HTTP over UDP: an Experimental Investigation of QUIC

Gaetano Carlucci
Politecnico di Bari &
Quavlive, Italy
gaetano.carlucci@poliba.it

Luca De Cicco
Politecnico di Bari &
Quavlive, Italy
l.decicco@poliba.it

Saverio Mascolo
Politecnico di Bari &
Quavlive, Italy
mascolo@poliba.it

ABSTRACT

This paper investigates "Quick UDP Internet Connections" (QUIC), which was proposed by Google in 2012 as a reliable protocol on top of UDP in order to reduce Web Page retrieval time. We first check, through experiments, if QUIC can be safely deployed in the Internet and then we evaluate the Web page load time in comparison with SPDY and HTTP. We have found that QUIC reduces the overall page retrieval time with respect to HTTP in case of a channel without induced random losses and outperforms SPDY in the case of a lossy channel. The FEC module, when enabled, worsens the performance of QUIC.

Keywords

TCP, UDP, HTTP, SPDY, QUIC, congestion control

1. INTRODUCTION

HTTP is the most used application level protocol over the Internet [1]. Recent studies argue that HTTP will represent the narrow waist of the future Internet [12]. During the past decade its adoption has experienced a tremendous growth [10] mainly fueled by the wide diffusion of HTTP-based infrastructure such as Content Distribution Networks (CDNs), proxies, caches, and other middle-boxes. This growth is driven by video content delivered using HTTP, which is today the first source of Internet traffic. In fact, video over HTTP is the mainstream choice employed by all major video distribution platforms including the ones based on the recent MPEG-DASH[1], HLS standards, and the VoD systems such as YouTube and NetFlix [5].

Despite its widespread use, some inefficiencies of HTTP are hindering the development of a faster Internet. The goal of improving HTTP/1.1 has attracted researchers [12] and industries, among which Google that has recently proposed SPDY [4] within the IETF working group HTTPbis[2]. SPDY is meant to overcome the following inefficiencies of HTTP: 1) a HTTP client can only fetch one resource at a time, even if this issue can be in part mitigated with pipelining; 2) a client-server *pull-based* communication model is used; if a server knows that a client needs a resource, there is no mechanism to push the content to the client; 3) it redundantly sends several headers on the same channel. HTTP/1.1 Web browsers attempt to address these issues by opening multiple concurrent HTTP connections, even though this approach is discouraged in the HTTP/1.1 RFC which suggests a limit of two concurrent connections for each client-server pair.

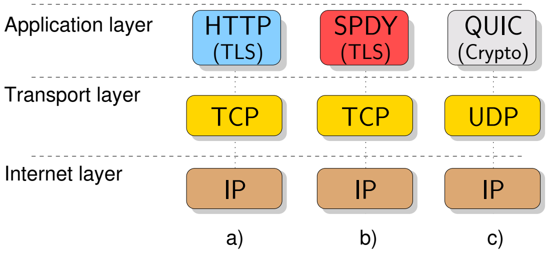


Figure 1: HTTP, SPDY and QUIC stack

To address these issues SPDY introduces the following features: 1) it multiplexes concurrent HTTP requests on a single TCP socket; 2) it compresses HTTP headers; 3) it enables the server to push data to the client whenever possible; 4) it allows prioritization among parallel requests.

SPDY has already been deployed world-wide in the Chrome Web browser and it shows encouraging improvements with respect to HTTP/1.1 [2] in the case of wired networks, but it is still not clear whether it is able to improve performances in the case of cellular networks [7]. The fact that SPDY employs only one TCP socket to deliver all the Web resources to the client has the advantage of decreasing the port usage at the server, but has a key drawback: by multiplexing streams over a single TCP connection put SPDY in disadvantage when compared to several HTTP/1.1 connections each with a separate congestion window. In fact, a single lost packet in an underlying TCP connection triggers a congestion window decrease for all of the multiplexed SPDY streams, whereas in the case of N parallel HTTP/1.1 connections it would affect only one out of the N parallel connections. Another disadvantage of SPDY is that an out-of-order packet delivery for TCP induces *head of line blocking* for all the SPDY streams multiplexed on that TCP connection. Moreover, SPDY connection startup latency depends on TCP handshake which requires one RTT and in the case SSL/TLS is employed up to three RTT.

