Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Un estudio en profundidad de los protocolos QUIC y HTTP/3 y su impacto en servicios web modernos

Tesis de Licenciatura en Ciencias de la Computación

Sebastian Uriel Sujarchuk

Director: Dr. Claudio Enrique Righetti

Buenos Aires, 2023

Resumen

Con la reciente estandarización de los protocolos QUIC y HTTP/3 comienza el largo proceso para comenzar a adoptar estas nuevas tecnologías. Por un lado, QUIC introduce una serie de mejoras y cambios para adaptarse a la Internet actual, y se posiciona como potencial reemplazo de TCP en lo que respecta a protocolos de transporte para comunicaciones confiables. A su vez, HTTP/3 surge como el primer protocolo construido sobre QUIC, permitiéndole aprovechar sus nueva capacidades. En la actualidad, todavía no existe una gran cantidad de servicios que implementen QUIC y HTTP/3. No obstante, estos ya fueron adoptados por algunos de los sitios más grandes de Internet. En esta tesis presentaremos un estudio de los protocolos QUIC y HTTP/3 con el objetivo de entender cómo funcionan y qué cambios introducen respecto a sus predecesores. Adicionalmente, proponemos un análisis del impacto de estos protocolos en el rendimiento de algunos servicios web, comparándolos con versiones anteriores de HTTP (que funcionan sobre el protocolo TCP). Para realizar estas comparaciones se utilizaron tres implementaciones distintas de QUIC y HTTP/3 desarrolladas en distintos lenguajes. Con estas se observó la latencia y se la comparó con una implementación de HTTP versión 1 y 2 del lenguaje correspondiente. Con el fin de evaluar el impacto en producción de estos protocolos, las mediciones fueron realizadas con servidores que se encuentran productivos actualmente. En base a la experimentación presentada, pudimos comprobar una leve mejora de HTTP/3 con respecto a HTTP/2 en ciertos casos. No obstante, estos resultados se pueden ver afectados por la implementación utilizada. Para sitios más optimizados, el rendimiento demostró ser muy similar tanto para HTTP/2 como para HTTP/3, con rendimientos menos variados para el primero. Adicionalmente, también observamos que la presencia de CDNs y caches puede hacer que estas mejoras sean pequeñas, o incluso no apreciables. Esperamos que estos resultados puedan ser de utilidad a la hora de ponderar si es necesario implementar QUIC y HTTP/3 en el corto plazo o si es más razonable esperar, pues en la actualidad, la implementación de estos protocolos requiere un esfuerzo adicional por parte de los desarrolladores.

Palabras Clave: QUIC, HTTP/3, HTTP, Protocolos de Transporte, TCP.

Abstract

With the recent standardization of the QUIC and HTTP/3 protocols, it begins the long process of adopting these new technologies. On one hand, QUIC introduces a series of improvements better suited to the Internet of the present, and positions itself as a potential candidate to replace TCP as the de facto transport protocol for reliable communications. Furthermore, HTTP/3 emerges as the first protocol built on top of QUIC. Currently, there are only a few services implementing QUIC and HTTP/3. Nevertheless, these were already adopted by some of the Internet's biggest sites. During this thesis we will present a study of the protocols QUIC and HTTP/3 with the goal of understanding how they work and what changes are introduced compared to their predecessors. Additionally, we propose an analysis on the impact of these protocols in the performance of productive web services, comparing them against previous versions of HTTP (that run on top of the TCP protocol.). To perform these comparisons, three implementations of QUIC and HTTP/3 in different languages were used. With these, we observed the resulting latency and compared it against an implementation of the HTTP versions 1 and 2 of the corresponding language. With the intent of understanding the impact on production of these protocols, we used productive services as our test subjects during this experiment. The results observed from the experimentation show a slight improvement of HTTP/3 in comparison to its predecessors, but only in certain cases. Nevertheless, the results can be heavily affected by the choice of implementation. For more optimized sites, the performance for HTTP/2 and HTTP/3 was very similar, with more consistent results on the former. Furthermore, we also noticed that the presence of CDNs and caches can make these improvements smaller, or even negligible. We expect that these results can be of use at the time of evaluating whether it is necessary to add support for QUIC and HTTP/3 on the short term, or if it is more reasonable to wait, given that at the time of writing this thesis, the implementation of these protocols is not transparent and requires an additional effort from developers.

Keywords: QUIC, HTTP/3, HTTP, Transport Protocols, TCP.

Dedicado a Maaian, Carina, David, y Federico.

Índice general

Capítulo 1

Introducción

Durante 2021 y 2022 la *IETF* terminó la estandarización de los protocolos QUIC [9] y HTTP/3[18] con la aprobación de sus RFCs correspondientes (RFC9000 y RFC9114). Esto marca un hito importante en el avance de Internet. HTTP, siendo el protocolo de mayor adopción, ha funcionado exclusivamente sobre TCP desde sus orígenes en los años 90. El cambio a un nuevo protocolo como QUIC rompe con esta continuidad y permite la introducción de cambios y mejoras que antes no eran posibles o eran muy costosas de implementar.

Durante esta tesis se podrá notar que QUIC y HTTP/3 suelen ser mencionados en conjunto pues están muy relacionados en sus motivaciones y orígenes. QUIC surge inicialmente como un protocolo para mejorar el rendimiento de HTTP/2. Dadas las mejoras introducidas en QUIC, el grupo de trabajo HTTP decide adoptarlo como protocolo de transporte. Este cambio disruptivo resulta en la creación de HTTP/3.

A pesar de haber sido recientemente estandarizados, el trabajo en ambos protocolos comienza en el año 2012, desde sus orígenes en Google, y ya venían siendo adoptados en sus versiones previas. Las aplicaciones de ambos protocolos eran bastante específicas y llevadas a cabo por empresas de grandes recursos y tráfico que necesitaban continuar mejorando la calidad de su servicio, o *Early Adopters*[1] que buscaban probar los avances introducidos.

Con las primeras versiones finalmente aprobadas y estandarizadas, se espera que llegue una nueva ola de grupos interesados en implementar QUIC y HTTP/3. Ante la implementación de un nuevo protocolo es importante preguntarse si se obtendrán beneficios tangibles.

1.1. Contribuciones

En esta tesis realizamos una revisión y análisis exhaustivo de los protocolos QUIC y HTTP/3 con el objetivo de entender sus motivaciones, cómo funcionan y las innovaciones que introducen con respecto a TCP y las versiones anteriores de HTTP. Este estudio contribuye al conocimiento actual sobre los protocolos de red y su impacto en la evolución de los servicios web, y proporciona una base sólida para futuras investigaciones en este

[1] Los Early adopters hacen referencia al conjunto de individuos y organizaciones que adopta una tecnología de forma temprana, previo a su uso masivo. Normalmente se los considera líderes de opinión en estas nuevas innovaciones. Este concepto surge del ciclo de adopción tecnológica propuesto por Geoffrey Moore en su trabajo Crossing the Chasm. [35]

campo. Adicionalmente, presentamos una comparación del impacto en la latencia de la implementación de HTTP/3 (basada en QUIC) en contraposición con sus versiones anteriores (basadas en TCP). Esto se llevará a cabo experimentando sobre varios sitios web que implementan estos protocolos. En base al análisis y los resultados de la experimentación, esperamos que este trabajo facilite el proceso de toma de decisiones a la hora de implementar QUIC y HTTP/3 en un entorno productivo.

Durante esta tesis trabajamos con tres implementaciones de QUIC y HTTP/3 distintas. Por limitaciones de tiempo y de practicidad, queda por fuera de este trabajo el uso de otras implementaciones. Por estas mismas razones también limitamos la metodología de experimentación específicamente a la latencia punta a punta para los servicios con los trabajamos y no utilizamos otras métricas para medir el impacto de los protocolos.

1.2. Trabajos Relacionados

Para el desarrollo de esta tesis, se llevó a cabo un análisis sobre diversos trabajos relacionados con la implementación y el rendimiento de QUIC y HTTP/3 en distintos contextos. Marx et al. [10] realizaron una comparación de múltiples implementaciones para investigar y evaluar su rendimiento a través de pruebas sintéticas en entornos controlados. Los resultados de este trabajo enfatizan la importancia de realizar ajustes específicos para optimizar el rendimiento de cada implementación.

Por otro lado, Trevisan et al. [11] y Yu et al. [12] contribuyeron investigando el estado de los despliegues de QUIC en producción. El primero trabajó con un conjunto de conjuntos de datos públicamente disponibles que miden la adopción y el rendimiento basados en datos de navegación reales. El segundo desarrolló un sistema de medición utilizando implementaciones reales de QUIC y HTTP/3, introduciendo condiciones sintéticas en la red. Estos trabajos se llevaron a cabo antes de la estandarización de QUIC y HTTP/3, y en sus respectivas conclusiones se reconoció la importancia de revisar los resultados una vez que ambos protocolos estuvieran estandarizados.

En esta tesis, se busca utilizar una metodología de experimentación basada en la generación de tráfico hacia servidores productivos [11][12], empleando la latencia como medida principal de comparación. Se trabajará únicamente con implementaciones que utilicen las versiones estandarizadas de QUIC y HTTP/3 en lugar de versiones previas. Estas implementaciones utilizan sus configuraciones por defecto, a diferencia del enfoque de Marx et al. [10] Esto es importante para comprender su impacto directo en producción.

Por último, este trabajo también incluye un análisis exhaustivo de los protocolos QUIC y HTTP/3 con el objetivo de comprender en mayor detalle su funcionamiento, sus motivaciones, su contexto histórico y las diferencias principales con sus predecesores.

1.3. Estructura de la Tesis

Esta tesis se encuentra estructurada en seis capítulos.

En el primer capítulo, se comienza presentando una introducción al problema y su contexto. También se revisan las contribuciones y limitaciones propuestas por este trabajo.

En el segundo capítulo se procede a realizar una revisión exhaustiva del protocolo QUIC, comenzando por sus motivaciones y orígenes en Google. Luego, se examinan los diversos aspectos y mecanismos que componen el protocolo. A su vez, se contrastan estos

con los mecanismos ya existentes en TCP y TLS. Después, se presentan algunas de las extensiones existentes para el protocolo QUIC y las problemáticas que estas resuelven.

En el tercer capítulo se analiza el funcionamiento del protocolo HTTP/3. Sobre este se realiza una revisión sobre su funcionamiento, cómo se mapea sobre QUIC, y que funcionalidades nuevas introduce con respecto a sus predecesores.

En el cuarto capítulo se pasa a explicar la metodología de experimentación utilizada, cuales son sus objetivos y qué herramientas fueron utilizadas para lograrlos.

En el quinto capítulo se presentan los resultados obtenidos sobre las varias mediciones realizadas. Para cada una de estas, se realiza una apreciación sobre los valores observados y su posible significado.

Finalmente, en el sexto capítulo, se exponen las conclusiones sobre todo el trabajo realizado y los posibles trabajos futuros.

Capítulo 2

El protocolo QUIC

QUIC es un protocolo de transporte de red confiable y seguro originalmente desarrollado por Google en el año 2012 para aumentar el rendimiento del tráfico HTTP/2 sumado a despliegues rápidos y evolución continua. QUIC funciona como un reemplazo al stack tradicional compuesto por HTTP, TLS y TCP. [1][2] El uso de UDP como protocolo subyacente facilita el soporte del protocolo para transitar por cualquier capa de red intermedia. [1]

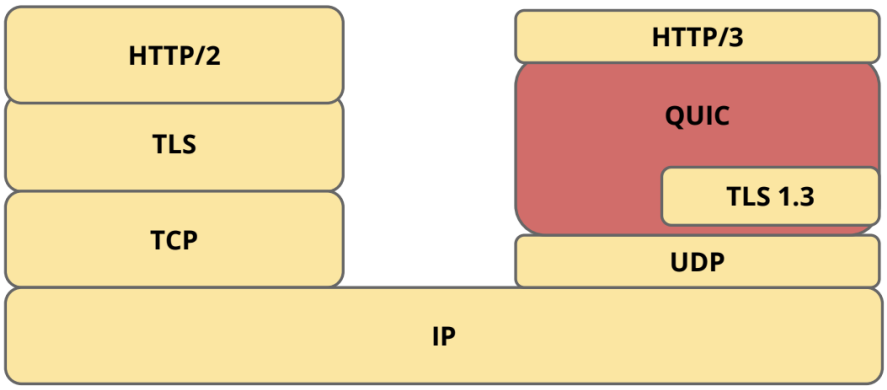


Figura 2.1: Comparación entre stacks TCP/HTTP2 y QUIC/HTTP3 [24]

QUIC introduce una variedad de cambios, aprendiendo de las lecciones de su predecesores TCP y TLS[1]. Algunos de los más significativos son:

- Implementación en *User Space* con el objetivo de acelerar la velocidad de actualización y desarrollo.
- Seguridad incluida en el protocolo. QUIC no necesita implementar TLS encima para establecer una conexión segura, sino que integra TLS 1.3 dentro del protocolo. Esto

[1]El protocolo TLS, (previamente conocido como SSL) significa Transport Layer Security. Este está compuesto por un conjunto de protocolos criptográficos para proveer una conexión segura entre dos puntos, garantizando autenticación, confidencialidad, e integridad. [36] Aunque TLS es independiente del protocolo de transporte subyacente, este ha sido mayormente implementado sobre TCP facilitando la aparición de protocolo seguros en capas superiores, como HTTPS.

permite tener mejoras interesantes como el handshake criptográfico de QUIC, que ahorra round trips con respecto a TCP y TLS por separado.

- Multiplexado de streams en una conexión QUIC, permitiendo resolver el problema de Head-of-Line Blocking. [2]
- Migración de conexiones para permitir establecer una conexión desde otra parte de la red.
- Control de congestión desacoplado: QUIC no define ningún mecanismo para el control de congestión, permitiendo la implementación de algoritmos conocidos y abriendo la posibilidad a implementaciones personalizadas.

Desde mayo de 2021, QUIC se encuentra estandarizado en el RFC9000[9] de la IETF. Durante este capítulo se realiza un análisis del protocolo QUIC, analizando su funcionamiento y los cambios mencionados previamente.

2.1. Historia y Motivaciones

Con más de 40 años de Internet facilitada por los protocolos TCP/IP [3], es fácil darse cuenta que esta ha cambiado considerablemente desde sus orígenes. El protocolo TCP no fue originalmente diseñado para soportar los casos de usos del presente. El crecimiento de los servicios web y su uso como plataforma para aplicaciones fue generando una demanda cada vez más grande en lo que respecta a eficiencia y reducción de la latencia. Adicionalmente, los requerimientos de seguridad cada vez más necesarios resultaron en la transición del tráfico inseguro a seguro basado en el protocolo TLS, por lo que también es necesario considerar la latencia generada por ese proceso extra.[1] La necesidad de menores tiempos de latencia, y el crecimiento de casos de uso han generado importantes avances en TCP. No obstante, esto mismo también ha resaltado sus puntos de fricción.

A su vez, el desarrollo e implementación de nuevos protocolos se ve dificultado por varios factores. Por un lado, se encuentra la presencia de Middleboxes. Estos son componentes que realizan funciones por fuera del ruteo de paquetes, como NATs y Firewalls [6]. Una conexión entre dos puntos puede pasar por una gran cantidad de saltos intermedios, representados por estos Middleboxes, como se puede observar en la figura 2.2. A su vez, estos suelen inspeccionar y manipular paquetes, y suelen descartar aquellos protocolos que no soporten. Aunque los Middleboxes son fundamentales para el funcionamiento de Internet, un análisis más profundo muestra que estos, accidentalmente, generan un acoplamiento a los protocolos de red que soportan.

[2] El Head-of-Line Blocking es un fenómeno que ocurre en protocolos de transmisión secuencial como TCP, donde la pérdida o retraso de un paquete afecta a todos los demás encolados detrás de él, ya que estos deben entregarse en orden.

Figura 2.2: Una comunicación entre dos dispositivos puede tener que pasar por varios middleboxes. Los Firewalls y NAT gateways son ejemplos comunes de estos componentes.

Tomemos por ejemplo TCP: aunque es de esperar que todos los Middleboxes lo soporten, no todos soportan las mismas versiones del protocolo. Esto resulta en que si un Middlebox encuentra un campo de TCP desconocido, el paquete se descarte. Dado que los Middleboxes, no se actualizan al mismo ritmo que otros componentes de Internet, la flexibilidad y velocidad para agregar cambios nuevos a los protocolos se ve considerablemente reducida. Este fenómeno se conoce como la *osificación* de un protocolo. En el caso de TCP, un cambio puede tardar años en tener suficiente soporte como para ser utilizado. [4][5][7]. Una de las soluciones para evitar más endurecimiento de los protocolos es la encriptación de los campos para evitar que los Middleboxes puedan inspeccionarlos y generar mayor acoplamiento.[1][5][7]

Otro punto de fricción viene dado por la velocidad de despliegue de nuevos cambios. En la actualidad, TCP suele estar implementado a nivel del kernel del Sistema Operativo. Esto implica que cualquier cambio dentro de la implementación del protocolo requiere una actualización del sistema operativo. Estos procesos suelen ser lentos y muchos dispositivos suelen quedarse con versiones antiguas.

Otra falencia conocida de TCP viene dada por la cantidad de *round trips* involucrados en el handshake de la conexión, en especial cuando este se usa en conjunto con TLS. El desacoplamiento de estos, aunque útil, implica que cada protocolo debe pasar por un proceso de handshaking antes de poder enviar datos. Con el crecimiento del tráfico HTTPS este incremento es cada vez más notorio.

