Evaluating QUIC's Performance Against Performance Enhancing Proxy over Satellite Link

John Border, Bhavit Shah, Chi-Jiun Su, and Rob Torres
Hughes Network Systems
Germantown, USA
{john.border, bhavit.shah, chi-jiun.su, rob.torres}@hughes.com

***Abstract*—Geosynchronous orbit satellite broadband serves a crucial role in bridging the digital divide by connecting under-served and unserved areas where terrestrial infrastructure is infeasible. Most of the TCP implementations are not optimized for high bandwidth delay product (BDP) satellite links. TCP Performance Enhancing Proxies (PEP) have alleviated degraded TCP performance in satellite broadband. The new transport protocol, QUIC, encrypts not only payloads but also most of the transport layer header. Consequently, it is no longer feasible to accelerate transport layer performance over a high BDP satellite link. The paper performs a measurement study to compare the performance of the QUIC protocol to that of the current accelerated transport layer over a GEO satellite link. The results showed that QUIC achieves only 20% of the throughput provided by the transport layer acceleration optimized for a satellite link. It also shows the need for a complementary end-to-end loss recovery mechanism for QUIC.**

Keywords—QUIC Transport Protocol, Transmission Control Protocol, Performance Enhancing Proxy, Hypertext Transfer Protocol, Satellite Broadband

I. Introduction

QUIC is a new transport layer protocol, originally developed by Google [1] and currently being standardized at the IETF [2]. QUIC aims to provide security using built-in end-to-end encryption features resulting in faster connection establishment and to allow for rapid deployment by implementing the protocol in user space. QUIC by Google and QUIC by IETF can be distinguished by using the terms gQUIC for Google QUIC and QUIC for IETF QUIC.

The Transmission Control Protocol or TCP [3] is the most widely used transport layer protocol across the Internet. TCP is a sliding window protocol where a sender can send a specific number of packets, the window, without having to receive an acknowledgement. As acknowledgements are received for the packets at the front on the window, the window slides allowing the sender to keep sending. If acknowledgements are not received before reaching the end of the window, the sender must wait for an acknowledgement before resuming and repeating the process. A sender can ensure optimum throughput by transmitting enough data to completely fill the link and keep the link busy before it must stop to receive an acknowledgement from the receiver. To do this, the window must be large enough to allow time for acknowledgements to be received before transmission stalls.

Geostationary Equatorial Orbit or GEO satellite-based communication systems are characterized by long delays, resulting in a large bandwidth delay product (BDP) path. It also possible to have a relatively high bandwidth (for ex: 300 Mbps), which makes the BDP even higher. Such link characteristics impact performance of transport layer protocols like TCP. A sender must send a significantly large amount of data to achieve optimum throughput, thus requiring an unusually large sender and receiver window size. This is not possible without making any modifications to native TCP as default TCP configuration and behavior assumes a much smaller delay. Furthermore, such long delays also impact error recovery since any lost packets must be recovered end to end. To improve end-to-end performance over such links, performance-enhancing proxies or PEPs [4] are used to implement a modified TCP connection such as Split TCP. Split TCP terminates a connection from one end system and initiates a corresponding connection to the other end system. This allows different connection characteristics for different parts of the end-to-end path, while allowing the end systems to continue using native TCP. Using such a modified TCP significantly improves the throughput achieved over such high-BDP links.

Over a high-BDP link, native QUIC also suffers the same issue as native TCP i.e. poor performance. But contrary to native TCP, QUIC natively encrypts the connection metadata as well which makes it impossible for middle-boxes to provide performance enhancement to a QUIC connection by splitting QUIC in the same way as TCP. Hence QUIC suffers from relatively poor performance (compared to Split TCP) on high latency links.

On top of transport layer protocols, Hypertext Transfer Protocol or HTTP [5] is the most commonly used application layer protocol. As of now, HTTP has two mainstream variants: HTTP/1.1 and HTTP/2 [6]. A new version, HTTP/3 [7], to run over QUIC, is under development at the IETF. HTTP/3 will be used with iQUIC. HTTP/2 is being used with gQUIC. HTTP/2 builds on top of the HTTP/1.1 architecture and aims to help HTTP performance by resolving issues like head-of-line blocking amongst and is usually encrypted end-to-end. Native TCP allows both HTTP/1.1 and HTTP/2, while gQUIC allows only HTTP/2 connections. An important point to note here is that, usually, PEP throughput performance improvement of HTTP/2 flows over TCP is less significant than for HTTP/1.1 flows as HTTP/2 employs application layer flow control at both stream level and connection level. When there is just one flow, HTTP/2's stream flow control window limits the throughput over high BDP link.