All the inefficiencies mentioned above are due to the use

[1] http://dashif.org/mpeg-dash/
[2] http://datatracker.ietf.org/doc/draft-ietf-httpbis-http2/

ACM SAC'15, April 13 - 17 2015, Salamanca, Spain
Copyright 2015 ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2695664.2695706 .

of TCP as transport protocol. This has motivated Akamai and Google to propose new reliable protocols on top of UDP. Akamai has proposed a Hybrid HTTP and UDP content delivery protocol that is employed in its CDN [11]. Google is aiming at reducing TCP handshake time [13] and, more recently in [8], at improving TCP loss recovery mechanism.

Unfortunately, the improvements above mentioned are not implemented in the default version of TCP. Motivated by this evidence Google has proposed QUIC [14] over UDP to replace HTTP over TCP. QUIC has been already deployed by Google in their servers, such as the ones powering YouTube, and can be activated in the Web client Chrome browser that runs on billions of desktop and mobile devices. This puts Google in the position of driving a switch of a sizable amount of traffic from HTTP over TCP to QUIC over UDP.

The contribution of this paper is twofold: 1) we provide an experimental investigation of QUIC to check whether its deployment can be harmful for the network; 2) we assess the performance of QUIC compared to SPDY and HTTP/1.1 in terms of Page Load Time reduction. To the best of our knowledge, this is the first experimental investigation of QUIC.

2. THE QUIC PROTOCOL

Quick UDP Internet Connections (QUIC) is an experimental protocol proposed by Google and designed to provide security and reliability along with reduced connection and transport latency. Google has already deployed QUIC protocol in their servers and has a client implementation in the Chrome web browsers.

Figure 1 shows the main architectural differences between HTTP over TCP (Figure 1(a)), SPDY over TCP (Figure 1 (b)) and QUIC over UDP (Figure 1(c)). The main idea behind QUIC is to use the UDP to overcome the SPDY inefficiencies as discussed in Section 1 along with an ad-hoc designed encryption protocol named “QUIC Crypto” which provides a transport security service similar to TLS.

Since the UDP is unreliable and does not implement congestion control, QUIC implements retransmissions and congestion control at the application layer. Furthermore, QUIC leverages most part of the SPDY design choices to inherit its benefits.

A description of the QUIC protocol based on the official documentation [14] and on the analysis of the Chromium code base is following below[3].

2.1 Multiplexing

QUIC multiples several QUIC streams over the same UDP connection. Figure 2(a) shows that with HTTP/1.1 a client can only fetch one resource at a time (even if they are pipelined) whereas with QUIC (Figure 2(b)) a client can send multiple HTTP requests and receive multiple responses over the same UDP socket (in the case of Figure 2 three resources are requested). HTTP/1.1 web browsers attempt to minimize the impact of *head of line blocking* (HOL) by opening multiple concurrent HTTP connections, typically up to six.

The QUIC multiplexing feature has been inherited from SPDY and provides: 1) prioritization among QUIC streams; 2) traffic bundling over the same UDP connection; 3) compression of HTTP headers over the same connection.

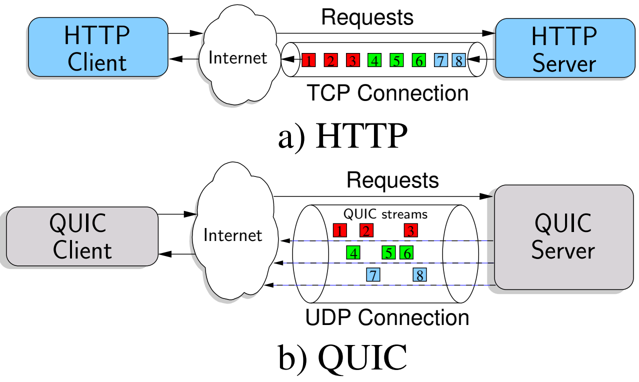


Figure 2: Multiplexing

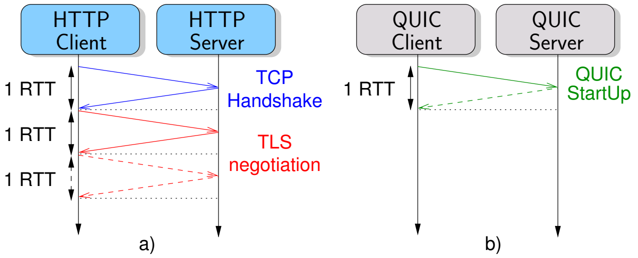


Figure 3: Startup latency

By using the UDP, QUIC is able to eliminate the HOL issue affecting the SPDY multiplexed streams; for instance in the case of Figure 2(a) if packet n.1 is lost and all the other packets are received, all packets must wait until the lost packet is retransmitted before TCP can deliver the data to the application (TCP does not allow out-of-order delivery); on the other hand UDP can deliver all the received packet to the application without waiting for the retransmission.

