSocketBase` class, and the default is the simple FIFO implementation. Besides the default `QuicSocketTxFifoScheduler` class, we provide two other schedulers, which implement the PFIFO and EDF policies. The former, which is implemented in the `QuicSocketTxPfifoScheduler` class, uses a priority queue to prioritize streams with the lowest identifying number: subframes from stream 1 are sent first, and if there are none, the scheduler checks for subframes from stream 2 and so on. The latter is implemented in the `QuicSocketTxEdfScheduler` class, and allows the application to set latency limits for each

Table 1: HTTP Model Parameters, from [23]

Parameter	Mean	Standard Deviation	Maximum	Fitting Curve
Main Object Size	31561 B	49219 B	8 MB	Weibull ($\lambda = 28242.8, k = 0.814944$)
Number of Main Objects	1.2	2.63	212	LogNormal ($\mu = 0.473844, \sigma = 0.688471$)
Embedded Object Size	23915 B	128079 B	8 MB	LogNormal ($\mu = 9.17979, \sigma = 1.24646$)
Number of Embedded Objects	31.93	37.65	1920	Exponential ($\lambda = 0.03132$)
Reading Time	39.7 s	324.92 s	10000 s	LogNormal ($\mu = -0.495204, \sigma = 2.7731$)

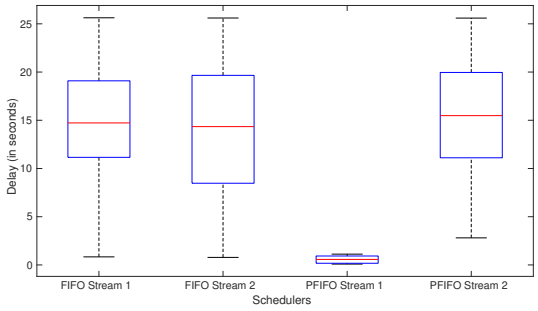


Figure 6: Delay for different streams with the FIFO and PFIFO schedulers.

stream using the `QuicSocketBase::SetLatency` method. In this case, each `QuicSocketTxItem` is assigned a deadline when it is added to the scheduling queue, and the frames with the lowest deadline are served first.

In both cases, retransmissions present an issue: since retransmitted packets may contain frames from different streams, there are two possible ways to handle them. If we choose a "retransmissions first" policy, lost packets are placed in a separate queue with the highest priority (after stream 0 frames), so that they are retransmitted as soon as possible. A second possibility is to fragment each lost packet back into separate frames, subjecting them to the standard scheduling policy. This policy can help avoid the HOL issue in higher-priority streams. The FIFO and EDF schedulers can switch between these two modes using the `RetxFirst` attribute, which is present in both scheduler classes (but not in the FIFO scheduler class, for which the two options are equivalent).

4.2 Performance evaluation

Efficient scheduling among HTTP streams was one of the main reasons for the development of QUIC. As we discussed above, the order in which objects in a webpage are downloaded can significantly affect the load time of the main elements, and thus the QoE of the user. This issue particularly affects low-bandwidth connections, as a lower capacity will mean a more noticeable delay. To show the influence of the queuing policy in QUIC, we choose such a scenario, with a client requesting a page from a server over a connection with a fixed capacity of 1 Mb/s and a minimum RTT of 10 ms. In this case, we use the standard `TcpNewReno` congestion control.

We use the HTTP application above to generate webpages, sending main objects first and their related embedded objects later. We assume that the application does not optimize the order of the objects: naturally, this is not always the case in real applications, but one of the great advantages of HTTP/3.0 is that it gives websites a simple way to allocate priorities. We assume that the application sends main objects and embedded objects below 12 kB, i.e., text and basic elements of the page, on stream 1, while larger embedded objects such as scripts or large images and videos are sent on stream 2.

Fig. 6 shows a boxplot of the load times for 100 random webpages in this scenario, using the model described above. The difference between a FIFO policy, which sends the embedded objects in the random order the application put them in, and a PFIFO policy, which prioritizes stream 1, is remarkable. It is easy to see that the basic elements of the webpage are loaded much faster under PFIFO, with a small additional delay for larger objects.

This simple example is a showcase of the possible advantages of transport layer scheduling among different streams, but the research on the subject is open. In our view, the ns-3 implementation provides a flexible and easy way to test different scheduling strategies in a replicable setting.

5 CONCLUSIONS AND FUTURE WORK

In this work, we present some extensions to the ns-3 QUIC module, which make it more flexible and allow researchers to investigate more complex and realistic scenarios. Our improvements consist of the integration of the BBR code from [12] in the module, which required the implementation of pacing and rate estimation in the module, and of the new scheduling interface, which allows users to extend a generic scheduler interface and specify different policies to prioritize streams.