In this paper, we try to illustrate the throughput disparities between HTTP/2 over native gQUIC and HTTP/1.1 over Split TCP over satellite links for large file downloads.

II. Related Work

There has been some prior work done to evaluate the performance of HTTP variants [8] as well as QUIC over Satellite [9]. One such study also compares the performance over multiple satellite operators [10]. There's also an IETF draft [11] which identifies the various characteristics of a satellite link that impact QUIC and discusses mechanisms to improve performance over such a link.

Annex to ISBN 978-3-903176-28-7© 2020 IFIP

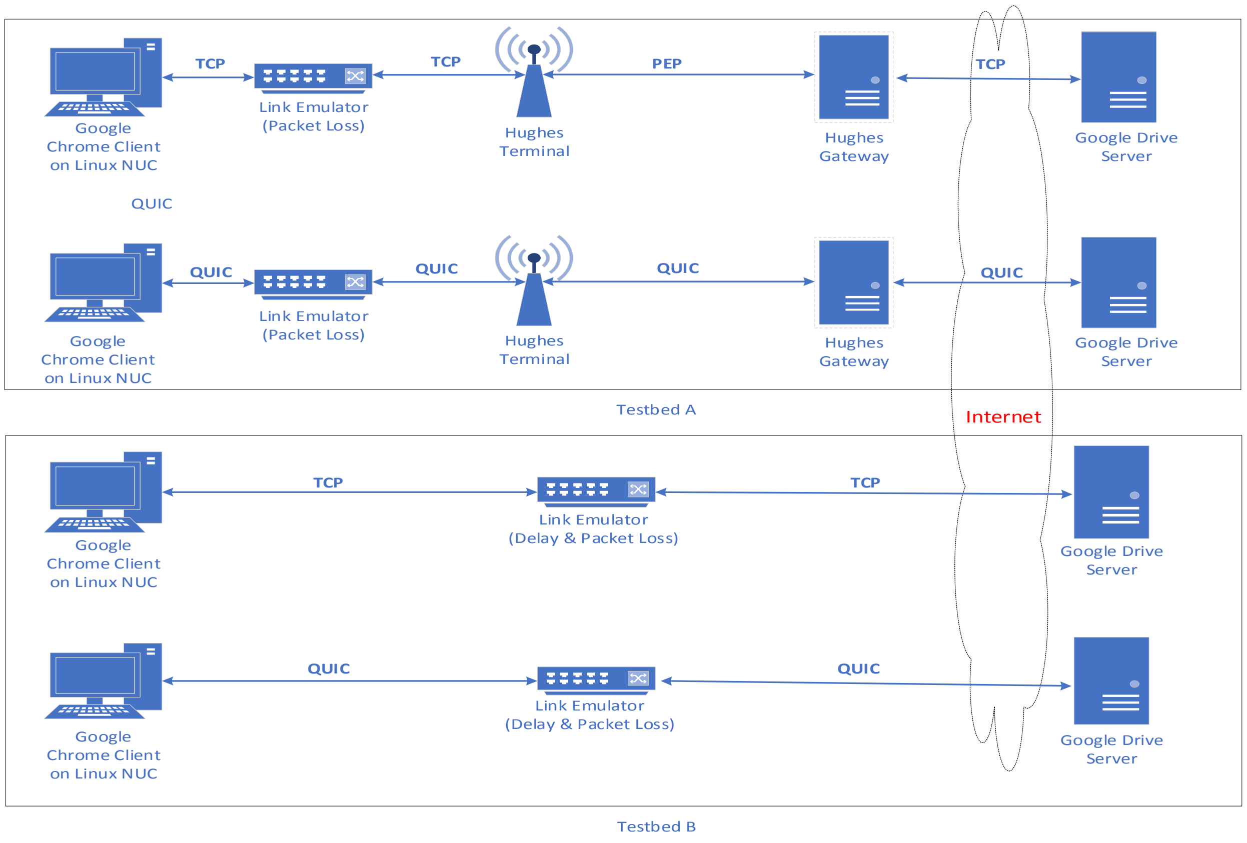
TCP
Google Chrome Client on Linux NUC
Link Emulator (Packet Loss)
TCP
Hughes Terminal
PEP
Hughes Gateway
TCP
Google Drive Server
QUIC
QUIC
Google Chrome Client on Linux NUC
Link Emulator (Packet Loss)
QUIC
Hughes Terminal
QUIC
Hughes Gateway
QUIC
Google Drive Server
Testbed A
Internet
TCP
Google Chrome Client on Linux NUC
Link Emulator (Delay & Packet Loss)
TCP
Google Drive Server
QUIC
Google Chrome Client on Linux NUC
Link Emulator (Delay & Packet Loss)
QUIC
Google Drive Server
Testbed B