El Head-of-line (HOL) Blocking es otro problema histórico que suele limitar el rendimiento del protocolo. Este suele aparecer en el contexto de HTTP/1.1, donde para reducir la latencia y el costo de mantener demasiadas conexiones, se limita la cantidad que se puede establecer con un servidor [10]. Por lo tanto, para hacer establecer una nueva conexión se debe esperar a que alguna de las anteriores se libere. Para resolver este problema, HTTP/2 puede multiplexar los pedidos en una misma conexión TCP, por lo que no es necesario esperar a que se liberen nuevas conexiones [14]. No obstante, el HOL Blocking sigue existiendo, pues se ve afectada por el control de congestión de TCP, es decir que si se pierde algún paquete, todos los streams deberán esperar a la retransmisión, como se puede observar en la figura 2.3.

Figura 2.3: Head of Line Blocking [22]

2.1.1. El modelo del reloj de arena

El proceso de osificación de TCP es especialmente importante en el contexto de la arquitectura de protocolos de Internet. Para entenderla se ha utilizado el modelo del reloj de arena como podemos observar en la figura 2.4. Este modelo muestra cómo los protocolos correspondientes a cada capa interactúan y dependen entre sí. En el medio, la parte más angosta del reloj, se encuentra el protocolo IP. Este queda posicionado como el único protocolo intermediario entre las capas de enlace y aplicación.

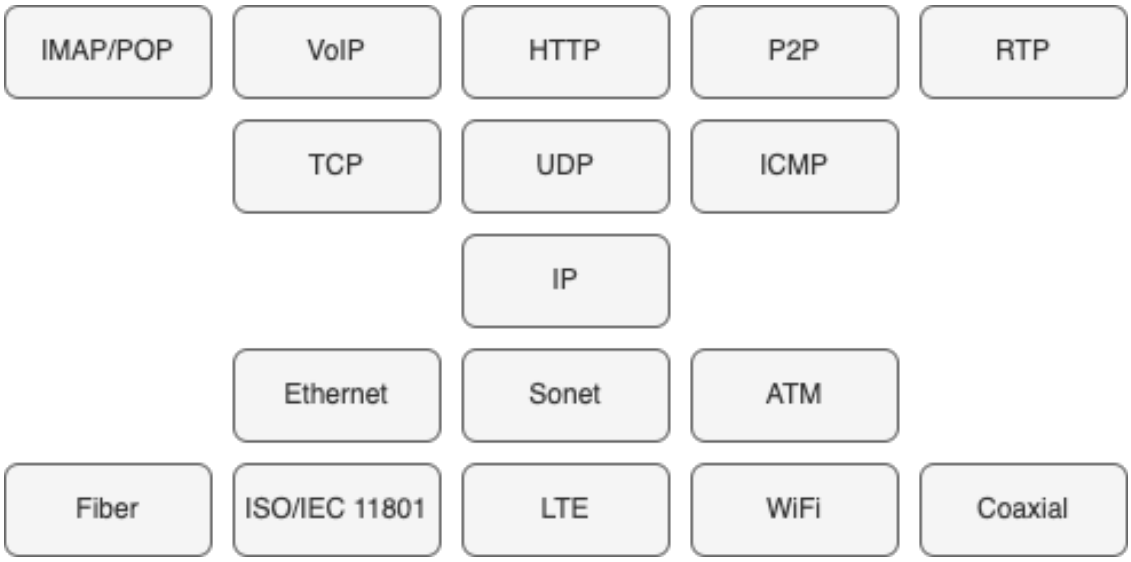


Figura 2.4: El modelo del reloj de arena original. Este referencia protocolos que operan en distintas capas de la red. Esta versión del modelos fue actualizado para referencia tecnologías modernas.

En su artículo sobre el modelo del reloj de arena, Rosemberg [4] argumenta que la presencia de middleboxes como NATs [3]y firewalls han llegado a un nivel de despliegue tan importante que resulta en que cualquier conexión entre dos puntos deba hacer uso de TCP o UDP, pues ambos dispositivos requieren estos protocolos. A su vez, otros protocolos como ICMP suelen ser filtrados como práctica estándar, disminuyendo fuertemente su efectividad. Por lo tanto, podemos decir que el Internet moderno es dependiente de TCP y UDP para funcionar de manera efectiva. Como consecuencia, se argumenta que la cintura

del reloj de arena fue moviéndose del protocolo IP a TCP y UDP.

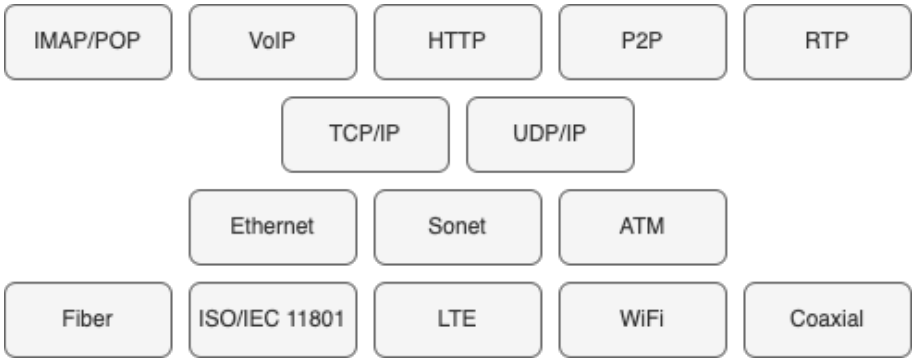


Figura 2.5: El modelo del reloj de arena original actualizado según Rosenberg. [4]

El modelo de la figura 2.5 es más claro para explicar porqué es tan difícil desarrollar y adoptar un nuevo protocolo. El soporte "monopolizado" por TCP y UDP en la capa de transporte ha llevado a la aparición de nuevo modelos y desarrollos donde la innovación debe ocurrir encima de alguno de estos protocolos.[7] En este contexto, UDP surge como punto base para la construcción de un nuevo protocolo dada su simplicidad[4]. A su vez, UDP también es soportado por todo tipo de middleboxes, lo cual ayuda a facilitar su despliegue y adopción. No obstante, utilizar UDP implica que si necesitamos funcionalidades como confiabilidad o control de congestión será necesario implementarlos por encima de este sin tener acceso a los avances desarrollados en las últimas décadas para TCP. Por otro lado, esto permite rehacer o revisar mecanismos existentes en TCP que en la actualidad serían muy difíciles de modificar.

2.1.2. Un nuevo protocolo

En 2012, Google decide desarrollar el protocolo QUIC (ahora conocido como gQUIC) usando UDP como transporte con el objetivo de mejorar el rendimiento del tráfico HTTPS. En 2015 se envía un draft a la IETF para la estandarización del protocolo [8], y en Mayo 2021 se finaliza en el RFC 9000. [9]

[3]Un NAT Gateway es un dispositivo o servicio que actúa como intermediario entre una red privada y una red externa, permitiendo que los dispositivos en la red privada accedan a Internet o a otra red externa mediante la traducción de direcciones IP y puertos. Es comúnmente utilizado en redes domésticas y empresariales para proteger los dispositivos de la red privada y para ahorrar en la asignación de direcciones IP públicas.

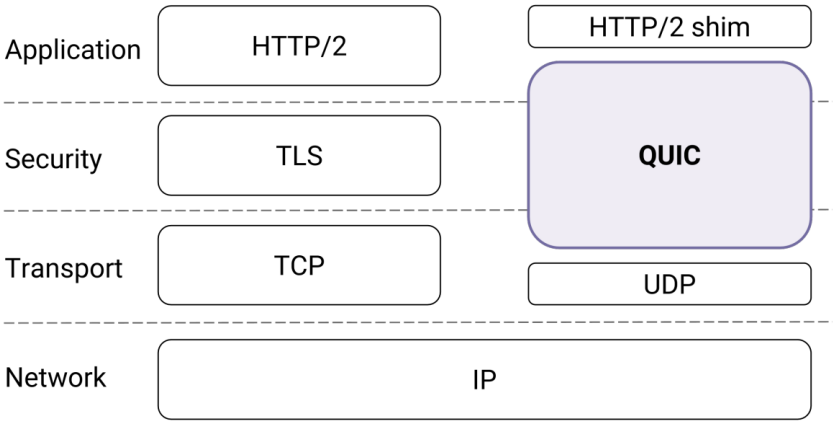


Figura 2.6: Implementación original de QUIC (ahora conocido como gQUIC) por Google. [1] En su concepcion orginal, gQUIC fue pensado como un experimento para mejorar el tráfico HTTPS. Dado que HTTP/2 fue diseñado para funcionar sobre el protocolo TCP, se introdujo una capa de adaptación representada en la figura como HTTP/2 shim[4].

QUIC está diseñado como un protocolo transportado por encima de UDP, lo que le permite atravesar middleboxes sin problemas. Adicionalmente, su implementación existe en el user space, lo cual implica una mayor facilidad de despliegue y evolución del protocolo. Más aún, QUIC provee las interfaces para poder implementar fácilmente nuevos mecanismos de control de congestión.

Para reducir la latencia y securizar el protocolo, el handshake QUIC integra TLS, permitiendo ahorrar round trips en el establecimiento de una nueva conexión. Adicionalmente, al usar TLS 1.3 y su capacidad para resumir sesiones, es posible comenzar a enviar la información lo antes posible en ciertos casos, a través de un 0-RTT handshake.

QUIC es un protocolo orientado a conexiones, al igual que TCP, pero a diferencia de este último, QUIC permite la multiplexación de streams en una única conexión sin sufrir Head-of-line blocking.

A continuación se procede a revisar en mayor detalle las distintas características del protocolo.

2.2. Streams

Uno de los cambios más interesantes en comparación con TCP, es la introducción de streams en la capa de transporte. Estos son una abstracción liviana dentro de una conexión QUIC que representa un flujo ordenado de bytes confiable. [9] Este concepto es la clave para resolver el bloqueo HOL que se encuentra en TCP. Al soportar una cantidad arbitraria de streams dentro de una conexión, QUIC asegura que la pérdida de un paquete UDP afecte solo a los streams cuyos datos estaban siendo transportados. [1][9] Una forma de pensar esto podría ser ver cada stream QUIC como si fuese una conexión TCP aislada.

[4]En el contexto del desarrollo de software, un *shim* (espaciador o calce) es un módulo que permite el funcionamiento entre dos capas o componentes originalmente incompatibles.

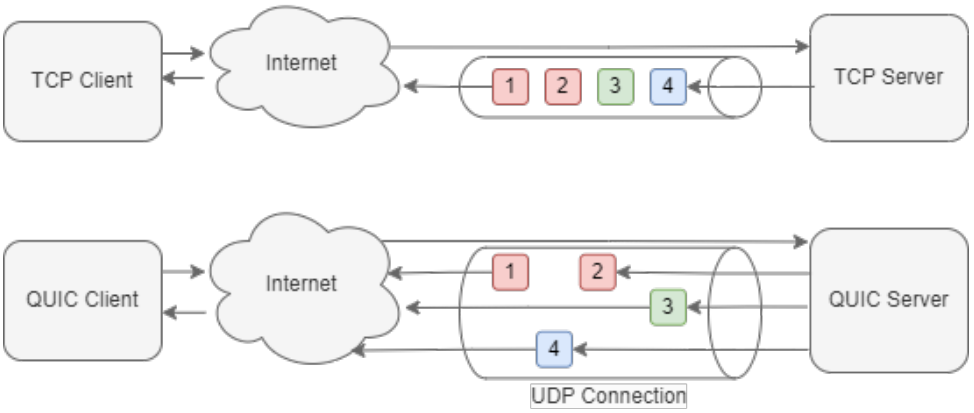


Figura 2.7: Comparación de una conexión QUIC vs. una conexión TCP tradicional.

En la figura 2.7, se puede observar una comparación entre una conexión TCP y una realizada a través de QUIC. En este escenario, es importante notar cómo el orden de los paquetes se mantiene dentro de los streams, pero no es necesario respetarlo entre cada uno de ellos. Por lo tanto, es posible que el paquete 4 llegue a su destino antes que los paquetes 2 y 3.

2.2.1. Multiplexado

La introducción de streams introduce el problema del multiplexado dentro de una conexión QUIC. En otras palabras, cómo distribuimos la conexión de manera justa entre todos los streams involucrados. La especificación de QUIC no describe cómo resolver este problema sino que lo deja libre para ser resuelto por los implementadores. A su vez se recomienda que la implementación soporte la posibilidad de recibir indicadores de prioridad relativa entre streams por parte de la aplicación. Algunas técnicas de multiplexado soportadas en las diversas implementaciones de QUIC incluyen Round Robin, Round Robin cada 14 paquetes, Secuencial FIFO/LIFO, entre otras. Una representación gráfica de estas puede observarse en la figura 2.8.

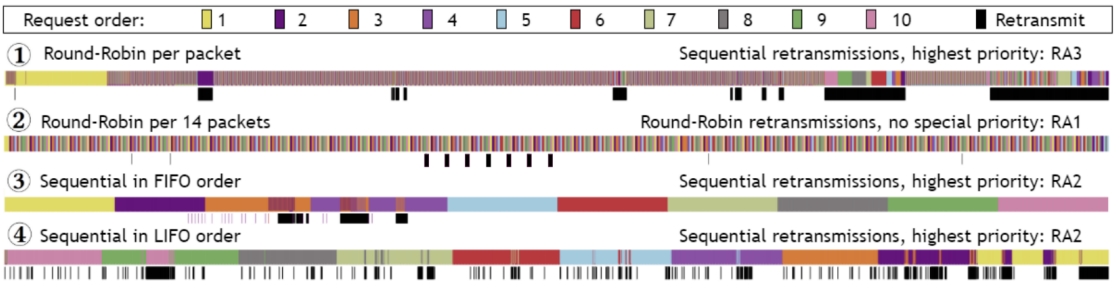


Figura 2.8: Este gráfico examina diversas estrategias de multiplexado de streams para llevar a cabo una serie de descargas en paralelo. En esta representación, los datos se desplazan de izquierda a derecha, y cada rectángulo coloreado simboliza un paquete asociado a un stream particular. Las áreas coloreadas extensas indican descargas secuenciales que involucran varios paquetes pertenecientes al mismo flujo. [10]

2.3. Conexiones

La especificación de QUIC define una conexión como un estado compartido entre un cliente y un servidor. Para establecer y securizar una conexión, se utiliza un handshake que implementa simultáneamente seguridad en la capa de transporte. [16] Es decir, que aparte de definir los parámetros de la conexión, también se realiza el proceso de verificación e intercambio de claves de TLS.

Es importante recordar que QUIC debe volver a construir el concepto de conexión dado que este no se encuentra presente en UDP, un protocolo orientado a datagramas. Lo que podría parecer inicialmente un retrabajo para generar conexiones sobre UDP, es una oportunidad para el protocolo QUIC de revisar y mejorar ciertos puntos de fricción sobre las conexiones TCP, como la identificación y el handshake entre otros.

2.3.1. Connection ID

En QUIC, cada conexión posee un conjunto de *IDs* que la identifican. Estos son elegidos independientemente por el cliente y el servidor. Adicionalmente, cada participante de la conexión puede tener múltiples IDs disponibles y ofrecerlos de antemano. De esta manera no es posible para un observador identificar a qué conexión pertenece un paquete correlacionando los IDs.

La función principal de IDs de conexión de QUIC es asegurar que la conexión persista sobre cambios en los protocolos inferiores. [9] Por ejemplo, si un cliente cambia de IP, el servidor podría identificar que los paquetes recibidos corresponden al mismo cliente pues ya conoce el/los IDs. Esto es un diferencial significativo con TCP, que utiliza IPs y puertos de origen y destino para identificar las conexiones. Por lo tanto, ante un cambio de alguno de estos campos, TCP debe considerar la conexión como nueva.

La capacidad de QUIC para sobrevivir a cambios en la red se conoce como migración de conexiones y es especialmente útil en sistemas móviles donde los cambios entre redes WiFi y móviles son frecuentes. En la figura 2.9 se puede observar cómo el mecanismo funciona en un dispositivo móvil que cambia de una red WiFi a 4G. También es importante notar que el connection ID no es reutilizado al realizar la migración de conexiones para prevenir

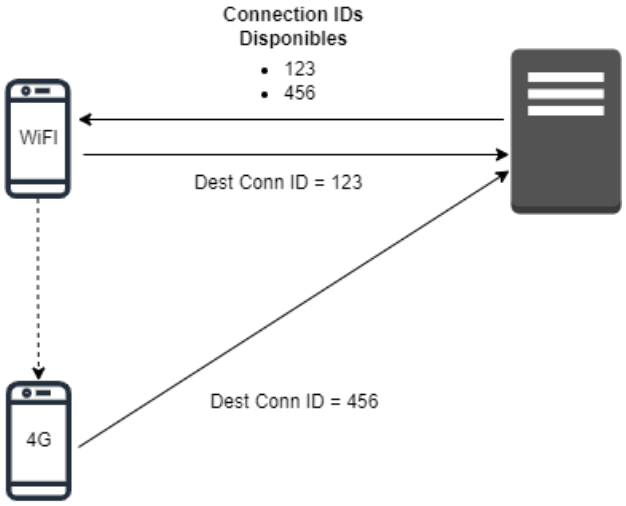


Figura 2.9: Esquema de migración de conexiones en un dispositivo móvil.

una posible correlación del tráfico por parte de un observador. [9]

2.3.2. Handshake

El handshake es el proceso por el cual dos puntos establecen una conexión y sus parámetros. Recordemos que UDP, al ser orientado a datagramas, no incluye el concepto de conexión y por lo tanto es responsabilidad de QUIC volver a implementarlo.

El handshake de QUIC toma las lecciones aprendidas de TCP y sus casos de la actualidad, en particular el uso conjunto con TLS para securizar la capa de transporte. [16] QUIC resuelve esto integrando su handshake con el de TLS de manera tal que podemos establecer toda la conexión en lapso de un RTT como se observa en la figura 2.10.