2.2 Connection startup latency and security

Figure 3(a) shows the time required to setup a TCP connection: it takes one RTT for the handshake and at least one extra RTT or two in the case of an encrypted connection over TLS. When QUIC is used (Figure 3(b)), the time taken to set up a connection is at most one RTT; in the case the client has already talked to the server before, the startup latency takes zero RTT even in case of an encrypted connection (QUIC uses its own encryption algorithm named QUIC-Crypto). QUIC-Crypto decrypts packets independently: this avoids serialized decoding dependency which would damage QUIC ability to provide out-of-order delivery to reduce the HOL.

2.3 Forward Error Correction

Another interesting feature of QUIC is the Forward Error Correction (FEC) module that copes with packet losses. The benefit of the FEC module could be particular effective in further reducing HOL over a single QUIC stream by promptly recovering a lost packet, especially in the case of high RTT where retransmissions could considerably affect the HOL latency. It works as follows: one FEC packetis computed at the end of a series of packets as the XORsum of the packets payload; these packets compose a*“FEC Group”*

[3] https://code.google.com/p/chromium/

(F_G). For example F_G has size 4, if three data packets plus a FEC packet are sent: in this way at most one packet loss within a *FEC group* can be recovered. It is clear that the *FEC group* size plays a remarkable role: a small F_G implies high redundancy at cost of bandwidth, whereas a large F_G implies low redundancy at low bandwidth cost. The decision on F_G size is still considered an open issue; more details are provided in Section 4.1.

2.4 Pluggable Congestion Control

QUIC has been designed to support two congestion control algorithms: 1) the first is an implementation of the TCP CUBIC [9]; 2) the second is a pacing-based congestion control algorithm that computes the application sending rate based on an estimate of the *relative forward delay* defined as the difference between the inter-arrival time of two consecutive data packets at the receiver and the inter-departure time of the same packets at the sender; this approach is similar to the one proposed in the WebRTC framework [6]. At the beginning of the connection the end-hosts can negotiate the algorithm to employ. At the time of writing QUIC implements only TCP CUBIC. We remind that the congestion window W in CUBIC increases as proposed in [9], i.e. $W(t) = C(t{-}K)^3 + W_{max}$, where C is a parameter called a scaling factor, t is the elapsed time since the last window reduction, and K is the time period required to increase W_{max} when no loss is detected. K is given by $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$ where β is a constant multiplication decrease factor applied for window reduction at the time of a loss event as follows:

$$W(t) \leftarrow W(t^*)(1-\beta\ ) \qquad (1)$$

where $W(t^*)$ is the value reached by $W(t)$ when a packet loss is detected at t^* and the new W_{max} becomes $W(t^*)$. When a packet loss is detected $W(t)$ is reduced according to (1). The main difference with respect to CUBIC implementation in TCP, is the value of the decrease factor: in TCP β_{TCP} is equal to 0.3 whereas in QUIC $\beta_{QUIC} = \beta_{TCP}/n$, where $n = 2$. In practice, $\beta_{QUIC} = 0.15$ is equivalent to β_{TCP} when 2 concurrent TCP CUBIC flows compete for the available bandwidth. An initial value of 10 for the congestion window is set in both QUIC and TCP CUBIC.

2.5 Connection Identifier

A QUIC connection is uniquely identified by a *CID* (Connection IDentifier) at the application layer and not by the pairs of IP addresses and port number. The first advantage is that, since CIDs are not based on IP addresses, any handover between two networks can be transparently handled by QUIC without needing to re-establish the connection. Moreover, the CID is useful in the case of NAT unbinding, when to restore a connection, a new pair of IP addresses is typically required. Finally with CID, QUIC can provide native support to multi-path; for example a mobile device could use all the network connections for the same CID. It is important to notice that, at the time of this writing, multi-path is still not implemented in Chrome.

3. TESTBED AND METRICS

In this Section we describe the testbed and the metrics employed to carry out the experimental evaluation of QUIC.

We employ the testbed configurations shown in Figure 4. The testbed in Figure 4(a) has been used to compare the dynamics of the CUBIC congestion control algorithm implemented within QUIC with the one implemented in TCP. The testbed in Figure 4(b) has been employed to measure the Web page load time when HTTP/1.1, SPDY or QUIC is employed.