Future work on the module will involve full alignment with the latest version of the QUIC draft, along with the implementation of multipath capabilities [8] for the protocol, for which the standardization process is also ongoing.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant agreement No. 813999, and from the U.S. Army Research Office through "Towards Intelligent Tactical Ad Hoc Networks" under Grant W911NF1910232.

REFERENCES

[1] Mike Belshe, Martin Thomson, and Roberto Peon. 2015. *Hypertext Transfer Protocol version 2.* RFC 7540. IETF. https://rfc-editor.org/rfc/rfc7540.txt

[2] M. Bishop. 2018. *Hypertext Transfer Protocol (HTTP) over QUIC.* draft-ietf-quic-http-13. IETF. https://tools.ietf.org/html/draft-ietf-quic-http-13

[3] M. Bishop. 2020. *Hypertext Transfer Protocol Version 3 (HTTP/3).* draft-ietf-quic-http-25. IETF. https://tools.ietf.org/html/draft-ietf-quic-http-25

[4] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (Oct. 2016), 20–53.

[5] G. Carlucci, L. De Cicco, and S. Mascolo. 2015. HTTP over UDP: An Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, Salamanca, Spain, 609–614.

[6] Maurizio Casoni and Natale Patriciello. 2016. Next-generation TCP for ns-3 Simulator. *Simulation Modelling Practice and Theory* 66 (2016), 81–93.

[7] Alvise De Biasio, Federico Chiariotti, Michele Polese, Andrea Zanella, and Michele Zorzi. 2019. A QUIC Implementation for ns-3. In *Proceedings of the 2019 Workshop on ns-3*. 1–8.

[8] Quentin De Coninck and Olivier Bonaventure. 2017. Multipath QUIC: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 160–166.

[9] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. 2015. Is HTTP/2 really faster than HTTP/1.1?. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 293–299.

[10] J. Iyengar and I. Swett. 2018. *QUIC Loss Detection and Congestion Control.* draft-ietf-quic-recovery-13. IETF. https://tools.ietf.org/id/draft-ietf-quic-recovery-13.txt

[11] J. Iyengar and M. Thomson. 2018. *QUIC: A UDP-Based Multiplexed and Secure Transport.* draft-ietf-quic-transport-13. IETF. https://tools.ietf.org/id/draft-ietf-quic-transport-13

[12] Vivek Jain, Viyom Mittal, and Mohit P. Tahiliani. 2018. Design and Implementation of TCP BBR in Ns-3. In *10th Workshop on ns-3 (WNS3 '18)*. ACM, Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3199902.3199911

[13] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. ACM, London, United Kingdom, 290–303.

[14] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R Das. 2017. Improving user perceived page load times using gaze. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 545–559.

[15] L. Kleinrock. 1979. Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications. In *Conference Record, International Conference on Communications*. Boston, Massachusetts, 43.1.1–43.1.10.

[16] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, Los Angeles, CA, USA, 183–196.

[17] Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. 2019. Resource Multiplexing and Prioritization in HTTP/2 over TCP versus HTTP/3 over QUIC. (Sept. 2019). https://h3.edm.uhasselt.be/files/ResourceMultiplexing_H2andH3_Marx2020.pdf

[18] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. 2001. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 287–297.

[19] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster page loads using fine-grained dependency tracking. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*.

[20] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. 2017. De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives. *IEEE Communications Surveys & Tutorials* 19, 1 (First quarter 2017), 619–639.

[21] Natale Patriciello. 2017. A SACK-based Conservative Loss Recovery Algorithm for Ns-3 TCP: A Linux-inspired Proposal. In *Proceedings of the Workshop on Ns-3 (WNS3 '17)*. ACM, Porto, Portugal, 1–8.

[22] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. 2018. A Survey on Recent Advances in Transport Layer Protocols. *Submitted to IEEE Communications Surveys and Tutorials* (2018). https://arxiv.org/abs/1810.03884

[23] Rastin Pries, Zsolt Magyari, and Phuoc Tran-Gia. 2012. An HTTP web traffic model based on the top one million visited web pages. In *Proceedings of the 8th Euro-NF Conference on Next Generation Internet NGI 2012*. IEEE, 133–139.

[24] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. 2018. A First Look at QUIC in the Wild. In *Passive and Active Measurement*, Robert Beverly, Georgios Smaragdakis, and Anja Feldmann (Eds.). Springer International Publishing, 255–268.

[25] M. Thomson and S. Turner. 2018. *Using TLS to Secure QUIC.* draft-ietf-quic-tls-13. IETF. https://tools.ietf.org/html/draft-ietf-quic-tls-13