Fig. 1. Testbed setup for the QUIC performance evaluation using Satellite as well as Direct link

III. Setup

A. Testbed

For this experiment, we use Google Drive as the server and the latest Google Chrome (v77.x when tested) on Intel Next Unit of Computing (NUC) running Linux distribution Ubuntu v16.04 LTS (with default TCP/UDP settings).

The testing is done over two controlled testbeds as shown in Fig.1. Testbed A is connected to the Internet through a Hughes terminal HT2000 and gateway. The measured RTT between the terminal and gateway is 600ms. This RTT is typical for a clean satellite link. (Latency can go as high as 1500ms or more during congestion, especially for lower priority traffic but this was not tested). Testbed B is connected to the Internet over a 1Gbps direct link via a link emulator emulating satellite like delay (600ms RTT). In both cases, the same link emulator is also used to introduce packet loss on the link between the client device and the satellite terminal. The packet loss rates (PLR) used are 0%, 0.1% and 1%, based on expected PLR on a Wi-Fi link between client device and satellite terminal. The error rate on links between terminal and gateway as well as gateway and the server are typically low.

An important distinction between Testbeds A and B is the type of TCP connection that takes place. In Testbed A, the TCP connection is spoofed due to the presence of Hughes' PEP [12] which is a split TCP connection (with no packet compression), whereas in Testbed B, the TCP connection remains unspoofed, meaning it uses native TCP. There is no difference in the type of gQUIC connection that takes place in either testbeds (that we know of).

We repeatedly download the same 1GB file from the Google Drive server over TCP and gQUIC, record the download time for each download and calculate the average download throughput to gauge performance. As the server is outside our control, we do this testing back to back to mitigate the impact of any other factors like the TCP congestion algorithm employed by the server and the time of day related traffic volumes at the server. As the rest of the setup is within our control, we also make sure to run these tests when there is no other traffic running. As the testbeds stay free of other traffic, there is almost no additional processing delay at any of the middleboxes.

B. Google Chrome Behavior

Google Chrome, by default, tries to connect to Google Drive server using gQUIC. If a gQUIC connection fails, Google Chrome falls back to TCP. But Google Chrome also allows the user to specify the type of connection that the browser makes with the server by using its feature called "flags". A flag can allow users to choose the application layer protocol (HTTP/1.1 or HTTP/2) and/or the transport layer protocol (TCP/gQUIC) for their connection. The set of flags is provided in Table 1. We control the behavior of Google Chrome by running it with the required combination of flags. The negotiated gQUIC version was observed to be Q046.

Table 1 Controlling Google Chrome Behavior

Connection Type	**Flags per type of connection**	
	HTTP/2 Flag	***gQUIC Flag***
HTTP/1.1 over TCP	--disable-http2	--disable-quic
HTTP/1.1 over gQUIC	Not Supported	Not Supported
HTTP/2 over TCP	--enable-http2	--disable-quic
HTTP/2 over gQUIC	--enable-http2	--enable-quic

C. *Test Automation Tool*

We leverage a Node library called Puppeteer (v1.18.0) [13] to control Google Chrome to repeatedly download large files from Google Drive and automate the testing. A Node.js script automatically runs a new instance of Google Chrome with specified flags and browses to the specified download location within Google Drive to download a specific 1GB file. A file watcher is also setup within this script which captures the start time of the file download, constantly looks for changes to that file in the directory and captures the end time as soon as the file size reaches the specified file size of 1GB. Using the captured information, it calculates the download throughput per run. This script is typically run at least 100 times for each test.