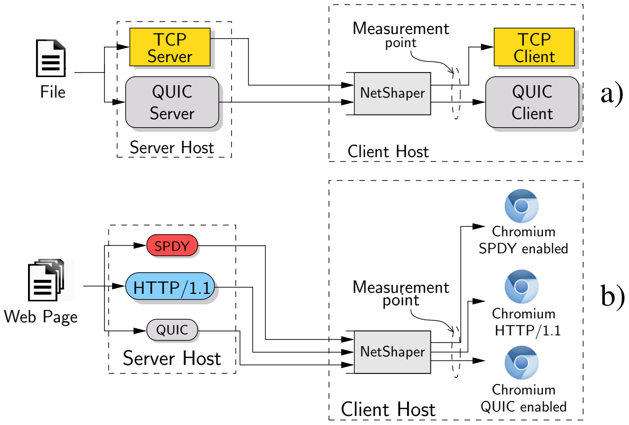


Figure 4: Testbed

The testbed consists of two DELL Precision T1650 64-bit machines (server/client hosts) running Debian Jessy with a Linux kernel 3.14.2.

In particular Figure 4(a) shows the testbed employed to compare the congestion control dynamics. A dummy QUIC server and a dummy QUIC client can be found in the Chromium browser code-base[4] which we have compiled and used in the testbed as a stand-alone server and client. The testbed employs the QUIC version 21 that is the latest available implementation at the time of this writing. We have modified the QUIC source code in order to log the relevant variables for this investigation. The TCP server is an iperf-like application which uses TCP CUBIC [9] and logs the congestion window, the slow-start threshold and the instantaneous RTT.

Figure 4(b) shows the testbed employed to compare the Web page load time when QUIC, SPDY over SSL/TLS or HTTP/1.1 over SSL/TLS is employed in the Chromium browser. The HTTP/1.1 over SSL/TLS server host employs Apache/2.4.10 with TLS 1.2. For SPDY over SSL/TLS, we have employed nghttp2[5] which supports SPDY 4 that is the Google implementation of HTTP/2.0; the page encryption is still provided by TLS 1.2. It is worth to notice that QUIC server always encrypts the data. Encryption is also mandatory for SPDY. On the client host we have employed three different configurations of the Chromium M39: one that has QUIC enabled and can fetch the Web page from the QUIC server, one that has SPDY 4 enabled and fetches the Web page from the SPDY server and one that has QUIC and SPDY disabled which fetches the Web page from the HTTP/1.1 Web server.

At the receiver we have used a tool called `NetShaper` that we have developed to perform bandwidth shaping and to allow propagation delays to be set. This tool uses the `nfqueue` library provided by `Netfilter`[6] in order to capture and redirect the traffic arriving at the client host to a user space tail-drop queue, where traffic shaping and measurement are performed.

[4] https://code.google.com/p/chromium/
[5] http://nghttp2.org/
[6] http://www.netfilter.org/

Table 1: Scenarios and parameters employed in the experimental evaluation

Dynamics Analysis	**Parameters**
S1.Impact of link capacity (b), induced random losses (L) and FEC (ON/OFF)	L:$\{0, 1, 2\}$% - b:$\{3, 6, 10\}$Mbps
S2.QUIC Vs TCP with varying buffer sizes (Q)	Q:$\{13, 30, 60\}$kByte
Page Load Time	**Parameters**
S3.Web page size: small (305kB), medium (690kB), large (2.1MB)	L:$\{0, 1\}$% - b:$\{3, 10\}$Mbps

Metrics.

We have considered the following metrics: 1) **Goodput** G, measured as the average network received rate (without considering retransmissions and forward error correction); 2) **Channel Utilization** U**,** measured as r/b, where r is the average received rate and b the link capacity; 3) **Loss Ratio** l**,** defined as (*byte lost/byte sent*) measured by the `NetShaper` tool; 4) **Page load time** P**,** defined as the time taken by the browser to download and process all the objects associated with a Web page (Document Object Module). To measure the page load time, we employ the Chromium browser developer console which measures the time elapsed since the user asks for the Web page until the page is fully loaded (when the load event is fired).

4. RESULTS

In this section we present the experimental results obtained by employing the two testbeds shown in Figure 4. The first goal is to compare the performance of the CUBIC congestion control algorithm implemented within QUIC with the one implemented in TCP. It is particularly important to check that the implementation of QUIC congestion control over UDP does not generate traffic that is harmful for the network stability. The second goal is to check to what extent QUIC is able to improve user performance in terms of Page Load Time. Towards this end we compare the performance of QUIC with those of SPDY and HTTP/1.1 (over TLS).