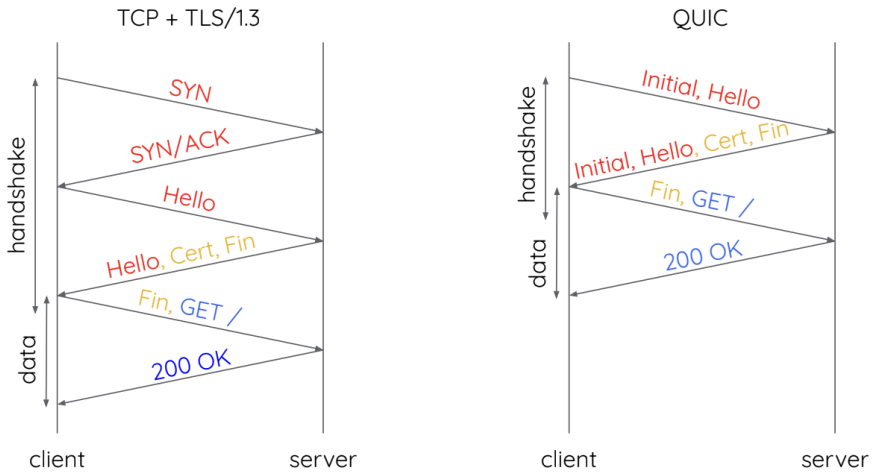


Figura 2.10: Handshake inicial en TCP+TLS1.3 Vs. QUIC [24]

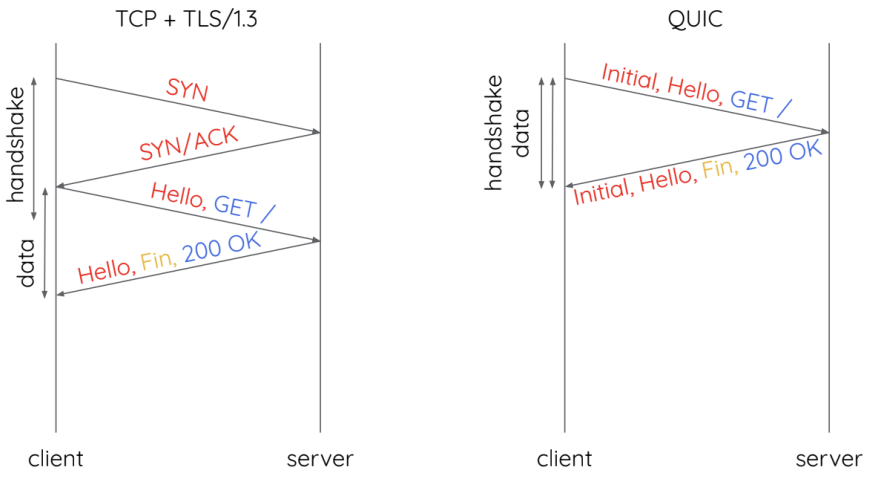


Figura 2.11: Handshake para conexiones subsecuentes en TCP+TLS1.3 Vs. QUIC [24]

QUIC también soporta un 0-RTT handshake para conexiones subsecuentes al mismo servidor, lo cual permite enviar datos directamente en el primer paquete. Este hace uso de la funcionalidad de *session resumption* de TLS 1.3 que permite la reutilización de las claves de una conexión anterior para evitar realizar un nuevo handshake y comenzar a enviar datos lo antes posible. [36]

Otro proceso importante que sucede durante el handshake es la negociación de versiones y parámetros. Para esto, el cliente propone una versión en el primer paquete de la conexión y continua el resto del handshake usando esta misma. Si el servidor soporta esa versión el handshake continúa normalmente y se evita enviar un ACK. En cambio, si esta no fuera soportada, el servidor enviará un paquete de negociación de versión con todas sus versiones soportadas.

2.3.3. Paquetes

La comunicación en QUIC se hace a través de paquetes transportados en datagramas UDP. Al mismo tiempo, un paquete está compuesto por un header seguido por uno o más frames. Estos están compuestos por un campo que indica su tipo y luego por una serie de campos que varían según este.

Cada tipo de frame tiene su su semántica asociada dentro del protocolo. Algunos ejemplo de frames son ACK, STREAM, CRYPTO, etc

Otra característica importante del protocolo es la presencia de dos tipos de encabezado de paquetes, el largo y el corto. El primero es utilizado solamente en paquetes relacionados al establecimiento de la conexión, mientras que el segundo está diseñado para enviar datos y reducir el tamaño de los paquetes una vez establecida la conexión.

[5] `https://docs.citrix.com/en-us/citrix-adc/current-release/system/http3-over-quic-protocol.html#how-http3-protocol-works`

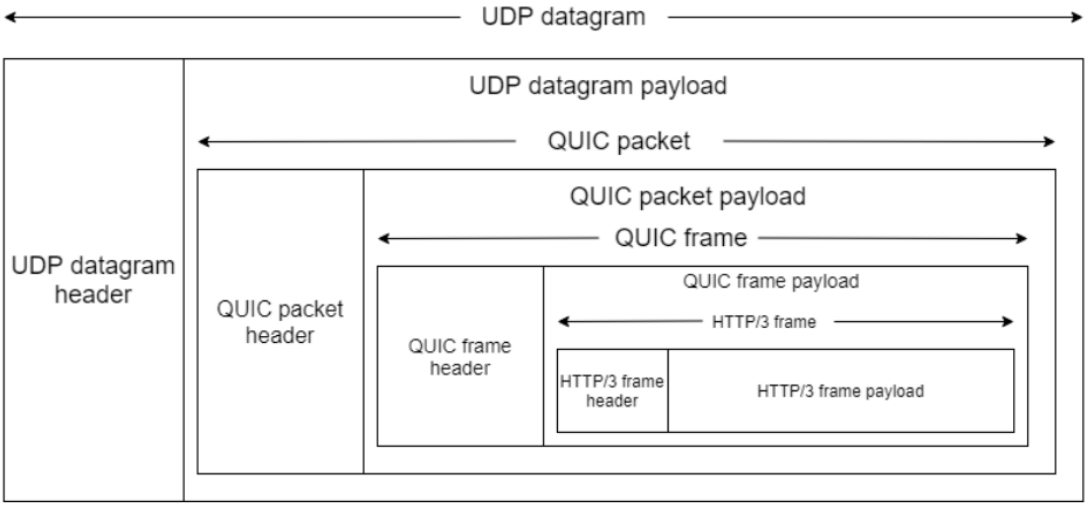


Figura 2.12: Estructura de un paquete QUIC. [5]

2.3.4. Tipos de paquete

Como se mencionó en la sección 2.3.3, no todos los paquetes QUIC son iguales. Cada tipo de paquetes tiene distintos roles y niveles de protección criptográfica. [9] A su vez, cada uno tiene asociado un tipo de header. A continuación, detallamos los distintos tipos de paquetes y sus roles dentro de la conexión QUIC.

Tipo de Paquete	Tamaño de Encabezado
Version negotiation	Largo
Initial	Largo
0-RTT	Largo
Handshake	Largo
Retry	Largo
1-RTT	Corto

Tabla 2.1: Tipos de Paquete QUIC y su tamaño de encabezado.[9]

- **Versión Negotiation**: Este tipo de paquete fue diseñado para ser independiente a la versión de QUIC utilizada. Estos paquetes no necesitan un ACK para indicar que se recibieron. Estos son únicamente utilizados como respuesta desde un servidor cuando detectan que un cliente envío un paquete que corresponde a una versión no soportada.
- **Initial**: Los paquetes Initial envían los primeros frames CRYPTO para intercambiar claves y comenzar a establecer la conexión. Estos se utilizan para enviar y recibir cualquier mensaje que inicie un handshake criptográfico. Una vez que se envía un paquete Handshake, estos se dejan de utilizar y se descartan sus claves.

- **0-RTT**: Estos paquetes se utilizan para enviar datos del cliente previo a la finalización del Handshake. Esto hace uso de la funcionalidad de *session resumption* de TLS 1.3. Es decir, reutiliza los parámetros negociados de una conexión anterior. [16]. Una vez que se comienzan a procesar paquetes de tipo 1-RTT, no se deben volver a enviar paquetes 0-RTT.
- **Handshake**: Este tipo de paquete es utilizado para enviar mensajes del handshake criptográfico subsiguientes al comienzo del handshake criptográfico.
- **Retry**: Este puede ser enviado por el servidor en respuesta a un paquete 0-RTT o Initial con el objetivo de validar el cliente de una conexión. Esto es importante pues si el cliente de una conexión no es validado, se corre el riesgo de un ataque de amplificación. El servidor usa los paquetes tipo retry para validar un cliente previo al comienzo del handshake. [9] [16]
- **1-RTT**: Este tipo de paquete es utilizado una vez que se completa el handshake criptográfico y las keys de la conexión fueron negociadas.

2.3.5. Combinación de paquetes

A pesar de necesitar de varios tipos de paquetes para realizar el handshake criptográfico, QUIC puede realizar este proceso en 4 datagramas UDP[6], como así lo indica el 1-RTT handshake. Esto se logra a través de un proceso de combinación de paquetes (*packet coalescing*) que permite transportar varios de estos en un único datagrama UDP. El receptor del datagrama debe procesar cada paquete de forma individual y enviar el ACK para cada uno por separado. [9] Una vez completado el handshake, esta funcionalidad se vuelve menos útil. La especificación de QUIC recomienda priorizar el envío de múltiples frames en un único paquete en vez de enviar varios paquetes del mismo tipo en un datagrama.

2.3.6. Números de paquete

Todos los paquetes QUIC llevan consigo un número de paquete. Estos son enteros incrementales y pueden ser usados una única vez una conexión.

El uso de números de paquetes únicos tiene varias ventajas. Por un lado permite ser utilizado como un *nonce*[7] criptográfico, pues cada número se utiliza una única vez. Por el otro, esto resuelve el problema de la ambigüedad de retransmisión en TCP. Este consiste en no poder diferenciar si un paquete fue retransmitido o solamente llegó tarde, pues ambos son identificados con el mismo número. [17] La solución de QUIC simplifica los algoritmos de detección de pérdidas y medidas de RTT. Es importante mencionar que en QUIC el número de paquete no se utiliza para ordenar el contenido de los frames, sino que se utiliza otro campo que lleva el orden de los datos para cada stream.

Los números de paquete están divididos en espacios. Estos definen el contexto en el que un paquete puede ser procesado y recibido [9]. Para cada espacio, se utilizan claves criptográficas distintas y solamente pueden enviarse ACK dentro de paquetes que pertenezcan a este mismo. Para complementar la definición inicial, el número de paquete QUIC

[6] El uso de paquetes retry puede cambiar la duración del handshake

[7] Un nonce es un número arbitrario que se utiliza una única vez en una comunicación criptográfica y permite prevenir ataques por replay.

es un entero incremental y único dentro de un espacio de números de paquete. Los espacios existentes son:

- **Initial Space**: Utilizado por los paquetes de tipo *Initial*.
- **Handshake Space**: Utilizado por los paquetes de tipo *Handshake*.
- **Application Data Space**: Utilizado por los paquetes de tipo *0-RTT* y *1-RTT*.

En el caso de los paquetes de tipo *Retry* y *Version Negotiation* no llevan número de paquete y por lo tanto no hacen uso de ningún espacio.

2.3.7. QUIC bit y multiplexado de protocolos

Una particularidad que se da en todos los tipos de paquetes, es el llamado *QUIC bit*. Este es el segundo bit más significativo de cada paquete QUIC y siempre tiene el valor 1. Cualquier otro valor resulta en un paquete invalido que debe ser descartado [9].

La razón de este campo particular es poder utilizarlo para distinguir un paquete QUIC de uno de otro tipo. Esto es necesario cuando queremos trabajar con multiplexar protocolos dentro de una única conexión UDP. Un esquema similar está definido para otros protocolos que funcionan sobre UDP como STUN, DTLS, RTP, etc. En la Figura 2.13 está definido el algoritmo que debe realizar el receptor para demultiplexar estos paquetes.

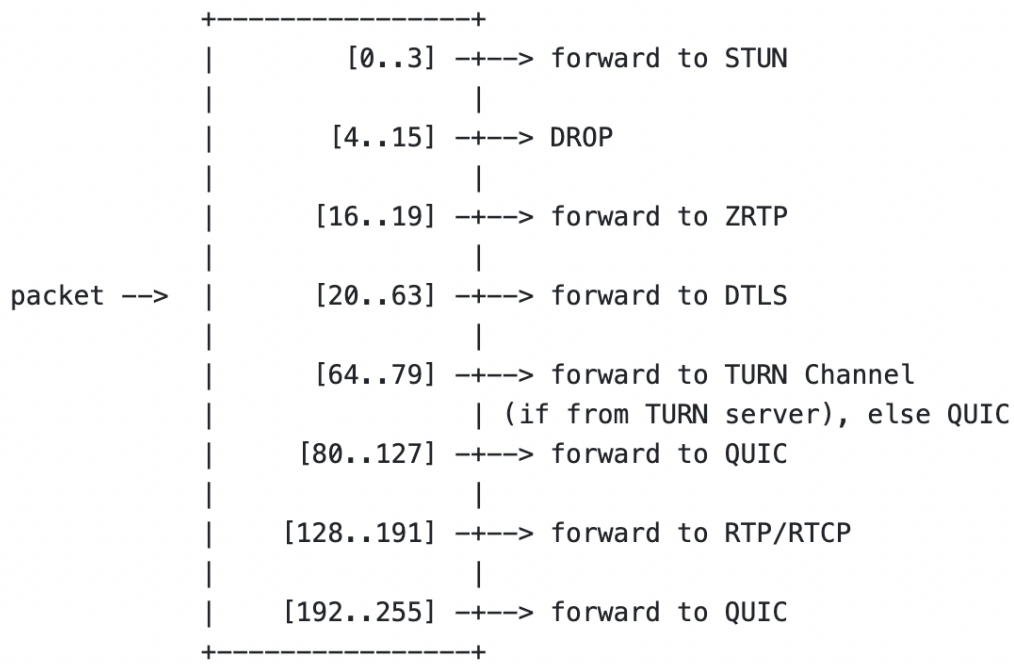


Figura 2.13: Demultiplexado de paquetes UDP. Este consiste en inspeccionar el primer byte de cada paquete con el objetivo de determinar el de protocolo utilizado. [41]

Es importante resaltar que el QUIC bit forma parte de los pocos campos expuestos en un paquete QUIC. Esto lo pone en riesgo de acoplarse a implementaciones y osificarse. En la sección 2.6.3 se analiza un funcionamiento alternativo para este campo.

2.4. Detección de pérdidas y Control de congestión

2.4.1. Estimación del Round Trip Time

Al igual que en TCP, el emisor QUIC toma muestras de los RTT de cada paquete para estimar cuánto tiempo le toma transmitir un paquete a través de la red. Para generar una muestra se observa el tiempo transcurrido desde que el paquete más grande que recibe ACK fue enviado.

Para generar una nueva muestra, se observan los frames ACK. Esta es generada usando únicamente el número de paquete más grande en el frame ACK. Esto se debe a que el receptor puede enviar el ACK Delay[8]solamente para el paquete más grande del frame. Adicionalmente, para no generar múltiples muestras por paquete, QUIC no utiliza ACK frames sí incluyen el número de paquete más grande pero este ya fue reconocido previamente.

La muestra de RTT se toma de la siguiente forma:

$$latest\_rtt = ack\_time - send\_time\_of\_largest\_acked$$

En base a las muestras, QUIC mantiene tres valores para estimar el RTT: *min_rtt* , *smoothed_rtt*, y *rttvar*.

2.4.2. Estimación del *min_rtt*

Como lo indica su nombre, *min_rtt* mantiene el valor mínimo de la red observado por el emisor. Su valor se calcula ante cada nueva muestra como el mínimo entre esta y el valor actual de *min_rtt*. Esto permite adaptarse sobre mejoras en los tiempos de la red. Si el valor real de RTT aumentara, el *min_rtt* no se adaptaría a este cambio, sino que se utilizarían otro tipo de estimaciones. El cálculo de *min_rtt*, representa los tiempos observados desde el emisor, por lo que no se toman en cuenta los ACK Delays reportados.

La especificación de control de congestión de QUIC [15] recomienda cambiar el valor de *min_rtt* al de la última muestra cuando se detecta congestión persistente. Esto permite reestablecer las estimaciones antes eventos disruptivos en la red.

2.4.3. Estimación del *smoothed_rtt* y *rttvar*

Estos valores suelen ser utilizados en conjunto. *smoothed_rtt* es una media móvil exponencial de las muestras de RTT, mientras que *rttvar* estima su variación.

A diferencia del *min_rtt*, para el cálculo de *smoothed_rtt* se usan muestras ajustadas en base al ACK Delay. Esto se basa en restar el delay reportado de lado del receptor al RTT observado en la muestra. El ACK Delay podría ignorarse si el valor ajustado de la muestra es menor que el *min_rtt*.

[8]El ACK Delay es un retraso introducido de manera intencionada entre la recepción de un paquete y el envío de su correspondiente ACK. Este retraso permite esperar la llegada de nuevos paquetes para enviar una única respuesta que contenga todos los ACKs. El ACK Delay se mide y codifica en un campo específico del frame de tipo ACK.

Los valores se *smoothed_rtt* y *rttvar* son calculados similarmente a los de TCP [23]. La estimación es inicializada durante el establecimiento de la conexión y cuando se detecta una migración de conexiones.