IV. Test Results

A. *Performance over Satellite Link*

As mentioned before, Testbed A has the Linux NUC connected to the Internet over a satellite link with a measured 600ms RTT via the Hughes components. In this setup, the 1GB file download from Google Drive over TCP-PEP/HTTP1.1 can achieve a maximum throughput of 250 Mbps via a "wget" like application. For our testing, Google Chrome establishes either a TCP-PEP or a gQUIC connection with Google Drive to download the same file. The performance across various packet loss rates is as follows:

1) *Throughput (Mbps) with 0% PLR*

As seen in Table 2 and Fig. 1, with no induced packet loss, the TCP-PEP connection using HTTP1.1 typically exceeds 200 Mbps. With HTTP2, web browser flow control limits the throughput even with TCP-PEP to 40 Mbps. HTTP2 on top of gQUIC is slightly slower than this.

TABLE 2 THROUGHPUT OVER SATELLITE LINK WITH 0% PLR

Throughput (Mbps)	Connection Type		
	TCP-PEP/HTTP1.1	***TCP-PEP/HTTP2***	***gQUIC/HTTP2***
Average	212.22	41.98	35.90
Max	231.53	44.60	38.76
Min	131.34	40.42	26.74

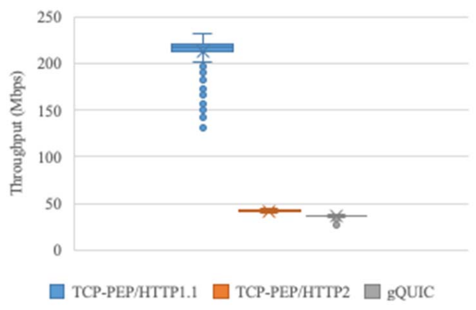

Fig. 2. Throughput over Satellite Link with 0% PLR

2) *Throughput (Mbps) with 0.1% PLR*

With a small (0.1%) induced packet loss rate, the TCP-PEP connection using HTTP1.1 does slow down slightly but local recovery from lost packets keeps the speed fairly high at 179 Mbps. Since the TCP-PEP also provides local recovery from lost packets for HTTP2, HTTP2 throughput stays near 40 Mbps. HTTP2 on top of gQUIC must recover from lost packets end to end. This results in about a 35% drop in throughput as seen in Table 3 and Fig. 3.

TABLE 3 THROUGHPUT OVER SATELLITE LINK WITH 0.1% PLR

Throughput (Mbps)	Connection Type		
	TCP-PEP/HTTP1.1	***TCP-PEP/HTTP2***	***gQUIC/HTTP2***
Average	170.83	43.42	23.72
Max	180.46	44.35	27.60
Min	129.17	29.99	14.15

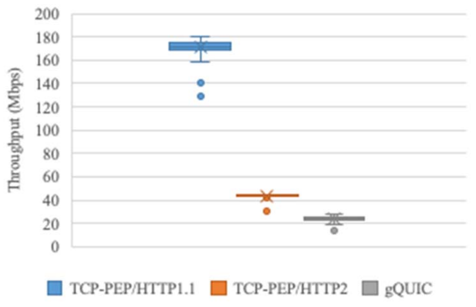

Fig. 3. Throughput over Satellite Link with 0.1% PLR

3) *Throughput (Mbps) with 1% PLR*

With a larger (1.0%) induced packet loss rate, the TCP-PEP connection using HTTP1.1 slows even more but again local recovery from lost packets keeps the speed around 120 Mbps. With the TCP-PEP local recovery from lost packets, HTTP2 throughput actually stays near 40 Mbps. The speed limit imposed by HTTP2 is able to absorb the lost in TCP speed. HTTP2 on top of gQUIC slows down even more to about 50% of the error-free throughput as seen in Table 4 and Fig. 4.

TABLE 4 THROUGHPUT OVER SATELLITE LINK WITH 1% PLR

Throughput (Mbps)	Connection Type		
	TCP-PEP/HTTP1.1	***TCP-PEP/HTTP2***	***gQUIC/HTTP2***
Average	118.25	41.13	17.23
Max	136.35	43.36	19.74
Min	86.85	28.02	9.62