We consider the latest available version of QUIC that at the time of writing is v.21. In this version CUBIC is the only congestion control algorithm implemented and the FEC module is turned off by default; the FEC action has been activated only in one scenario (see Section 4.1) to evaluate the FEC influence on the considered metrics.

Table 1 summarizes the scenarios and parameters considered in the experimental evaluation. For each combination of the parameters shown in Table 1 we have run ten experiments and evaluated the metrics by averaging over all the results.

4.1 QUIC flow dynamics (S1)

In this Section we compare the QUIC and TCP flow dynamics in isolation. We evaluate the impact of b and L variation (see Table 1 - S1) when FEC is enabled or disabled on the channel utilization U and loss ratio l. We set the base RTT RTT_m equal to 50ms and the bottleneck size Q to the bandwidth-delay product [3] with Tail Drop policy.

Figures 5(a), (b) and (c) show the results obtained for each value of L: each bar group represents the evaluated metric obtained for a certain link capacity; within each group the first bar shows the metrics for QUIC when FEC is disabled, the second bar shows the case of QUIC when FEC is enabled, and the third one shows the case of TCP.

Let us focus on the case $L = 0\%$ (Figure 5(a)): in every case the channel utilization is about 99%. When FEC is enabled, it takes roughly 33% of the channel utilization. Moreover QUIC introduces higher losses than TCP especially when FEC is enabled. This is due to the fact that a recovered loss by the FEC module does not trigger the congestion control and makes the algorithm more aggressive. At $L = 1\%$ (Figure 5(b)) and at $L = 2\%$ (Figure 5(c)) the QUIC channel utilization is not significantly affected by the introduced random losses; in particular, when FEC is enabled, it still takes about 33% of the channel utilization regardless of the parameters variation of Table 1. This means that, on average, the FEC Group $\bar{F}_G$ is equal to 3. On the other hand, as expected, the TCP goodput is significantly reduced when random losses are introduced and the reduction increases with the link capacity.

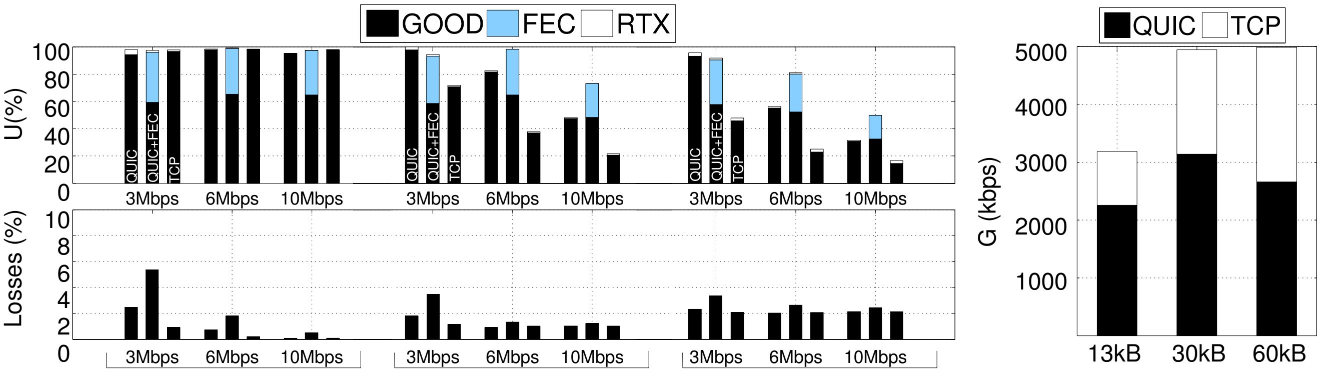
Figure 6(a) shows an experimental result in the case of $L = 0\%$, $b = 3$Mbps and FEC disabled. It is interesting to notice that the QUIC *cwnd* exhibits smaller oscillations than the ones presented by TCP *cwnd* due to the fact that QUIC employs a smaller decrease factor β (see Section 2.4); this explains the more aggressive nature of QUIC flows and the fact that it shows higher losses in Figure 5(a). Figure 6(b) shows that, when $L = 2\%$, the QUIC rate is not significantly affected by the introduced random losses.