2.4.4. Detección de pérdidas

La detección de pérdidas en QUIC está influenciada fuertemente por los mecanismos de *TCP Fast Retransmit* [29], *Early Retransmit* [30], *Forward Acknowledgment* [31], *SACK Loss Recovery* [32], y *RACK-TLP* [33]

A diferencia del cálculo de RTT y el control de congestión, los mecanismos de detección de pérdidas en QUIC se realizan por cada espacio de números de paquete. Esto se debe a que es necesario contar con las claves de encriptación para realizar ACKs. [24]

Un paquete es declarado como perdido si cumple con los siguientes criterios [15]: El paquete no recibió un ACK, y fue enviado previo a un paquete del que sí se tiene un ACK. El paquete fue $kPacketThreshold$ paquetes antes que uno que sí recibió un ACK, o fue enviado hace un tiempo suficientemente largo

Los umbrales mencionados en los criterios proveen una cierta tolerancia al protocolo ante la llegada de paquetes fuera de orden. A continuación veremos en mayor detalle cómo se definen.

La especificación de QUIC recomienda inicializar el umbral de paquetes en 3 basado en las prácticas existentes para TCP. [15][29] [32] Por otro lado no recomienda utilizar valores más bajos para así mantener una similitud con TCP.

El umbral de tiempo es utilizado para declarar como perdidos, paquetes que fueron enviados hace cierta cantidad de tiempo. Este umbral se calcula de la siguiente forma:

$$max(kTimeThreshold * max(smoothed\_rtt, latest\_rtt), kGranularity)$$

Donde:

- $smoothed\_rrt$ y $latest\_rtt$ son mediciones del RTT, como las descritas en la sección anterior.
- $kGranularity$ es la granularidad del reloj del sistema, se recomienda utilizar al menos $1ms$.
- $kTimeThreshold$ es un multiplicador del RTT. La especificación recomienda utilizar un valor de 9/8 (a diferencia de TCP RACK donde se suele usar 5/4)

2.4.5. Probe Timeout

El Probe Timeout (PTO) está asociado a un temporizador que genera el envío de uno o dos datagramas sonda (probe) cuando no se recibe el ACK de un paquete en el periodo de tiempo esperado. Este mecanismo se usa para detectar y poder recuperarse de la pérdida de paquetes al final de la ventana enviada. La implementación de QUIC está fuertemente influenciada por los avances realizados en TCP, (RACK-TLP[9] y F-RTO[10]).

[9]RACK-TLP es un algoritmo de detección de pérdidas en TCP que mejora la técnica existente de ACKs duplicados. Este se encuentra formado por dos partes. Por un lado, Recent Acknowledgment (RACK) para detectar pérdidas más rápidamente usando temporizadores por segmento. Por el otro, Tail loss Probe (TLP) provoca el envío de ACKs para evitar timeouts de retransmisión.

[10]Forward RTO-Recovery (F-RTO) es un algoritmo para detectar retransmisiones espurias en TCP y así evitar retransmisiones innecesarias luego de un timeout de retransmisión (RTO).

El cálculo del *PTO* está basado en el cálculo del retransmission timeout de TCP. Este tiene la siguiente forma

$$PTO = smoothed\_rtt + max(4 * rttvar, kGranularity) + max\_ack\_delay$$

En la práctica el valor del *PTO* debe ser al menos *kGranularity* para evitar que este expire inmediatamente. Cuando el temporizador expira, el emisor envía uno o dos paquetes que cumplen la función de sondas. La razón de enviar más de una sonda es prevenir la posibilidad de un timeout debido a la pérdida de un único paquete Adicionalmente, ante una expiración, se incrementa un backoff que resultará en que el nuevo valor del *PTO* se duplique hasta alcanzar el idle timeout. Este se reinicia una vez que se recibe un nuevo ACK. El uso de backoffs en esta etapa es importante pues la red puede encontrarse con una congestión muy intensa. Por esto mismo QUIC define el PTO backoff como común a todos los espacios de número de paquete.

2.4.6. Control de Congestión

Un cambio interesante que introduce QUIC en esta área es el desacoplamiento del control de congestión del envío confiable. Esto surge como consecuencia del uso de números de paquetes monotónicos, pues estos se utilizan para el control de congestión mientras que el orden de los bytes para envío confiable viene dado por el campo offset transportando en todos los STREAM frames.

Al igual que en TCP, QUIC utiliza un esquema de control de congestión basado en ventanas que limitan la cantidad máxima de bytes que pueden estar en tránsito. No obstante, QUIC no pretende ni especifica la implementación de algún algoritmo de control de congestión, sino que define una serie de señales genéricas de forma que el emisor pueda implementar el algoritmo de su elección. En general las implementación de QUIC existentes suelen implementar al menos el algoritmo *New Reno*[11].

2.4.7. Congestión Persistente

Una de las responsabilidades del controlador de congestión es detectar y declarar un estado de congestión persistente. Esto sucede cuando se establece la pérdida de todos los paquetes durante una ventana de tiempo suficientemente grande. La duración de la congestión persistente se calcula de la siguiente forma:

$$(smoothed\_rtt \ + \ max(4 * rttvar, \ kGranularity) \ + \ max\_ack\_delay) * \\ kPersistentCongestionThreshold$$

El valor de $kPersistentCongestionThreshold$ tiene como objetivo regularizar la sensibilidad a al hora de detectar congestión. Valores más altos resultan en un envío de paquetes más agresivo mientras la red experimenta congestión. Por otro lado, valores más chicos pueden resultar en casos donde la congestión se declare muy rápido innecesariamente. El valor recomendado en la especificación de QUIC es 3.

[11]New Reno es un algoritmo de control de congestión TCP originalmente propuesto en 1999. Este es una iteración sobre sus predecesores TCP Reno y Tahoe. New Reno utiliza la pérdida de paquetes como feedback e incluye técnicas ampliamente adoptadas, como Slow Start, AIMD, Fast Retransmit y Fast Recovery.

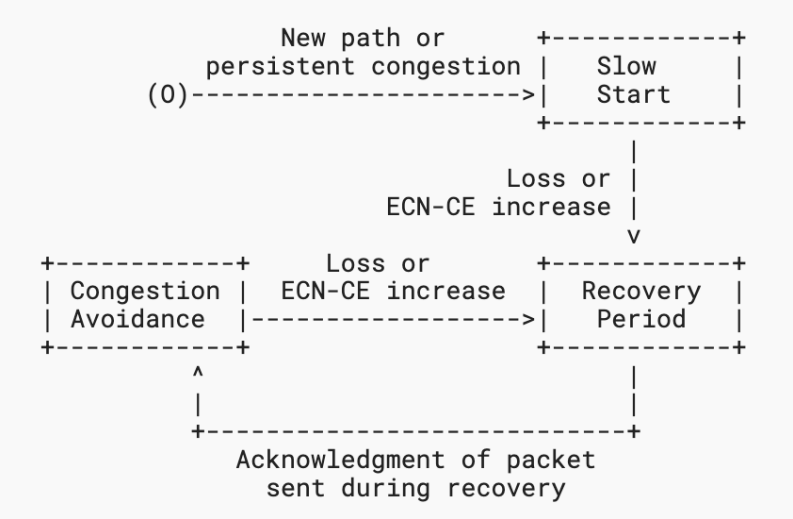


Figura 2.14: Implementación del algoritmo *New Reno* según QUIC [15]

2.5. Consideraciones Finales

Durante este capítulo hemos estudiado las distintas funcionalidades del protocolo QUIC. En la sección 2.1 analizamos cómo a pesar de tener más de 40 años, el protocolo TCP cuenta con falencias que no pueden ser fácilmente resueltas pues surgen de su diseño original. Es decir, TCP no fue pensado para el uso intensivo que tiene en la actualidad. El protocolo QUIC busca tomar estas lecciones aprendidas y aplicarlas en su diseño, buscando maximizar la eficiencia de este resolviendo varios problemas que dificultaron el avance en TCP. Uno de estos problemas era la implementación en el Kernel. Esto dificultaba el despliegue de actualizaciones y la velocidad de desarrollo. Al estar construido sobre UDP, QUIC es implementado en el user-space, lo cual permite desarrollar y desplegar con mayor flexibilidad. Este cambio no es menor, y conlleva ventajas tanto como desventajas. Por un lado ya mencionamos como este puede incrementar la velocidad en la que el protocolo evoluciona. No obstante, este cambio también resulta en una mayor variedad de implementaciones del protocolo. Esto puede ser beneficioso pues fomenta la competencia entre cada implementación, pero al mismo tiempo aumenta la complejidad pues cada una crecerá a ritmos distintos y será tarea del desarrollador elegir la que mejor se adecua.

Otro concepto importante que aparece en QUIC es el multiplexado de streams en una única conexión. Este no es algo nuevo pues ya existen otros protocolos que lo hacen. El diferencial en este caso es implementar esto en la capa de transporte, pues ahora los protocolos de capas superiores (como ya veremos con HTTP/3) podrán tener multiplexado de conexiones sin tener bloqueo HOL.

Cuando analizamos la detección de pérdidas y control de congestión en QUIC nos encontramos con que varios de los conceptos de TCP son reutilizados, pero con algunas variaciones para adaptarse mejor al protocolo o simplificar ciertos mecanismos que en TCP eran problemáticos. Una de las diferencias más interesantes es el desacoplamiento de los algoritmos de control de congestión, de manera que cualquier estrategia se pueda mapear sobre QUIC sin tener que cambiar el protocolo. Esto debería terminar resultando en la aparición de nuevos algoritmos, o incluso soluciones ad-hoc a cierto dominio.

Además de todos los cambios introducidos, hay un aspecto que no fue considerado que es el soporte del protocolo en middleboxes y componentes de red. Como ya fue mencionado, dado que todos estos soportan UDP también pueden aceptar tráfico QUIC. No obstante, el contenido de los paquetes viaja totalmente encriptado. Aunque esto es muy beneficioso si lo consideramos desde el punto de vista de la privacidad y combatir la censura de tráfico, también resulta en un desafío de seguridad a la hora de securizar una red y aplicar reglas al tráfico QUIC. [34] Esto podría resultar en casos donde el protocolo UDP sea bloqueado por un firewall, resultando en un problema de adopción para QUIC.

Finalmente, es importante considerar que QUIC se encuentra en su primera versión aprobada y el verdadero impacto y adopción de estas funcionalidades será realmente comprobado en los próximos años. A su vez, esto resultará en el desarrollo de nuevas y mejores herramientas para administrar su tráfico.

2.6. Extendiendo QUIC

Como fue mencionado en la sección 2.5, QUIC recién se encuentra en su primera versión. Esto implica que aún quedan un gran cantidad de mejoras e iteraciones para seguir madurando el protocolo. Algunos de estos cambios surgirán del feedback que se reciba sobre su funcionamiento en producción. No obstante, ya existe una serie de propuestas para expandir y evolucionar el protocolo. A continuación pasamos a detallar algunas de estas.

2.6.1. Datagramas sobre QUIC

En la sección 2.4 se analizaron los diversos mecanismos de QUIC para asegurar confiabilidad en el transporte de datos (tal como lo provee TCP). No obstante, aún existen casos de uso donde se prefiere el envío de datos de forma no confiable, como aplicaciones que necesitan transmitir en tiempo real. Originalmente estas se encontraban construidas directamente sobre UDP y DTLS[12]. En el RFC 9221, se describe una extensión de QUIC para soportar el envío de datagramas de forma no confiable. La especificación define la posibilidad de poder utilizar streams confiables y datagramas dentro de una misma conexión, permitiendo compartir el handshake y autenticación. A su vez en el RFC se argumenta sobre las ventajas de la implementación de TLS dentro de QUIC en comparación con DTLS. También es importante notar que los datagramas implementados en QUIC están sujetos al control de congestión de la conexión, lo cual puede resultar en mejor uso de ésta en comparación con utilizar UDP. [37]

Para integrarse con QUIC, se agrega un nuevo tipo de frame llamado DATAGRAM que se encargará de transportar datos de manera no confiable. Al igual que los STREAM frames, este es encriptado utilizando claves 0-RTT o 1-RTT. No obstante, a diferencia de este último, los datagramas no están asociados a ningún stream. Si una aplicación estuviera interesada en diferenciar grupos de datagramas utilizando algún identificador deberá hacerlo en un protocolo de capa superior.

[12]DTLS (Datagram Transport Layer Security) es un protocolo diseñado para proveer seguridad a una conexión UDP similar a lo realizado por TLS en TCP.

2.6.2. QUIC v2

A pesar del poco tiempo de vida QUIC, el grupo de trabajo responsable ya se encuentra trabajando en un borrador de la segunda versión del protocolo. No obstante, el objetivo principal de esta nueva versión no es agregar nuevas funcionalidades. QUIC v2 tiene el objetivo de atacar riesgos latentes de osificación dentro del protocolo y comenzar a hacer uso de los mecanismos de negociación de versiones. Este se encuentra planteado de forma que conforme el conjunto mínimo de cambios a realizar para especificar una nueva versión de QUIC. [38]

Durante la sección 2.1 se analizó el concepto de osificación y cómo se trabaja para combatirla. Por lo tanto, es importante entender dónde existe este riesgo dentro QUIC y por qué la versión 2 busca continuar minimizando el alcance de este problema. Un claro ejemplo de esto es el campo de versión que forma parte de los encabezados largos de cada paquete. Dado que actualmente solo existe una única versión del protocolo (descartando las experimentales, previas a la estandarización), un middlebox podría decidir osificar ese campo y prevenir cualquier otro valor que se encuentre en el mismo. Casos similares a este se repiten en diversas partes del protocolo, como la encriptación inicial, o incluso los códigos que identifican cada tipo de paquete. Cada uno de estos campos puede ser osificado por un middlebox. En respuesta a esto, QUIC v2 cambia todos estos campos para que utilicen valores distintos a la versión 1 con el objetivo de reducir el riesgo de osificación.

2.6.3. GREASE: Generate Random Extensions And Sustain Extensibility

A pesar de los todos los cambios y mejoras presentados hasta ahora, QUIC no es el único protocolo que cuenta con herramientas para combatir la osificación. Una de las técnicas más utilizadas es aquella que se conoce como GREASE.

2.6.3.1. GREASE en TLS

GREASE es un acrónimo de *Generate Random Extensions And Sustain Extensibility* (Generar extensiones aleatorias y mantener extensibilidad). Esta técnica surge originalmente en el contexto de TLS y busca combatir aquellas implementaciones que rechazan valores desconocidos en una comunicación. [42]

Un ejemplo de esto en TLS es durante el handshake, donde el cliente envía una lista de cifrados soportados y el servidor selecciona cuáles utilizar. Si el servidor rechazara valores desconocidos, al aparecer nuevos tipos de cifrados, este fallaría y sería necesario actualizar su implementación. No obstante, un servidor que ignora valores que no conoce, es capaz de seguir operando mientras nuevos cifrados son introducidos.

La técnica de GREASE busca agregar valores aleatorios dentro del protocolo de manera que aquellos servidores que rechacen valores desconocidos se vean obligados a fallar. En el contexto de TLS, se reservan distintos rangos de valores dentro del protocolo para ser utilizados para GREASE. Estos están esparcidos de manera que una implementación no pueda generar fácilmente condiciones sobre ellos, pues el objetivo de estos es que sean tratados como un valor desconocido y no se les dé ningún tratamiento especial.

2.6.3.2. GREASE en el QUIC bit

Tal como se analizó en la sección 2.3.7, el QUIC bit es un campo fijo y expuesto que tiene el objetivo de ser utilizado para demultiplexar paquetes en una conexión donde conviven múltiples protocolos. Como consecuencia este corre un alto riesgo de ser osificado, pues un servidor o cliente podría rechazar un paquete ante un valor distinto. Luego, si se necesitara cambiar la semántica de este bit, no sería posible pues los paquetes no serían aceptados por ser inválidos.

En el RFC9287 se propone una extensión a QUIC que permite que el QUIC bit pueda tener el valor 0 y mantener su validez. De esta forma, se puede enviar valores aleatorios en este. [43] Más aún, este cambio habilita a darle una semántica nueva a este campo. Para esto, se podría agregar una segunda extensión que permita darle otro significado al QUIC bit.

Dado que a través del uso de la técnica GREASE es posible aceptar valores aleatorios en el QUIC bit, se reduce el riesgo de modificar el protocolo. No obstante, se pierde la posibilidad de aplicar el algoritmo de demultiplexado presentado en la figura 2.13. Por lo tanto, dependerá del caso de uso habilitar o no esta extensión.

2.6.4. QUIC Multipath

El concepto de multipath, originalmente planteado para TCP, consiste en permitir que una única conexión TCP utilice múltiples caminos dentro de red con el objetivo de aumentar el rendimiento y la redundancia. Este fue originalmente planteado en el año 2009 y se ha encontrado en estado experimental[13] hasta su estandarización en 2020 en el RFC8684. [40]

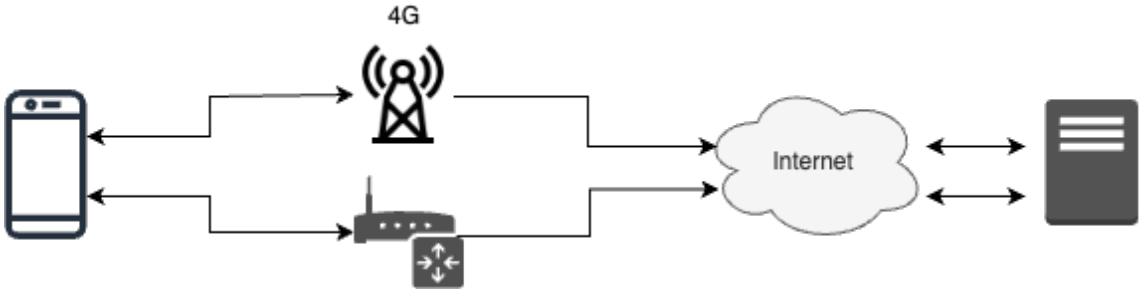


Figura 2.15: Esquema de una conexión multipath desde de un dispositivo móvil utilizando WiFi y 4G.