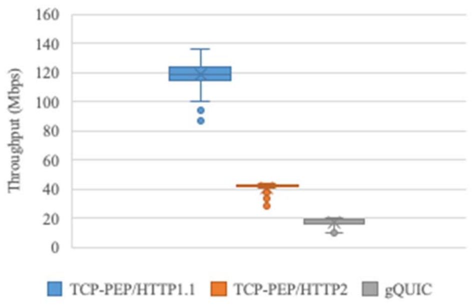

Fig. 4. Throughput over Satellite Link with 1% PLR

B. *Performance over Direct Link*

As mentioned before, Testbed B has the Linux NUC connected to the Internet over a direct link via a link emulator emulating a satellite-like 600ms RTT. In this setup, Google Chrome establishes either a native TCP or a gQUIC connection with Google Drive to download the 1GB file. The performance across various packet loss rates is as follows:

1) Throughput (Mbps) with 0% PLR

As seen in Table 5 and Fig. 4, with no induced packet loss, the TCP connection throughput is basically the same with HTTP1.1 and HTTP2. HTTP2 on top of gQUIC is slightly slower than this.

TABLE 5 THROUGHPUT OVER DIRECT LINK WITH 0% PLR

Throughput	**Connection Type**		
(Mbps)	***TCP/HTTP1.1***	***TCP/HTTP2***	***gQUIC/HTTP2***
Average	37.25	37.61	34.86
Max	38.08	38.35	35.47
Min	36.37	36.29	32.95

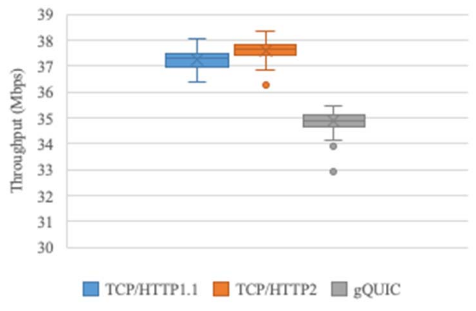

Fig. 5. Throughput over Direct Link with 0% PLR

2) Throughput (Mbps) with 0.1% PLR

With a small (0.1%) induced packet loss rate, since TCP must now recover from lost packets end to end, the TCP connection throughput is basically the same with HTTP1.1 and HTTP2 but drops to almost half. HTTP2 on top of gQUIC handles the packet loss much better with only a 20% reduction. In the presence of errors, gQUIC performance is better than end to end TCP performance as seen in Table 6 and Fig. 6.

TABLE 6 THROUGHPUT OVER DIRECT LINK WITH 0.1% PLR

Throughput	**Connection Type**		
(Mbps)	***TCP/HTTP1.1***	***TCP/HTTP2***	***gQUIC/HTTP2***
Average	19.27	19.76	27.25
Max	20.56	20.88	27.94
Min	15.74	16.65	26.40

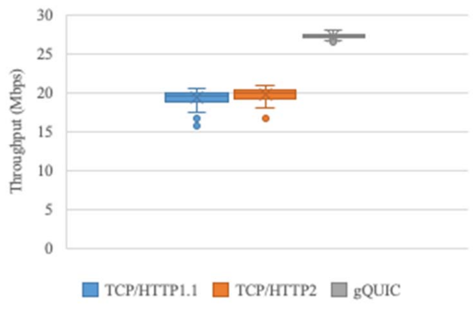

Fig. 6. Throughput over Direct Link with 0.1% PLR

3) Throughput (Mbps) with 1% PLR

With a larger (1.0%) induced packet loss rate, TCP throughput drops even more. The same is also true for gQUIC. But again, in the presence of packet loss, gQUIC performance is better than end to end TCP performance as seen in Table 7 and Fig. 7.

TABLE 7 THROUGHPUT OVER DIRECT LINK WITH 1% PLR

Throughput	**Connection Type**		
(Mbps)	***TCP/HTTP1.1***	***TCP/HTTP2***	***gQUIC/HTTP2***
Average	12.80	12.98	23.36
Max	13.48	13.87	23.87
Min	12.06	12.22	22.38

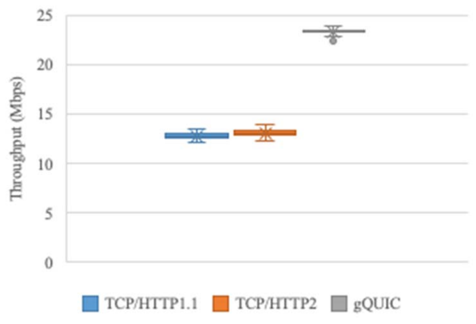

Fig. 7. Throughput over Direct Link with 1% PLR

V. ANALYSIS

For a high bandwidth delay product path like GEO satellite-based communications, native TCP is known to perform poorly. From the testing in Testbed B, with satellite-like emulated delay, native gQUIC performs like native TCP and even outperforms it if any packet loss is introduced as shown in Fig. 8. In absence of a performance enhancing proxy, native gQUIC would seem like a better alternative to native TCP.