Figure 6(c) shows the case of $L = 0\%$, $b = 3$Mbps with FEC enabled. FEC uses about 33% of the available bandwidth, which reduces the QUIC goodput. Moreover, despite the FEC action we measure that only a negligible amount of packets has been recovered when $L = 0\%$: this means that every time a congestion loss occurs it is likely that more than one packet is consecutively lost and these packets cannot be recovered because they belong to the same FEC Group. On the other hand, in the case of $L = 1\%$ or $L = 2\%$ we see that the number of packets which are recovered by FEC is higher but still at the expense of a remarkable goodput reduction.

4.2 QUIC and TCP CUBIC friendliness (S2)

In this section we evaluate the impact of the bottleneck buffer size Q on the goodput G when TCP and QUIC share the same bottleneck. We consider a link capacity $b = 5$Mbps and $RTT_m = 50$ms; three different values of the bottleneck buffer have been considered $Q = \{13, 30, 60\}$kB with Tail Drop policy. In particular, for $Q = 16$kB the network is under-buffered, for $Q = 30$kB the buffer is equal to the bandwidth-delay product and for $Q = 60$kB the network can be considered over-buffered [3]. Figure 7(a) shows the case of $Q = 30$kB : QUIC prevails over TCP, since it gets roughly 1.5 times the link capacity. Figure 7 (b) shows the case of $Q = 60$kB where TCP and QUIC fairly share the link capacity.

Figure 5(d) summarizes the results for each value of Q. It shows that for $Q = 13$kB, QUIC achieves a goodput that is roughly three times larger than that obtained by TCP. However, it is worth mentioning that full channel utilization is not achieved due to the fact that the network is under-buffered [3]. In the case of $Q = 30$kB QUIC gets roughly 1.5 times the goodput obtained by TCP. This is mainly due to the fact that QUIC has a smaller decrease factor β (see



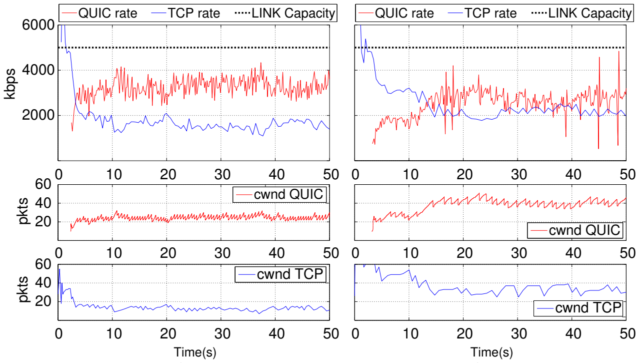
(a) L=0% (b) L=1% (c) L=2%

Impact of link capacity b, induced random losses L and FEC

(d) TCP Vs QUIC

Impact of buffer size

Figure 5: Impact of the parameter variation on the metrics



a) Q=30kB b) Q=60kB

Figure 7: One TCP flow with one concurrent QUIC flow over a 5 Mbps bottleneck

Section 2.4). In the case of $Q = 60$kB TCP and QUIC fairly share the link capacity since both the flows can enter the MaxProbing phase [9] when the network is over-buffered.

4.3 Page Load Time comparison (S3)

In this section we measure the Page Load Time P defined in Section 3 using the Chromium browser when QUIC, SPDY over TLS or HTTP/1.1 over TLS is employed to transport a small, a medium or a large Web page.

It is quite challenging to compare HTTP, SPDY and QUIC Page Load Time P due to the fact that it depends on many factors external to the protocols themselves [15], including Web page characteristics and browser processing (i.e., JavaScript evaluation and HTML parsing). In order to enforce reproducibility and eliminate external dependencies, we employ the controlled testbed shown in Figure 4(b) and a simple Web page containing only jpeg images without any JavaScript code and css file. Moreover, on the client side we employ the same version of the Chromium browser M39 which supports SPDY, QUIC and HTTP. The *small Web page* contains 18 jpeg images large 2.6kB and 3 larger jpeg images of 86.5kB. The *medium Web page* contains 40 jpeg images large 2.6kB and 7 jpeg images large 86.5kB. Finally, the *large Web page* contains 200 jpeg images large 2.6kB and 17 jpeg images large 86.5kB. Every page has an index.html that links the images. We consider two different combinations of the link capacity, i.e. $b = \{3, 10\}$Mbps, and two values for the random loss percentages, namely $L = \{0, 2\}\%$; the base RTT has been set to $RTT_m = 50$ms and the bottleneck has a Tail Drop buffer Q equal to the bandwidth-delay product. These parameters have been chosen similarly to [15].