Actualmente, se está trabajando en llevar las funcionalidad de multipath al contexto de QUIC, de manera que, al igual que en TCP, una única conexión pueda hacer uso de múltiples caminos de red. Para su implementación se busca reutilizar la mayor parte de los mecanismos definidos en la versión 1 de QUIC. No obstante, se introducen una serie de cambios y decisiones para poder dar soporte a múltiples caminos. [39] Por un lado, los mecanismos de control de detección de pérdidas y control de congestión pasan a estar dedicados a un camino. Esto también tiene como consecuencia la necesidad de mantener mediciones de RTT por camino. A su vez, para identificar un camino de red, se introduce

[13] En el año 2013, Multipath TCP fue estandarizado experimentalmente en el RFC6824.

un 4-tupla formada por IPs y puertos de origen y destino. Esto se utiliza para limitar que exista un único ID de conexión para ese camino. Adicionalmente, es importante diferenciar la creación de nuevos caminos de red de una migración de conexiones, pues estos podrían confundirse fácilmente. Multipath QUIC diferencia estos casos pues en una migración se observara un cambio en la 4-tupla sin un cambio en los ID de conexión.

Las conexiones multipath suelen conllevar una complejidad más alta que las versiones que no lo implementan. Aparte de las características analizadas hasta ahora, las implementaciones multipath (tanto en TCP como QUIC) incluyen un mecanismos de descubrimiento y manejo de caminos. Es decir, cómo se encuentran nuevos caminos y se toman las decisiones de darlos de alta o baja. La especificación actual de multipath QUIC deja esta responsabilidad a la aplicación que está implementando el protocolo. Así mismo sucede con el planificador de paquetes (scheduling). Este es el encargado de tomar la decisión del camino a utilizar para cada paquete. Por el momento la extensión multipath de QUIC no propone ningún algoritmo a utilizar sino que se apoya en el trabajo ya realizado para multipath TCP. [39]

2.6.5. Balanceo de Carga en QUIC

En la sección 2.3 se analizaron las conexiones en QUIC y cómo estas hacen uso de los IDs de conexión para poder ser identificadas y sobrevivir a cambios que sucedan en capas inferiores, como podrían ser cambios de IP o puerto. A pesar de todos los beneficios asociados a estas funcionalidades, esto no es compatible con el funcionamiento de los balanceadores carga tradicionales de capa 4[14]pues estos están diseñados para rutear paquetes en base a su IP y puerto.

Actualmente, existe un draft (QUIC-LB) para desarrollar un mecanismo que permita a un balanceador de carga utilizar IDs de conexión QUIC para rutear paquetes QUIC. [44] Una forma de lograr esto sería observar los IDs a medida que transitan por el balanceador e ir construyendo una tabla de qué relación guardan estos con su servidor de origen. No obstante, mantener esta tabla puede ser costoso y se corre el riesgo de rutear mal los paquetes cuando una conexión pasa a utilizar otro ID. QUIC-LB propone un algoritmo para codificar el identificador del servidor origen en el propio ID de conexión. No obstante para generar el identificador de servidor que sea entendible por todas las partes se requiere algún tipo de configuración compartida entre el balanceador de carga y los servidores. Este es denominado como un *agente de configuraciones*, y no está en el alcance de la especificación de QUIC LB.

2.6.5.1. Estructura del ID de conexión

Como fue mencionado en la sección anterior, QUIC-LB propone estructurar el ID de conexión para codificar información de ruteo en este. Para lograr esto se reserva el primer octeto de este para darle una nueva semántica.

Los primeros dos bits del ID de conexión están destinados para el rotado de configuraciones. Esto es fundamental, pues cuando el agente distribuye una nueva configuración

[14]Los balanceadores de carga de capa 4 (L4) son componentes (virtuales o físicos) encargados de distribuir tráfico entre múltiples servidores en base a la IP y puerto de origen. El uso de balanceadores de carga permite mejorar la disponibilidad y capacidad para escalar de un servicio. Al utilizar únicamente datos pertenecientes a la capa de transporte, reciben su denominación de balanceador de L4, pues no inspeccionan protocolo en capas superiores. No obstante, los balanceadores de capa 7 (L7) son su contraparte más avanzada que permite trabajar con protocolos de capa de aplicación.

a los servidores, se dará un periodo de transición entre que se deja de utilizar la versión previa para pasar a la actual. Estos bits permiten al balanceador de carga aplicar el ruteo del paquete con la configuración correspondiente. Adicionalmente, existe el valor reservado $0b11$ que se utiliza para especificar que se debe rutear en base a IP y puerto de origen y destino como lo haría un balanceador de capa 4 tradicional. En casos como este se recomienda deshabilitar la migración de conexiones pues puede generar inconsistencias.

Los últimos 6 bits del primer octeto, son utilizados para codificar el largo del ID de conexión (sin contar el primer octeto). Esto permite realizar optimizaciones en base a este ID[15], pues normalmente es variable en largo. No todos los servidores hacen uso de este campo, por lo que también pueden poner valores aleatorios.

Luego del primer octeto, el ID de conexión es utilizado por un lado para codificar el ID de servidor. Esto permite al balanceador de carga identificar el destino al cual debe enviar el paquete recibido. El resto del ID funciona como un nonce, que sirve para distinguir las conexiones que van a un mismo servidor.

Finalmente, el agente de configuración puede proveer un clave criptográfica para proteger el identificador del servidor de manera que un observador no pueda relacionar los IDs con su destino. No obstante, esta configuración es opcional.

2.6.5.2. Proceso de balanceo

El proceso de balanceo de carga propuesto por QUIC-LB está compuesto por los tres componentes que interactúan en la figura 2.17.

El agente asigna un ID a cada uno de los servidores disponibles y provee las configuraciones correspondientes. Estos estructurarán sus IDs de conexión de la forma explicada en la sección 2.6.5.1 para codificar el identificador asignado. A su vez, el agente también provee una configuración al balanceador de carga de manera que este último pueda mapear cada ID de servidor a un destino. En particular, el agente debe proveer la configuración al balanceador antes que al servidor. De esta forma, evitamos que los servidores generen IDs que no se puedan rutear durante un rotado de la misma. En caso de utilizar una clave criptográfica para proteger el ID de conexión, esta también debe proveerse tanto al

[15]La especificación de QUIC-LB da como ejemplo los servidores que utilizan dispositivos criptográficos físicos para aumentar el rendimiento. Estos pueden indexar las claves en base al ID de conexión. Al tener codificado el largo del ID en los paquetes, es posible obtenerlo más rápidamente, pues este puede variar en su tamaño.

```
QUIC-LB Connection ID {
    First Octet (8),
    Plaintext Block (40..152),
}

First Octet {
  Config Rotation (2),
  CID Len or Random Bits (6),
}

Plaintext Block {
    Server ID (8..),
    Nonce (32..),
}
```

Figura 2.16: Estructura del ID de conexión según la especificación QUIC-LB. [44]

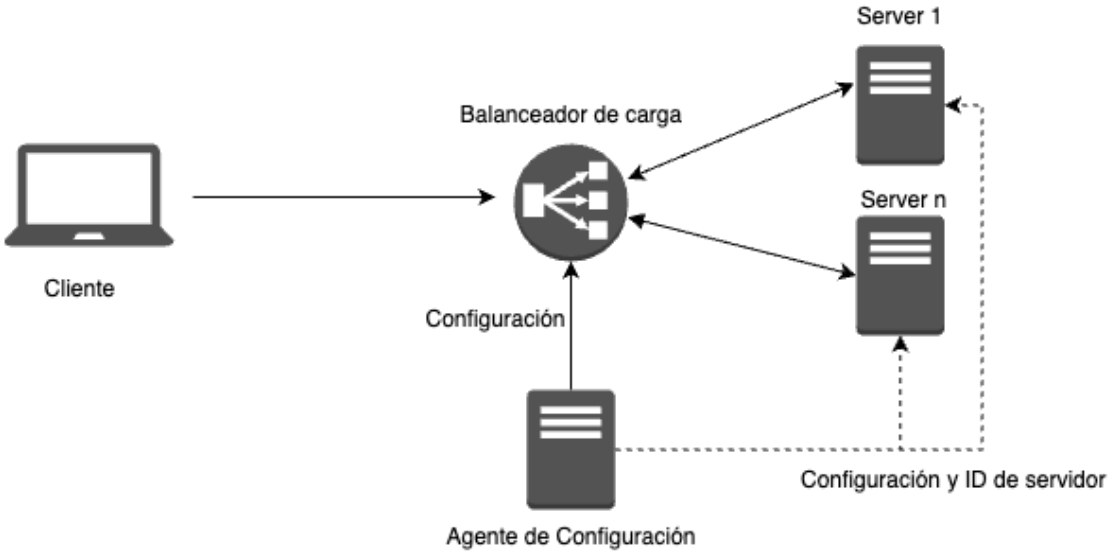


Figura 2.17: Esquema de balanceo de carga en base a IDs de conexión según la especificación QUIC-LB

balanceador como a los servidores.

Al recibir un paquete, el balanceador de carga se encuentra en condiciones de analizar los primeros octetos para extraer el ID del servidor y rutear el paquete al destino correspondiente. Una vez hecho esto, el balanceador puede asociar el ID de conexión o la 4-tupla de IPs y puertos de origen y destino con la decisión de ruteo. Esto permite no tener que extraer el identificador de servidor por cada paquete hasta que el ID de conexión o la 4-tupla cambien. No obstante, aunque este enfoque presenta algunas ventajas, hay algunas consideraciones importantes en la implementación de este método. Los balanceadores no pueden detectar el frame $CONNECTION\_CLOSE$ para indicar el final de una conexión (pues este está encriptado), por lo que dependen de un temporizador para eliminar el estado de la conexión. Esto puede causar problemas de ruteo si una conexión termina y se libera una dirección IP y un puerto, y otra conexión migra a esa IP y puerto antes de que se agote el tiempo de espera. Un tiempo de espera corto reduce la probabilidad de ruteo incorrecto, pero también aumenta la posibilidad de eliminar prematuramente el estado de la conexión.

Aunque los servidores generan IDs de conexión estructurados, al comienzo de una conexión QUIC, se utilizan IDs generados por el cliente, pues el servidor todavía no ha generado ningún paquete. Es de esperar que los IDs de conexión generados por el cliente no cumplan con la estructura de propuesta por QUIC-LB o contengan valores inválidos. En estos casos, el balanceador de carga debe descartar aquellos paquetes que usen el header corto. Para los que utilizan el header largo, se debe pasar a utilizar un algoritmo de balanceo que no requiera de coordinación con el servidor.

Capítulo 3

El protocolo HTTP/3

Como mencionamos en el capítulo anterior, QUIC surge con objetivo de mejorar el rendimiento de HTTP/2. La versión 3 de HTTP surge del esfuerzo del grupo de trabajo de QUIC en mapear la semántica de HTTP/2 sobre el protocolo QUIC. Inicialmente este mapeo se conocía como HTTP-over-QUIC. y finalmente pasó a llamarse HTTP/3, luego de un consenso con el grupo de trabajo de HTTP. El diseño de HTTP/3 reutiliza una buena parte del diseño HTTP/2 adaptándola al transporte en QUIC. Al hacer esto HTTP comienza a contar con funcionalidades introducidas en QUIC como 0-RTT, Multiplexado sin Head-of-line blocking, y migración de conexiones. Estas no hubiesen sido posibles de llevar a cabo si estuviéramos usando TCP. Por otro lado, HTTP/3 cambia el funcionamiento de ciertos aspectos del protocolo con respecto a sus predecesores para adaptarse mejor al esquema de comunicación de QUIC. Un claro ejemplo de esto es el protocolo de compresión de headers. En HTTP/2 se introduce HPACK que funciona asumiendo la entrega ordenada de los datos. Dado que esto no es garantía en QUIC, se crea una nueva versión del protocolo llamada QPACK [21] que resuelve este problema.

3.1. Mapeo con QUIC

Tal como se vio en el capítulo anterior, QUIC introduce su propio manejo de streams y multiplexado. Esto es similar al manejo de frames presente en HTTP/2. No obstante, en el caso de este último, las restricciones impuestas por TCP resultan en bloqueos HOL entre frames. QUIC supera esta limitación proveyendo confiabilidad y control de congestión en toda la conexión. HTTP/3 delega gran parte de su capa de framing a QUIC. A su vez, también depende de QUIC para proveer confidencialidad e integridad dado que este incluye TLS 1.3 en la capa de transporte.

3.1.1. Mapeo de Streams

Al igual que en HTTP/2, en HTTP/3 las interacciones entre cliente y servidor están basadas en streams HTTP. Un pedido HTTP es enviado en un request stream HTTP que está mapeado sobre un client stream bidireccional de QUIC. Cada request stream puede contener un único pedido HTTP. No obstante, el servidor puede enviar cero o más respuestas intermedias para ese pedido hasta enviar un respuesta HTTP final. En HTTP/3, el intercambio pedido/respuesta consume todo el ciclo de vida de un stream QUIC.

Similarmente, para interacciones server push, el servidor inicia un stream QUIC, unidireccional en este caso. Y al igual que en las interacciones donde hay un pedido, el servidor puede enviar cero o más respuestas intermedias seguidas por una respuesta final.

3.2. Descubrimiento y Creación de Conexiones

Un detalle muy interesante sobre el uso de HTTP/3, surge en el creado de una nueva conexión. Al igual que las versiones anteriores del protocolo para identificar un recurso se hace a través de un proceso conocido como acceso autoritativo. En todas las versiones de HTTP se usa una URI para identificar un recurso. El proceso para determinar si el acceso está permitido para una URI está definido por el esquema (URI Scheme). [26] En el caso de HTTP se utilizan los esquemas "http" y "https".

A pesar de que HTTP es independiente del protocolo de transporte utilizado, el esquema "http", por definición debe recibir conexiones TCP en el host y puerto indicado para asociar autoridad con el servidor de origen. [26]. Dado que HTTP/3 no utiliza TCP, no es posible utilizarlo para acceder a un recurso con esquema "http". No obstante, el servidor origen puede exponer en su respuesta la existencia de otro servicio que soporte HTTP/3. Para esto se puede usar la extensión del protocolo ALTSVC [27]. Esta consiste en devolver un header *Alt-Svc* (o un frame *ALTSVC* en HTTP/2) con el token *h3* indicando dónde se encuentra el servicio alternativo HTTP/3.

En el caso de un recurso con una URI "https", no se requiere de una conexión TCP para asociar autoridad, si no que esta se basa en la validación del certificado del servidor durante el handshake TLS. [26] Por lo tanto, es posible establecer una conexión QUIC para luego enviar un pedido HTTP. [18] No obstante, si el servidor no soporta QUIC o hubiese algún middlebox que no permitiera que la conexión se establezca correctamente, nos encontraríamos con un problema de conectividad. Una alternativa a esto, es abrir una conexión TCP con TLS y notificar la existencia de HTTP/3 como servicio alternativo. Esto es posible a traves de *ALPN* (Application-Layer Protocol Negotiation), una extensión del protocolo TLS que permite negociar el protocolo a utilizar por encima de la conexión segura. Al igual que en *ALTSVC*, el token *h3* es utilizado para denotar la existencia de un servicio HTTP/3.

En el caso de un recurso con una URI "https", no es necesario establecer una conexión TCP para asociar la autoridad. Esta asociación se basa en la validación del certificado del servidor durante el handshake TLS [26]. Por lo tanto, es posible establecer una conexión QUIC primero y luego enviar una solicitud HTTP [18]. No obstante, si el servidor no es compatible con QUIC o si algún middlebox impide que la conexión se establezca correctamente, se podrían observar problemas de conectividad. Como alternativa, es posible abrir una conexión TCP con TLS y notificar la disponibilidad de HTTP/3 como servicio alternativo. Esto se logra a través de *ALPN* (Application-Layer Protocol Negotiation), una extensión del protocolo TLS que permite negociar el protocolo a utilizar en la conexión segura. [45] Similar a *ALTSVC*, el token *h3* se utiliza para indicar la disponibilidad de un servicio HTTP/3. Otra alternativa a este escenario consiste en crear una conexión TCP y QUIC al mismo tiempo para quedarse con la que sea más conveniente.

La necesidad de utilizar TCP y TLS para descubrir la existencia de un servicio HTTP/3 termina resultando contraproducente y tiene un impacto directo en la cantidad de round trips necesarios para establecer una nueva conexión. Afortunadamente, existe un trabajo en curso para abordar este problema de una forma mas eficiente. Este consiste en exponer

información sobre los servicios y protocolos disponibles (utilizando tokens ALPN) a través de un nuevo registro DNS de tipo HTTPS. [46]

3.3. Comparación con HTTP/2

HTTP/2	**HTTP/3**
Envió de requests en paralelo a través de multiplexado en una única conexión TCP.	Envió de requests en paralelo a través de multiplexado en una única conexión UDP.
Introducción de funcionalidades de streams y control de envió de datos.	Streams y control de envió de datos es delegado a QUIC.
Bloqueo HOL en toda la conexión ante la perdida de un paquete.	La perdida de un paquete bloquea únicamente a los streams involucrados.
Soporta esquemas “https” y “http”. Aunque este ultimo es poco utilizado.	Soporta únicamente el esquema “https”
Soporta TLS con versión mayor o igual a 1.2	Soporta únicamente TLS 1.3.
Compresión de headers con garantía de envió en orden HPACK.	Compresión de headers con soporte para envió fuera de orden QPACK.

Tabla 3.1: Diferencias principales entre los protocolos HTTP/2 y HTTP/3

Según la especificación de HTTP/3, durante el diseño del protocolo, la similaridad con HTTP/2 fue preferible pero no obligatoria. En general, ambos protocolos divergen cuando es necesario aprovechar una nueva funcionalidad de QUIC que no existía previamente.