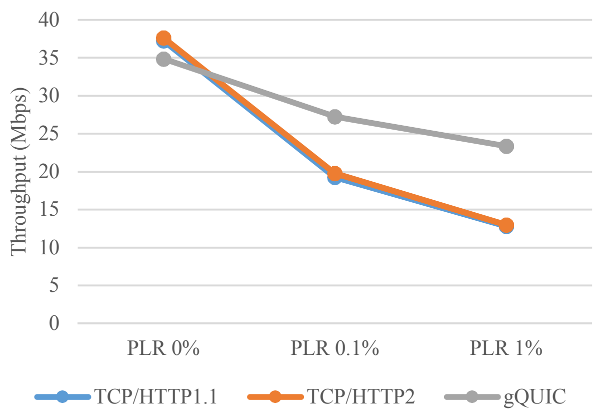

Fig. 8. Performance across various PLR in Testbed B

But as mentioned before, such high BDP paths use TCP-PEP instead of native TCP to improve the throughput performance. Based on the testing done using Testbed A, we confirm that TCP-PEP performs significantly better than native TCP. Additionally, in Fig. 9, we show that TCP-PEP outperforms native gQUIC as well even in presence of packet loss regardless of HTTP version. Thus, there seems to be a need to improve the performance of gQUIC to use it as a transport protocol over high BDP paths.

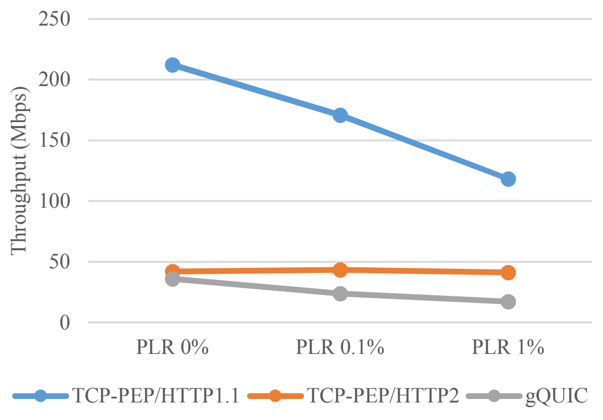

Fig. 9. Performance across various PLR in Testbed A

Looking at application layer performance in Fig. 10 and Fig. 11, we observe that HTTP/1.1 and HTTP/2 performance deteriorates with increasing packet loss rate. As expected, we see that HTTP/1.1 flows greatly benefit from TCP-PEP even in presence of packet loss. As HTTP/2 flows seem to self-limit their throughput, they perform only slightly better with PEP than native TCP in absence of packet loss, but interestingly their performance doesn't seem to deteriorate with increasing packet loss rates over TCP-PEP, compared to native gQUIC as seen in Fig. 9. In other words, the HTTP/2 flows seem to benefit as well from TCP-PEP in presence of packet loss. This may indicate a need for a complementary end to end loss recovery mechanism for higher RTT with gQUIC like packet level FEC scheme [14][15].