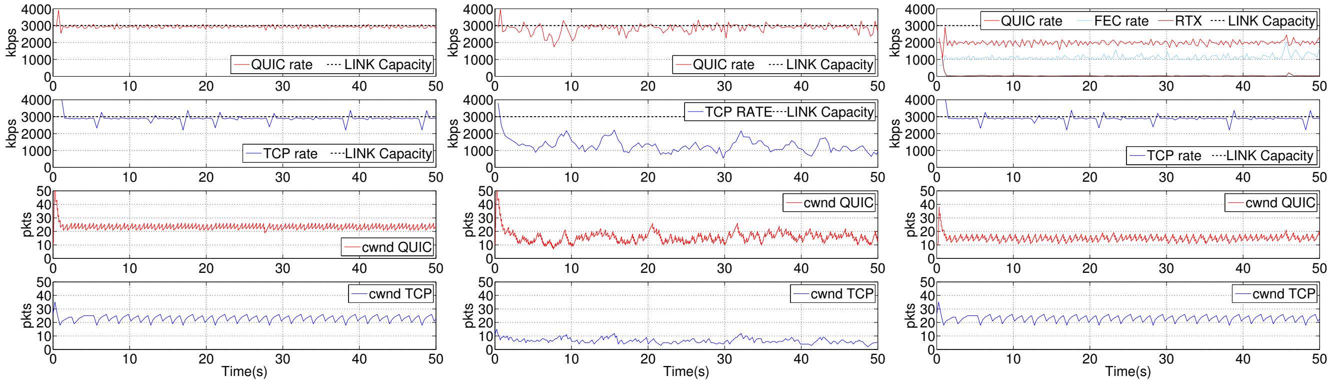
We have observed that when Chromium fetches the Web pages from the Apache Web server using HTTP/1.1 it opens maximum 6 TCP parallel sockets, so that more than one image can be downloaded simultaneously from the server. In the case of QUIC only one UDP socket is opened and six QUIC streams are multiplexed over this connection. Finally, in the case of SPDY 4 only one TCP socket is opened and, after the index.html is parsed, all the images are requested simultaneously; the requests are all multiplexed on top of the single TCP connection.

Figure 8 shows the percentage page load time improvement $I_m(\%)$ obtained by QUIC or SPDY with respect to HTTP/1.1, which is considered the baseline of the comparison:

$$I_m = \frac{P_{HTTP} - P_{QUIC/SPDY}}{P_{HTTP}} \cdot 100 \qquad (2)$$

Figure 8(a) shows that, when the channel is without induced losses both SPDY and QUIC outperform HTTP. In particular in the case of link capacity $b = 3$Mbps, SPDY provides a higher improvement with respect to QUIC due to the higher level of multiplexing. When $b = 10Mbps$ in the case of small or medium size page, QUIC shows a higher improvement due to its zero RTT start-up latency; the impact of the zero RTT start-up latency becomes less remarkable in the case of large Web page, since in this case the transport time prevails. Moreover, in the case of large Web page and when $b = 10Mbps$, the benefit of the higher level of multiplexing used by SPDY is more effective than the one employed by QUIC which multiplexes six streams only (version 21).

In the scenario of Figure 8(b) where 2% of random losses are introduced QUIC provides a lower improvement than in the case of channel without induced random losses. In the case of *medium* and *large* Web page at 10Mbps even QUIC increases the Page Load Time. Even if it may look counter-intuitive based on the dynamics comparison shown in Figure 6(b), it should be noted that, in the case of HTTP/1.1, the browser opens 6 parallel TCP connections and the effect of the random losses will be distributed among the 6 TCP flows, which has less impact than in the case of a 6 streams multiplexed over a single UDP connection (see Figure 2(b)). This is also confirmed by observing that in the case of SPDY,