Dado que QUIC implementa funcionalidades relacionadas con el manejo de streams equivalentes a las de HTTP/2, varias de las características de este relacionadas con el framing de este ultimo dejan de ser necesarias y quedan deprecadas en favor de QUIC. Esto resulta en varios tipos de frames HTTP y mecanismos de control de flujo que ya no se utilizan.

Muchas de las diferencias entre las versiones de HTTP surgen de la garantía de entrega ordenada de todo los frames en HTTP/2, En el caso de QUIC, el orden solo se puede garantizar a nivel stream. Un resumen sobre las diferencias principales de cada protocolo puede observarse en la tabla 3.1.

Es importante notar que una conexión HTTP/3 incurre en un mayor uso de recursos en comparación con sus predecesores. Esto se debe a que para operar la conexión es necesario mantener un mayor estado.[18] Esto puede desincentivar su uso en sistemas cuyos recursos son muy limitados, como ciertos dispositivos IoT, aunque existen alternativas para superar este problema.

3.4. Compresión de Headers

En la sección 3.1, observábamos como muchas de las funcionalidades originalmente introducidas en HTTP/2 son fácilmente mapeadas sobre QUIC para el desarrollo de

HTTP/3. No obstante, esto no aplica para la compresión de headers. Esta funcionalidad es importante dado que, en la práctica, los requests HTTP suelen llevar información redundante en los headers.[19] El uso de compresión de headers puede resultar en mejoras notorias en el uso de ancho de banda y latencia.[20]

En HTTP/2, se introduce el esquema de compresión HPACK. Este es un proceso donde se mantiene un contexto de compresión y descompresión para toda la conexión y se necesita que los bloques de headers sean transmitidos en orden y de forma continua. HPACK mantiene una tabla con dos secciones, la estática y la dinámica. Ambas tienen un índice, el nombre del header y su valor asociado. La tabla estática tiene una lista de headers predefinidos por convención y no puede ser editada. Por otro lado, la tabla dinámica comienza vacía donde termina la parte estática y se irá completando en base a los headers enviados durante la conexión. Si la tabla dinámica se llenara, se irán desalojando las entradas más antiguas de la tabla. Para facilitar este proceso, las entradas nuevas son agregadas al comienzo de la tabla dinámica. Para esto, será necesario que los headers lleguen en orden. A la hora de enviar headers, si se encuentran el header y su valor en la tabla podemos referenciarlos usando únicamente el índice de la entrada de la tabla. Si tuviéramos una entrada con el nombre del header pero no su valor, podríamos referenciar el nombre con su índice y enviar el valor usando el menor en entre una codificación *Huffman*[28] [1] y el contenido original. Cuando enviamos un header nuevo tenemos la posibilidad de indicarle si queremos indexarlo, o no.

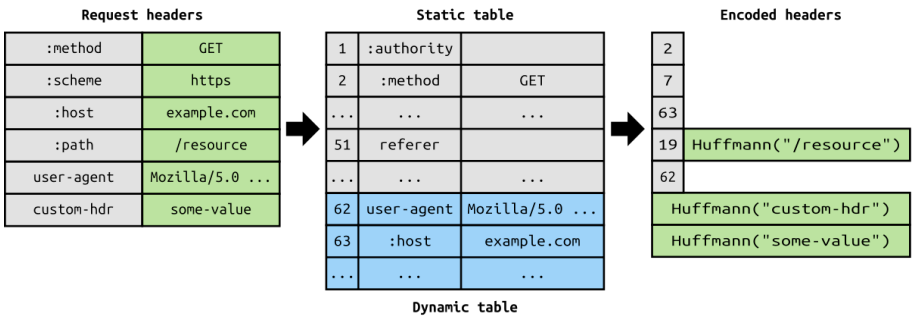


Figura 3.1: Compresión de headers HPACK en HTTP/2.

Como fue mencionado en las secciones anteriores, QUIC no garantiza el envío de paquetes en orden. Por lo tanto, la implementación de HPACK no será posible sin introducir head of line (HOL) blocking. Es por esto que en HTTP/3 se introduce el esquema QPACK, que toma las características más valiosas de su predecesor y las adapta para un envío de paquetes fuera de orden, buscando un *trade off* entre reducir el bloqueo HOL y mantener el nivel de compresión. [21]

Algunas diferencias introducidas en QPACK, son la separación del direccionamiento de la tabla dinámica y la tabla estática (en HPACK hay una única tabla y la parte dinámica comienza a partir de la posición 61). Para distinguir qué tabla se está usando se introduce un bit adicional. QPACK introduce facilidades para duplicar headers eficientemente y permitir a los headers más usados mantenerse arriba de la tabla dinámica. Otra introducción importante en QPACK, es el uso de streams unidireccionales entre cada parte (el encoder

[1] La especificación de HPACK, define su propio codificación basada en estadísticas de uso sobre una muestra de headers HTTP.

y decoder). Estos se usan para actualizar las tablas desde el encoder (codificador) al decoder (decodificador) y para enviar ACKs. De esta forma, en QPACK sólo se podrá usar referencias cuando estas hayan recibido un ACK de parte del decoder. QPACK también agrega la posibilidad de utilizar entradas con índices relativos para facilitar el envío fuera de orden.

3.5. Consideraciones Finales

Durante este capítulo, observamos cómo gran parte del funcionamiento de HTTP/3 es un mapeo de la semántica y funcionalidad de HTTP/2 sobre el protocolo QUIC. Esto resulta fácil de comprender cuando se considera que HTTP/3 comienza inicialmente como un proyecto para mapear la semántica de HTTP/2 sobre QUIC. Es decir que su esfuerzo inicial no consistió en desarrollar un nuevo protocolo con nuevas funcionalidades, sino que fue transportar la semántica existente de HTTP sobre otro protocolo. Este contexto también explica por qué hubo tan poco tiempo entre la creación de HTTP/2 y HTTP/3, con 7 años de diferencia entre sus aprobaciones con respecto al último salto de versiones donde pasaron 16 años hasta la aprobación de un nuevo estándar. [2]

Otro punto importante a considerar es el impacto de lo estudiado en la sección 3.2, sobre el descubrimiento y creación de conexiones. Esto tiene como consecuencia que, al crear una nueva conexión hacia un servicio que no sea conocido, sea necesario crear primero una conexión TCP con TLS para luego descubrir si existe un servicio HTTP/3, y finalmente establecer una conexión con este. Este mecanismo de descubrimiento de servicios alternativos resulta ineficiente en este caso de uso y, como resultado, aún se depende del stack TCP/TLS hasta que obtener una solución más efectiva, como los registros DNS de tipo HTTPS.

[2] Aquí considerando las fechas de aprobación de los RFCs correspondientes a cada protocolo. HTTP/1.1 en 1999, HTTP/2 en 2015, y HTTP/3 en 2022.

Capítulo 4

Metodología de Experimentación

En esta sección pasaremos a explicar la metodología utilizada para la realización de experimentos. Comenzaremos discutiendo los racionales de la metodología a utilizar y los racionales que llevan a esta. Luego, especificaremos el entorno de pruebas y las herramientas utilizadas para la generación de datos.

4.1. Consideraciones

A pesar de la gran cantidad de beneficios que puede tener un nuevo protocolo de red como lo es HTTP/3, en la práctica, existen muchos factores que pueden afectar el rendimiento o practicidad de los protocolos. Los entornos de prueba sintéticos son útiles para observar cómo el protocolo se comporta en escenarios específicos y controlados. El problema con este tipo de pruebas es que no necesariamente reflejan la experiencia real de un usuario. Es por esto que enfocaremos la experimentación hacia implementaciones de QUIC y HTTP/3 que sean productivas y ya existentes. Esto significa hacer experimentos contra servidores reales que hoy implementan QUIC y HTTP/3, lo cual nos permitirá intentar ver si hay una mejora en el rendimiento.

Al experimentar con implementaciones productivas perderemos cierto control sobre el entorno de pruebas y el estado de la red, pero al mismo tiempo, podremos tener alguna evidencia empírica del rendimiento de QUIC en producción. Adicionalmente, si consideramos que cualquier servidor que soporte HTTP/3 debería soportar las versiones anteriores por retro-compatibilidad, tendremos suficiente información para comparar el rendimiento de las distintas versiones del protocolo. La lista de servidores se encuentra en la tabla 4.1. Esta incluye los dominios y ubicaciones de cada servidor con el cual se experimento durante esta tesis.

Esta lista de servicios incluye algunos de los sitios web más importantes de Internet. Por esto mismo, si detectamos una mejora, aunque pequeña, podríamos ver un gran impacto al considerar la cantidad de tráfico total.

Dado que no tendremos acceso a datos sobre el funcionamiento de los servicios con los que estaremos trabajando, la cantidad de cosas que podemos medir se verá reducida. Continuando con la idea de ver el impacto que podría tener QUIC y HTTP/3 en producción, nos enfocaremos principalmente en medir la latencia *end-to-end* de cada *request* HTTP que hagamos. Esto nos permitirá comparar versiones de HTTP y tener una idea de la experiencia que estos le podrán ofrecer a un usuario final.

Dominio Servidor	Ubicación Geográfica
www.google.com	Ciudad de Buenos Aires, Argentina
www.facebook.com	Ciudad de Buenos Aires, Argentina
www.litespeedtech.com	Virginia, Estados Unidos
www.instagram.com	Ciudad de Buenos Aires, Argentina
www.youtube.com	Ciudad de Buenos Aires, Argentina
shops.myshopify.com	Ontario, Canada
quic.aiortc.org	Dublin, Irlanda

Tabla 4.1: Servidores que usaremos para comparar latencias entre HTTP/3 y sus versiones anteriores. Nótese que aunque ninguno de estos servidores es originario de Argentina, la mayoría tiene presencia en dicho país a través de su red de contenido (CDNs).

4.2. Implementaciones de QUIC y HTTP/3

Una de los grandes cambios introducidos en QUIC es su implementación en el user space. A diferencia de TCP, que está totalmente implementado a nivel kernel y luego es utilizado por las distintas implementaciones de HTTP de cada lenguaje, en el caso de QUIC y HTTP/3, ambos protocolos deben ser implementados y únicamente dependen de la implementación de UDP del kernel. Como ya mencionamos, esto tiene beneficios interesantes como un incremento en la velocidad de desarrollo, pero al mismo tiempo, tiene como consecuencia la necesidad de una implementación de QUIC por lenguaje de programación. Es de esperar que, dado que QUIC recientemente terminó de ser estandarizado, ningún lenguaje cuente con una implementación en su biblioteca estándar Por esta razón, tendremos que trabajar con las implementaciones externas que se encuentren disponibles para cada lenguaje. El grupo de trabajo de QUIC ofrece una lista con varias opciones.[1]

Por cuestiones de practicidad, no podemos probar todas las bibliotecas existentes, por lo tanto tomamos la decisión de enfocarnos en las siguientes implementaciones:

Nombre	Lenguaje
quic-go[2]	Go Lang
aioquic[3]	Python
jetty[4]/quiche[5]	Java

Tabla 4.2: Implementaciones de QUIC y HTTP/3 a utilizar.

Dado que QUIC y HTTP/3 fueron estandarizados recientemente, un fenómeno que se da en estas librerías es que no todas implementan las mismas versiones del protocolo. Más aún, estas implementan varios drafts previos a la aprobación de los RFCs 9000[9] y 9114[18] (QUIC y HTTP/3 respectivamente). No obstante nos enfocaremos en las versiones aprobadas de cada estándar pues el soporte de drafts anteriores debería tender a desaparecer con el tiempo.

[1] `https://github.com/quicwg/base-drafts/wiki/Implementations`

[2] `https://github.com/lucas-clemente/quic-go`

[3] `https://github.com/aiortc/aioquic`

[4] `https://github.com/eclipse/jetty.project`

[5] La implementación de QUIC y HTTP/3 de Jetty está desarrollada sobre la biblioteca Quiche de Cloudflare. `https://github.com/cloudflare/quiche`

4.3. Entorno de trabajo

Como ya fue mencionado, el foco de la experimentación estará orientado a casos reales que puedan reflejar la experiencia de un usuario final o una aplicación promedio. Por esto mismo, hace sentido realizar los experimentos desde un equipo de trabajo sin poner muchas restricciones sobre el entorno utilizado. El equipo utilizado para realizar los experimentos será el siguiente:

- Apple MacBook Pro 2020
  - **CPU:** Apple M1
  - **RAM:** 16GB
  - **SO:** MAC OS Big Sur

Este equipo cuenta con todas las configuraciones por defecto del kernel y funciona sobre una red Wi-Fi 5 de 5.8GHz con 100$Mbps$ de bajada y 20$Mbps$ de subida. Todas las mediciones fueron realizadas desde la Ciudad de Buenos Aires, Argentina.

4.4. Armado de los experimentos

Para el armado de los experimentos se desarrollaron implementaciones que utilizan la bibliotecas mencionadas en la sección anterior y realizan mediciones. Adicionalmente, algo que nos interesa es poder comparar con versiones anteriores de HTTP, por lo tanto, también tendremos que poder implementar las bibliotecas de HTTP versión 1 y 2 correspondientes a cada lenguaje.

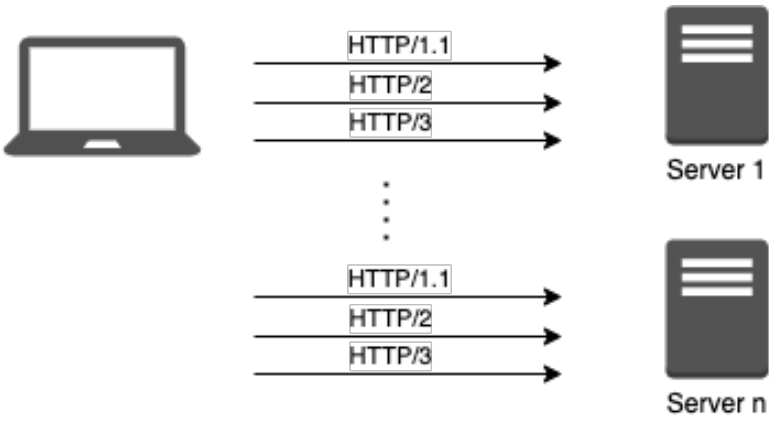
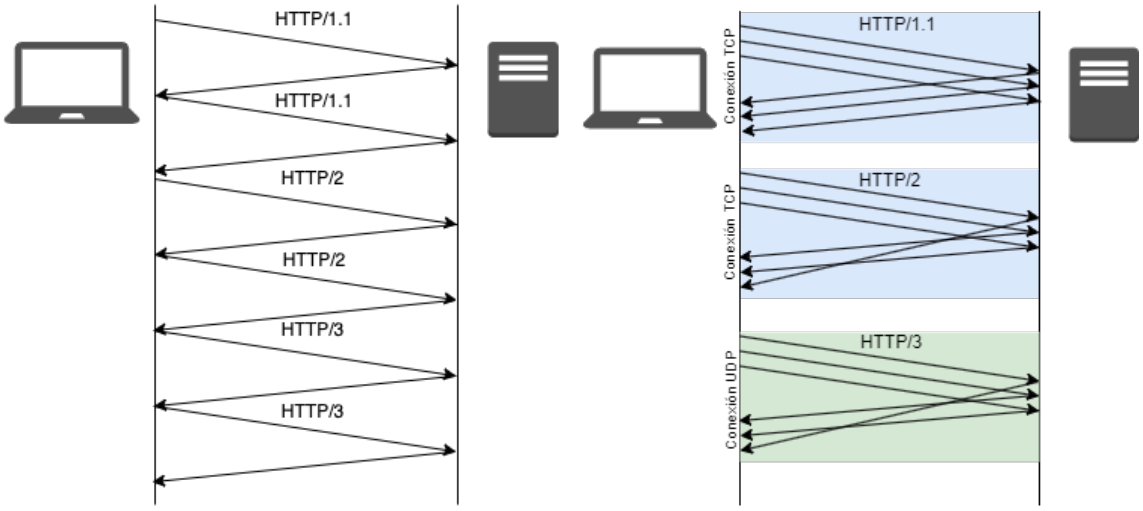


Figura 4.1: Esquema para el desarrollo de los experimentos.

En la figura 4.1 podemos ver cómo será la metodología de comparación utilizada en las distintas mediciones realizadas. Para cada uno de los requests HTTP realizados nos interesará saber la duración total. Esto lo utilizamos como medida para comparar el rendimiento de los protocolos por cada lenguaje. Cabe destacar que hacer un único request para medir la latencia no es una buena idea. Variaciones en la red pueden resultar en mediciones con mucha varianza. Por lo tanto, en la práctica, debemos realizar varios requests

por cada servidor y versión de protocolo. De esta forma podremos ver una distribución de la duración y sacar mejores conclusiones.



(a) Serie (b) Paralelo

Figura 4.2: Comparación de generación de requests HTTP

Al trabajar con múltiples requests HTTP es importante considerar cómo estos serán organizados. Durante este experimento trabajamos con requests en serie o concurrentes. En la figura 4.2 podemos observar un comparación gráfica de estas estrategias. Los requests en serie, son ejecutados en orden uno tras otro para cada protocolo, donde el final de uno resulta en el comienzo del siguiente. Por otro lado, los requests concurrentes son enviados uno tras otro sin esperar que uno termine para comenzar el siguiente. Una vez enviados, se esperan las respuestas de cada uno.