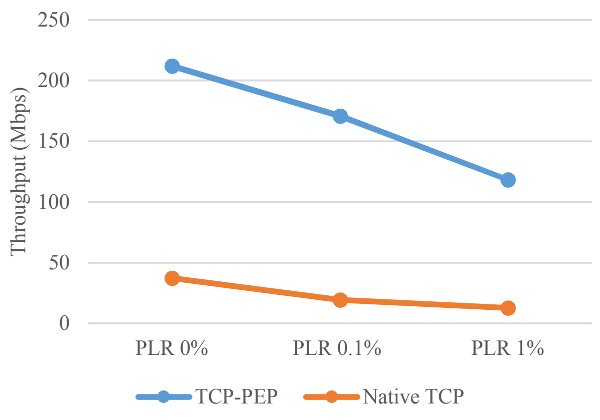

Fig. 10. Performance of HTTP/1.1 over TCP

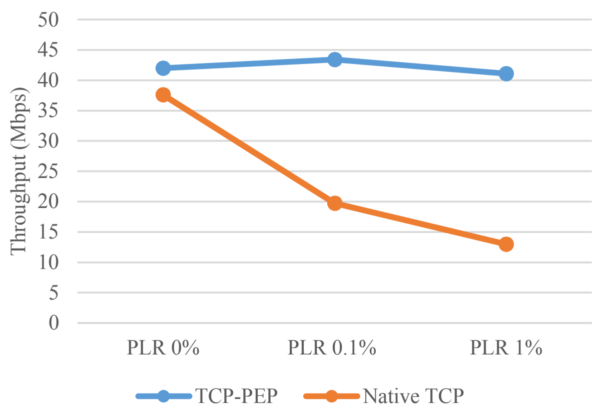

Fig. 11. Performance of HTTP/2 over TCP

VI. CONCLUSION

In this paper, we presented and evaluated the performance of new transport and application layer protocols over a high BDP GEO satellite link. We leverage a public server and a well-known user client to mimic user experience. As expected, TCP-PEP outperforms gQUIC over such links with or without packet loss. We conclude that using gQUIC natively over such high BDP paths leads to an unsatisfactory user experience. Potential approaches to improve performance like increased initial and maximum congestion windows as well as introducing FEC schemes have been discussed in [11]. TCP-PEP also seems to help HTTP/2 flows although the performance lags behind HTTP/1.1. Based on the test results, we see the need to assess gQUIC's lower performance over satellite link as compared to a direct link under similar conditions and better understand the HTTP/2 behavior over TCP-PEP.

VII. FUTURE WORK

Given the quick evolution of QUIC within IETF, we acknowledge the limited relevance of the gQUIC tests. To expand the scope of this paper, we plan to similarly evaluate QUIC as well. To complete the evaluation of the QUIC variants, we also intend to test with higher RTTs because under certain conditions, the satellite delays can go above 1500ms due to MAC layer queuing delay. As the server is the biggest aspect of our testing that was beyond our control, we also plan on introducing our own QUIC server to the testbed in future.

REFERENCES

[1] A. Langley et al., “The QUIC Transport Protocol: Design and Internet-scale Deployment,” In *Proc. of the Conf of the ACM Special Interest Group on Data Communication,* August 2017, pp. 183–196.

[2] J. Iyengar, and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-27, February 2020, Work in Progress.

[3] J. Postel, “Transmission Control Protocol,” RFC 793, September 1981.

[4] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, “Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations,” RFC 3135, June 2001.

[5] R. Fielding, and J. Reschke, “Hypertest Transfer Protocol (HTTP/1.1): Semantics and Content,” RFC 7231, June 2014.

[6] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” RFC 7540, May 2015.

[7] M. Bishop, “Hypertext Transfer Protocol Version 3 (HTTP/3),” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-27, February 2020, Work in Progress.

[8] R. Secchi, A. C. Mohideen, and G. Fairhurst, “Performance Analysis of Next Generation Web Access via Satellite,” International Journal of Satellite Communications and Networking, vol. 36, pp. 29–43, December 2016.

[9] L. Thomas, E. Dubois, N. Kuhn, and E. Lochin, "Google QUIC performance over a public SATCOM access", International Journal of Satellite Communications and Networking , 2019.

[10] J. Deutschmann, K. Hielscher and R. German, "Satellite Internet Performance Measurements," 2019 International Conference on Networked Systems (NetSys), Munich, Germany, 2019, pp. 1-4, doi: 10.1109/NetSys.2019.

[11] N. Kuhn, G. Fairhurst, J. Border, and E. Stephan, “QUIC for SATCOM,” Internet Engineering Task Force, Internet-Draft draft-kuhn-quic-4-sat-04, March 2020, Work in Progress.

[12] Hughes Network Systems, LLC., “JUPITERTM System Bandwidth Efficiency,” https://www.hughes.com/signage/collateral-library/jupitertm-system-bandwidth-efficiency, Germantown, USA, November 2015.

[13] Puppeteer: https://developers.google.com/web/tools/puppeteer

[14] Swett, I., Montpetit, M., Roca, V., and F. Michel, "Coding for QUIC", draft-swett-nwcrg-coding-for-quic-03 (work in progress), July 2019.

[15] Roca, V., Michel, F., Swett, I., and M. Montpetit, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for QUIC", draft-roca-nwcrg-rlc-fec-scheme-for-quic-02 (work in progress), November 2019.