(a) $L = 0\%$,$b = 3$Mbps,FEC OFF

(b) $L = 2\%$,$b = 3$Mbps,FEC OFF

(c) $L = 0\%$,$b = 3$Mbps,FEC ON

Figure 6: Rate and cwnd dynamics results in 3 different experiments

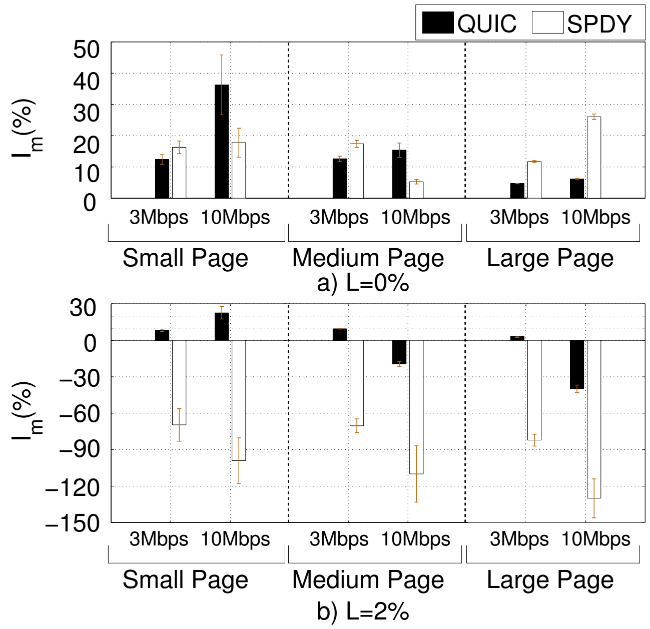


Figure 8: Page load Time Improvement with respect to HTTP/1.1

which employs only one TCP socket, the improvement with respect to HTTP is always negative and in the worst case, when $b = 10$Mbps, it is on average under -120%; this is due to the fact that, in the case of a larger link capacity, the congestion window reduction due to the random losses have remarkable influence on the goodput (see Figures 5(a)-(c)). The fact that using QUIC is beneficial compared to SPDY in the presence of induced random losses is due to the different β factor (see Section 2.4).

5. CONCLUSIONS

In this paper we have carried out an experimental investigation of QUIC, a protocol recently proposed by Google to transport HTTP traffic over UDP. We have compared the performance of the latest QUIC implementation with the standard HTTP/1.1 under different network conditions. We have found that QUIC shows higher goodput with respect to TCP CUBIC in the case of under-buffered networks and also in the presence of random losses but at the cost of an increased packet loss ratio. Moreover, when QUIC is employed to transport a Web page, it reduces the overall page retrieval time with respect to HTTP/1.1 in case of a channel without random losses and outperforms SPDY in the case of a lossy channel. The FEC module, when enabled, worsens the performance of QUIC.

6. ACKNOWLEDGEMENTS

This work has been partially supported by the Italian Ministry of Education, Universities and Research (MIUR) through the MAIVISTO project (PAC02L1 00061). Gaetano Carlucci has also been partially supported by “Scuola Interpolitecnica di Dottorato”. Authors would also like to thank Ali Begen, Josh Gahm, Vittorio Palmisano, and Sergio Zaza for their help in setting up the testbed and for their helpful discussions.

7. REFERENCES

[1] Global Internet Phenomena Report, 2014. https://www.sandvine.com/trends/global-internet-phenomena/.

[2] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[3] G. Appenzeller et al. Sizing router buffers. In *Proc. of ACM SIGCOMM '04*, Portland, Oregon, USA, 2004.

[4] M. Belshe and R. Peon. Spdy protocol. *IETF Draft*, 2012.

[5] Cisco. Cisco Visual Networking Index:Forecast and Methodology 2013-2018. *White Paper*, June 2014.

[6] L. De Cicco et al. Understanding the dynamic behaviour of the google congestion control for rtcweb. In *in Proc. of Packet Video Workshop 2013*, San Jose, CA, USA.

[7] J. Erman et al. Towards a spdy'ier mobile web? In *Proc. of ACM CoNEXT*, pages 303–314, 2013.

[8] T. Flach et al. Reducing web latency: The virtue of gentle aggression. In *Proc. of ACM SIGCOMM '13*, Hong Kong.

[9] S. Ha et al. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS '08*, 42(5):64–74, 2008.

[10] C. Labovitz et al. Internet inter-domain traffic. In *Proc. of ACM SIGCOMM '11*, pages 75–86, Toronto, Canada, 2011.

[11] M. Ponec and A. Alness. Hybrid http and udp content delivery, Aug. 23 2013. US Patent App. 13/974,087.

[12] L. Popa et al. Http as the narrow waist of the future internet. In *Proc. of ACM SIGCOMM Workshop on HotNets*, pages 6:1–6:6, Monterey, California, 2010.

[13] S. Radhakrishnan et al. Tcp fast open. In *in Proc. of CoNEXT '11*, pages 21:1–21:12, Tokyo, Japan, 2011.

[14] J. Roskind. Multiplexed Stream Transport over UDP, 2013.

[15] X. S. Wang and et. al. How speedy is spdy. In *Proc. of the 11th USENIX*, pages 387–399, San Diego, CA, USA, 2014.