Capítulo 5

Resultados

En esta sección pasaremos a detallar los resultados de la experimentación realizada durante esta tesis. Comenzaremos detallando la comparación de latencia contra los servidores mencionados en la sección anterior. Luego discutiremos cómo el multiplexado que permite QUIC impacta en la latencia. Durante este capítulo se utilizaran las visualizaciones de tipo *box-plot* para representar y comparar las distribuciones de los resultados obtenidos. Estos nos permitirán visualizar fácilmente los los mínimos, máximos, y percentiles principales de cada distribución analizada.[47] A su vez, en la discusión de estos, se hará referencia a estos percentiles utilizando la notación Px, donde x es el percentil correspondiente. Por ejemplo, P25 denota el percentil 25.

Para facilitar la visualización e interpretación de los resultados en las secciones 5.1 y 5.2, estos se presentan sin los outliers detectados durante las mediciones. En el apéndice B se presentan estos mismos con sus outliers correspondientes.

5.1. Comparación de versiones de HTTP

A continuación presentamos los resultados de los experimentos realizados en el esquema presentado en la sección 4.4. Es decir que por cada lenguaje y protocolo realizamos 30 requests a cada servidor para medir la latencia y obtener una distribución por cada uno.

En la Figura 5.1 podemos observar resultados con una gran dispersión, en especial para el caso de HTTP/3. En la mayoría de los casos se puede distinguir que los percentiles más bajos (menores al $P25$) suelen ser comparables a los HTTP/2. Más aún, en los casos de Instagram, Facebook y Litespeedtech los mínimos de HTTP/3 son menores al resto. No obstante, la mediana de HTTP/3 suele ser peor en comparación con HTTP/2, e incluso con respecto a HTTP/1.1 para ciertos casos.

Este comportamiento se repite bastante en los servicios de mayor renombre como Google, Facebook, Instagram y Youtube. Estos resultados pueden tener varias explicaciones. Por un lado, dado que estamos considerando algunos de los sitios web más grandes, no es sorpresa que estos tengan un nivel de optimización que resulte en rendimientos muy similares para las versiones más nuevas de HTTP. Adicionalmente, es posible que este tipo de prueba no haya reflejado una situación donde los beneficios de QUIC puedan ser apreciados. Otro punto importante a considerar, es la madurez de las bibliotecas utilizadas. Dada la antigüedad de estos protocolos, las implementaciones de HTTP/1.1 y HTTP/2 tienen varios años más de desarrollo con respecto a HTTP/3.

Otro comportamiento que se vio de forma consistente en todos los casos fue la mejora

de HTTP/2 con respecto a HTTP/1.1. Esto tiene sentido pues si miramos la diferencia de cambios que con HTTP/2, hay varios cambios introducidos que mejoran el rendimiento, como por ejemplo el framing binario de los datos. Por otro lado, si consideramos los cambios de HTTP/2 con HTTP/3, los cambios suceden mayormente en el cambio del protocolo de transporte, mientras que las implementaciones de HTTP se mantienen bastante similares.

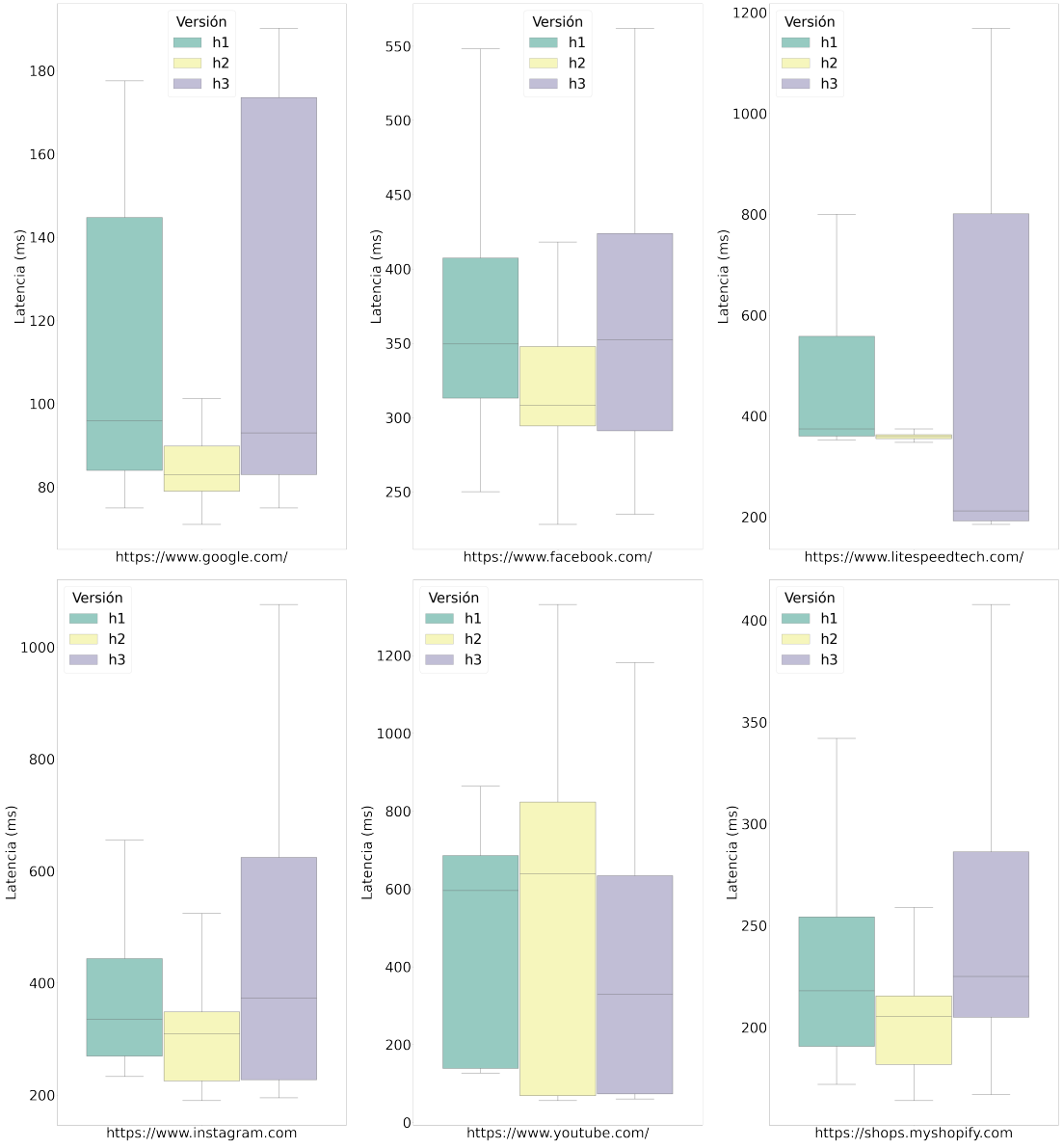


Figura 5.1: Comparación de latencia por cada servidor y versión del protocolo HTTP de las tres implementaciones utilizadas.

5.2. Comparación de implementaciones

En la sección 5.1 observamos las latencias para cada versión de HTTP. No obstante, es importante recordar que las mediciones fueron realizadas con tres implementaciones distintas de QUIC y HTTP/3 (tal como explicamos en la sección 4.2).

En la Figura 5.2 podemos observar el rendimiento de la implementación correspondiente a cada lenguaje. Los resultados fueron bastante similares en cada caso. La implementación en Go resultó ser la más rápida de las tres de manera consistente. Esta es seguida por la de Java y finalmente por la de Python. Las razones para estos resultados pueden tener diversas justificaciones, como la madurez de cada implementación, o los propios límites de velocidad intrínsecos a cada lenguaje. Estos resultados también sirven para explicar parte de la dispersión observada en la figura 5.1, pues podemos observar que en la mayoría de los sitios medidos tuvimos diferencias significativas entre cada implementación, siendo el caso más notorio el de Litespeedtech. Por otro lado, si observamos los resultados obtenidos en el caso de Facebook, los resultados fueron muchos más cercanos entre cada implementación, resultando en una distribución con menos varianza. Los resultados obtenidos para la implementación desarrollada en Go demostraron ser los menos variados y más veloces. A su vez, es importante observar como este funcionó en comparación con las versiones previas de HTTP.

En la Figura 5.3 observamos como la implementación en Go de QUIC y HTTP/3 resulta mejor en todos los casos con respecto a HTTP/1.1. Por otro lado, al comparar los resultados con los de HTTP/2, este último termina siendo superior en la mayoría de los casos con medianas y P75 superiores. No obstante, también se observan algunas excepciones como Facebook e Instagram donde las medianas para HTTP/3 son levemente mejores. Finalmente también se obtuvo un caso muy distinto al resto que es el de Litespeedtech donde los rendimientos para HTTP/3 son completamente superiores al resto y con una dispersión muy baja en comparación con el resto de los casos. Es importante considerar que LiteSpeed es un gran promotor de QUIC y HTTP/3, y utiliza estas tecnologías como parte en su propuesta de valor y estrategia de ventas. Por lo tanto, sería de esperar que sus servicios estén optimizados, o favorezcan, estas tecnologías sobre sus versiones anteriores.

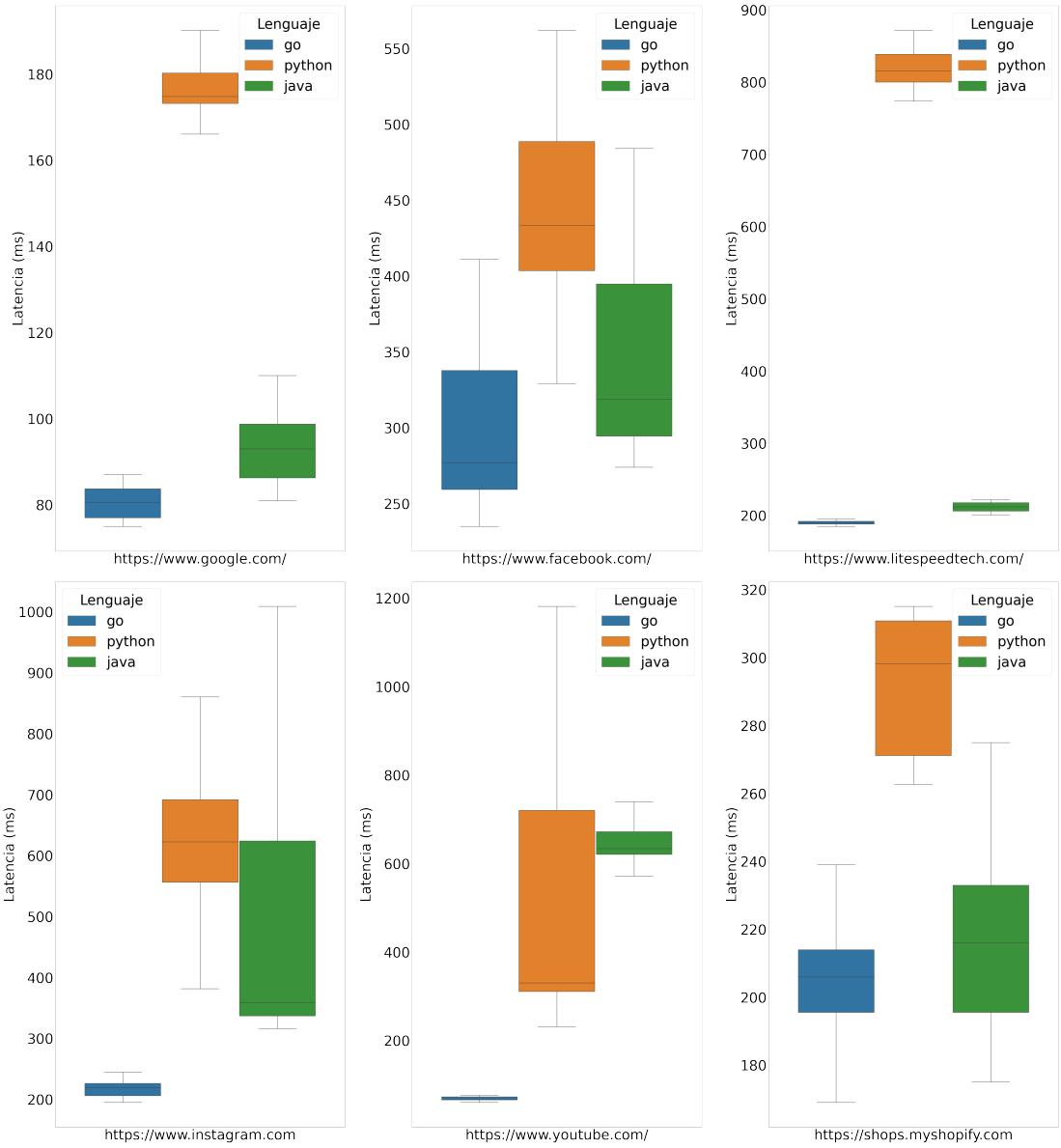


Figura 5.2: Comparación de latencia por cada servidor e implementación de QUIC y HTTP/3.

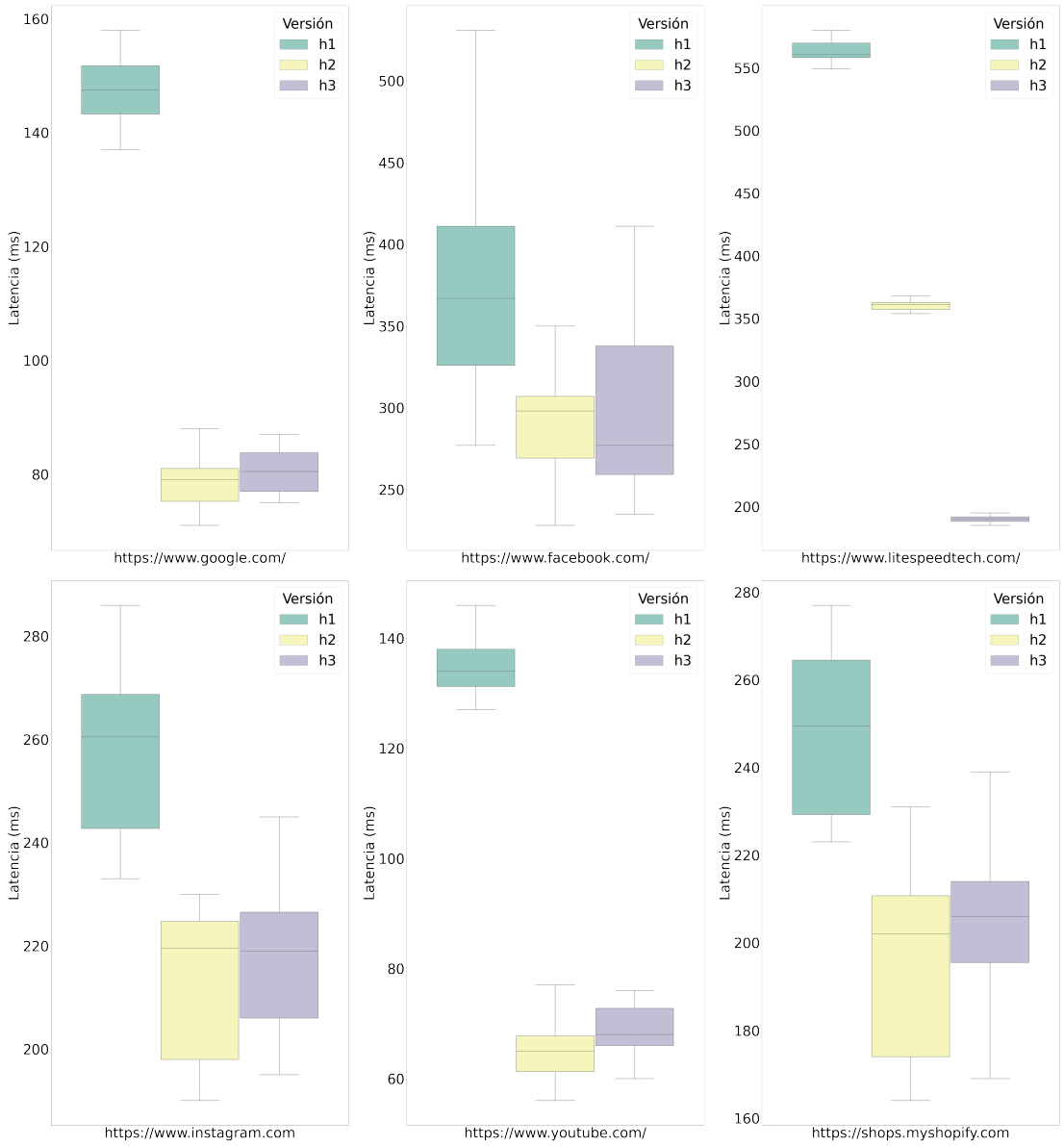


Figura 5.3: Comparación de latencia por cada servidor y versión del protocolo HTTP únicamente para las mediciones usando Go Lang.

5.3. Análisis de latencia y multiplexado

Uno de los cambios introducidos por QUIC en comparación con TCP es el concepto de streams y su multiplexado. Previamente esto debía implementarse en protocolos de capas superiores como es en el caso de HTTP/2. Como ya fue mencionado en la sección 2.2, esto puede traer algunos problemas como el head of line blocking. En el caso de HTTP/3, el concepto de streams y multiplexado sigue existiendo pero es delegado a QUIC para su implementación. En esta medición realizamos una comparación de las latencias para cada versión de HTTP. En este caso se generaron varios requests y se observó la distribución de las latencias tanto como la sumatoria del tiempo total que tomó realizar todos. Para simular un caso más similar a la realidad realizaremos una nueva medición con varios pedidos espaciados aleatoriamente y otra donde enviamos todos los pedidos juntos. Estas mediciones fueron realizadas tomando Google como destino. Dado los resultados del experimento anterior, y con el objetivo de simplificar la cantidad de variables del experimento, estas mediciones fueron solo realizadas únicamente con la implementación en Go, que mostró ser la más rápida en la mayoría de los casos.

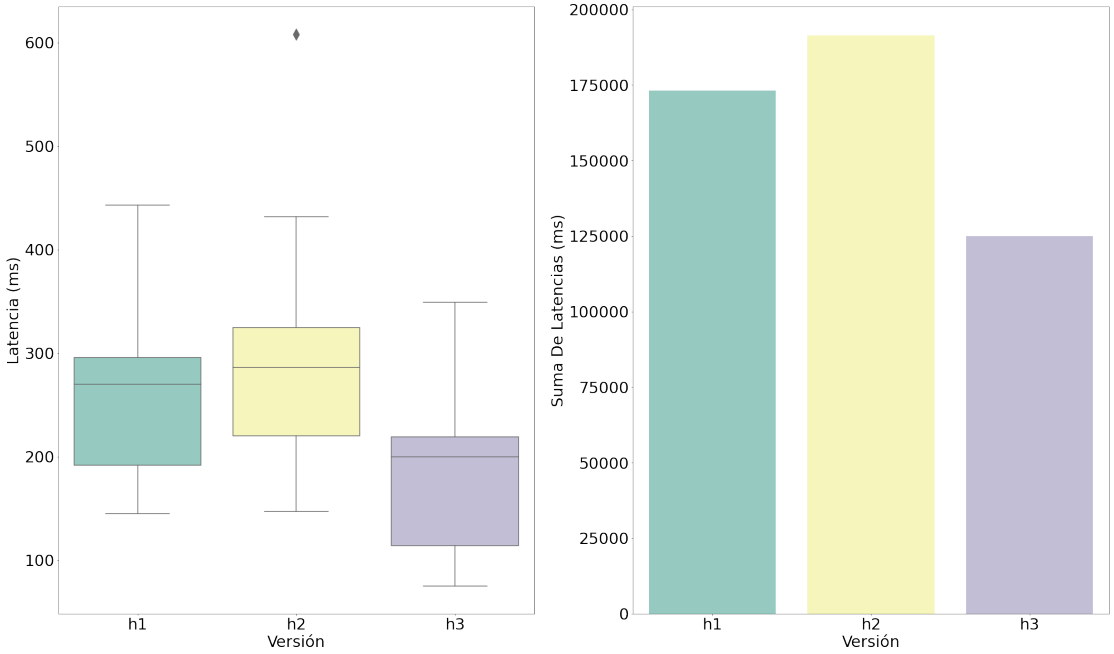


Figura 5.4: Comparación de latencia con requests espaciados aleatoriamente. Distribución de latencias (izq). Tiempo total (Der).

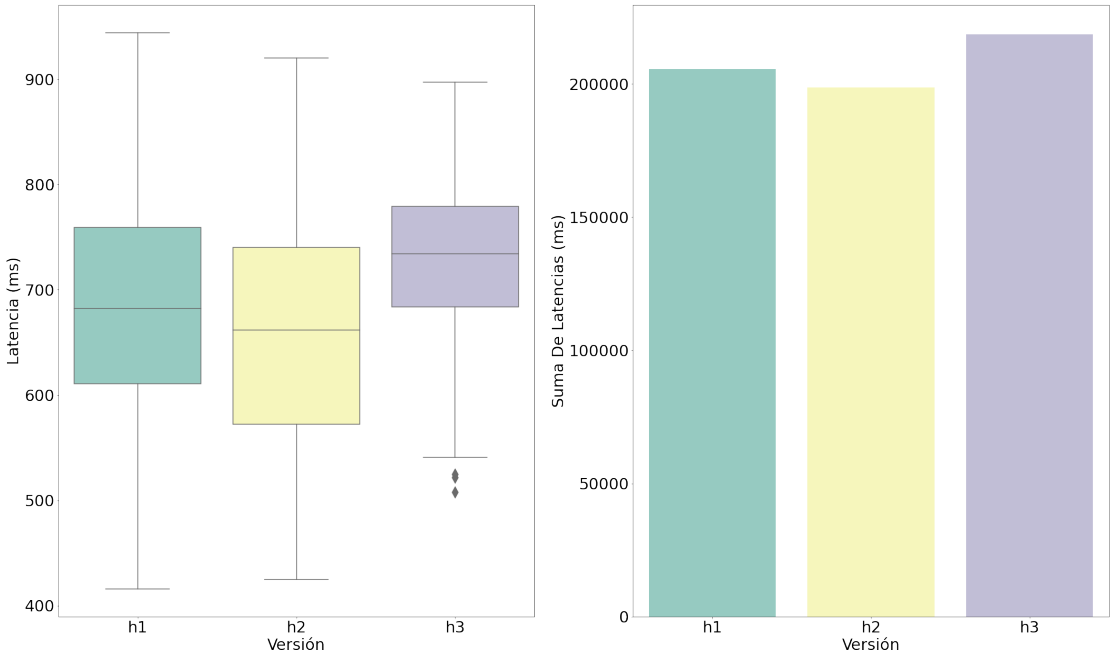


Figura 5.5: Comparación de latencia con requests concurrentes. Distribución de latencias (izq). Tiempo total (Der).

En la figura 5.4 se observa como HTTP/3 dio resultados notoriamente mejores al resto. Por otro lado, en la figura 5.5 el comportamiento fue contrario a lo esperado y se observa una latencia más alta en general. Este último resultado podría estar relacionado a una congestión ocasionada por la ejecución concurrente, pues QUIC debería poder alcanzar un uso más alto en una única conexión.

5.3.1. Recreado de conexiones

Algo que hasta ahora no consideramos en la experimentación realizada es el impacto de la reutilización de conexiones. En los resultados vistos hasta ahora, tanto en TCP como en QUIC, estamos inadvertidamente reutilizando las conexiones en cada request subsiguiente al primero. Para entender el impacto de este fenómeno podemos volver a realizar el experimento, pero forzando el recreado de la conexión tras cada request HTTP. Para simplificar la implementación y evitar posibles condiciones de carrera o reutilizaciones de conexión accidentales, realizaremos cada request de forma serial.

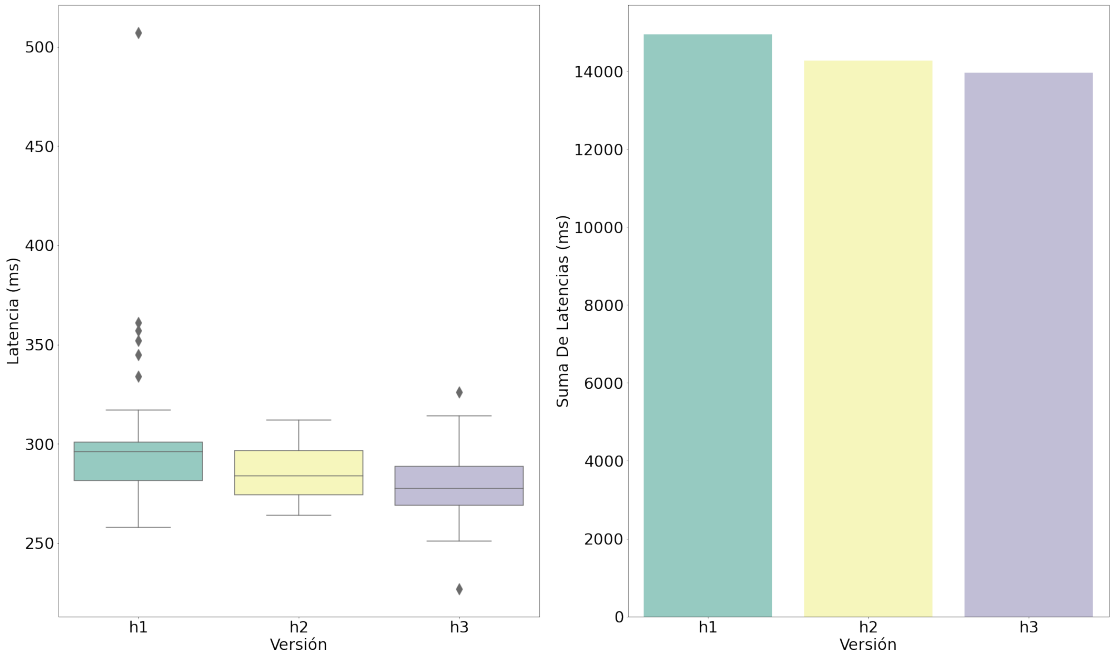


Figura 5.6: Comparación de latencia con recreado de conexiones. Distribución de latencias (izq). Tiempo total (Der).

En la Figura 5.6 podemos observar una leve mejora de HTTP/3 con respecto a sus versiones anteriores. Esto podemos atribuirlo a que en este caso se generan la condiciones correctas para el uso del mecanismo de *session resumption* de TLS 1.3 y el handshake 0-RTT de QUIC. Es importante recordar que el conjunto de ambos permite que en el caso de HTTP/3 sea posible enviar datos desde el primer paquete sin tener que volver negociar los parámetros de la conexión. En el caso de las versiones previas de HTTP, es posible reestablecer la sesión de TLS pero el handshake de TCP sigue siendo obligatorio y necesario para establecer la conexión.

5.3.2. Resultados con otro servicio (aiortc)

Una observación sobre los resultados previamente presentados es la gran dispersión que hay entre las latencias. Esto se repite para todos los protocolos. Esto puede explicarse por la existencia de CDNs y middleboxes que resulten en que no necesariamente estemos llegando siempre hacia los servidores de Google (o cualquier otro de los sitios presentados). Esto puede impactar los resultados observados, alterando las diferencias entre los tiempos de respuesta de cada protocolo.

Dado que no contamos con una forma de llegar a los servidores de Google (o cualquier sitio) salteando las CDNs, nos vemos obligados a buscar alternativas. Por ejemplo, el servidor quic.aiortc.org, es un sitio para hacer pruebas con QUIC mantenido por los creadores de aioquic (la implementación de Python). Este no aparenta tener una CDN (dado su escala tiene sentido). Por lo tanto, a continuación pasamos a repetir la última medición, pero esta vez usando el sitio quic.aiortc.org [1].

[1]Durante finales de 2022, al momento de escribir esta tesis y trabajar en las mediciones sobre

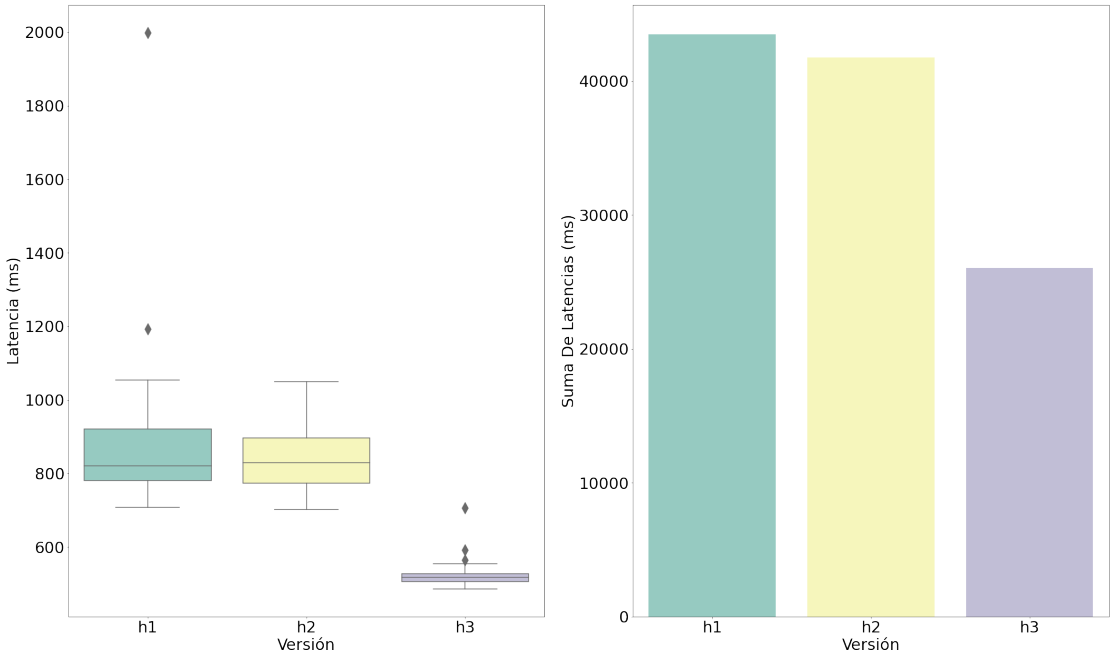


Figura 5.7: Comparación de latencia con recreado de conexiones hacia quic.aiortc.org. Distribución de latencias (izq). Tiempo total (Der).

En la figura 5.7 resultados podemos ver una ventaja significativa de HTTP/3 contra sus versiones previas. Más aún, también podemos ver una reducción en la variabilidad de las latencias. La falta de redes de distribución sumado al uso de 0-RTT Handshakes y el resumen de sesiones TLS resultan en una ventaja considerable y consistente por parte de HTTP/3. No obstante, esta medición se aleja de la premisa inicial de comparar latencias en el contexto de ambientes productivos, pues quic.aiortc.org es un servicio de prueba.

5.3.3. Resultados de aiortc en relación al tamaño de respuesta

Otro aspecto interesante que no fue explorado hasta ahora es la relación entre la latencia y el tamaño de la respuesta obtenida. Para ver esto podemos utilizar la utilidad ofrecida por quic.aiortc.org para especificar el tamaño de la respuesta devuelta. A continuación mostramos el promedio de latencia para tamaños de respuesta crecientes desde 10B hasta 5MB. Los resultados se presentan en dos variantes, con requests en serie y concurrentes.

quic.aiortc.org, el sitio se encontraba operativo. Durante principios de 2023 este fue dado baja.

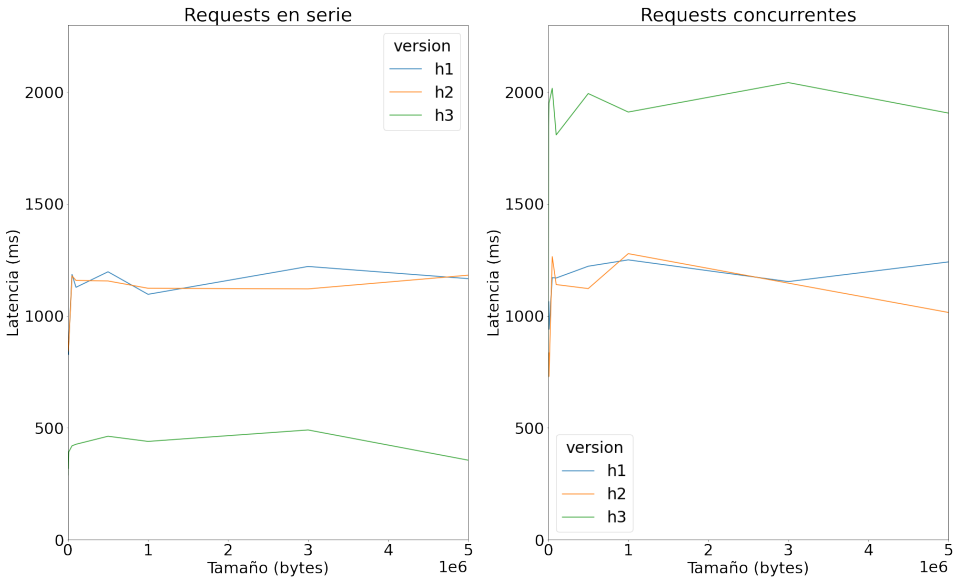


Figura 5.8: Promedio de latencias con respecto al tamaño de la respuesta. Izq - Requests en serie. Der - Requests concurrentes

En la figura 5.8 podemos observar comportamientos similares para las versiones 1 y 2 de HTTP en ambos escenarios. En el caso de HTTP/3 se pueden ver resultados opuestos. Por un lado, en la medición con concurrencia los resultados para HTTP/3 fueron notoriamente peores que el resto. Por el otro, al realizar los pedidos en serie, los resultados fueron considerablemente superiores. Estos resultados coinciden con los observados en la sección 5.3, donde el rendimiento de QUIC y HTTP/3 fue inferior cuando los requests fueron realizados concurrentemente.

Otro punto importante a notar es la relación entre la latencia y el tamaño de la respuesta, donde en ambos casos esta se comportó igual. No se observó un crecimiento de latencia a medida que se incrementaron los tamaños de la respuesta.

Una explicación para esto es un escenario de congestión en la red o al menos que el servidor de aiortc que soporte la carga enviada (como ya fue mencionado, en esta parte nos desviamos de la idea inicial de probar implementaciones productivas). Por otro lado, para las versiones anteriores, HTTP/2 tuvo un rendimiento levemente mejor que se hizo más notorio para respuestas más pesadas cuando se realizaban los requests concurrentemente.

Capítulo 6

Conclusiones

Durante esta tesis hemos tenido la oportunidad de hacer una revisión sobre el contexto, las motivaciones y los cambios introducidos en QUIC y HTTP/3. La necesidad de poder armar un protocolo que se adapte a las necesidades de la Internet moderna se vio dificultada por la osificación del protocolo TCP, que no permite la introducción de cambios disruptivos sin afectar todos los dispositivos que lo usen (en especial aquellos que no se actualizan más). El uso de UDP como transporte abrió la posibilidad a la creación de nuevos protocolos como QUIC, que deben implementar ciertas garantías dadas en TCP, pero al mismo tiempo pueden resolver las limitaciones de este. Dado que UDP no tiene comparte esas limitaciones. Adicionalmente, la implementación de QUIC en el user-space resulta en una mayor facilidad y flexibilidad a la hora de desarrollar y desplegar cambios. Esto resultó en el desarrollo de funcionalidades novedosas como el migrado de conexiones, multiplexado a nivel capa de transporte, 0-RTT Handshake y otras más. Estas mejoras y su buena recepción posicionaron resultaron en el uso de QUIC como transporte para HTTP/3. En la sección 3 estudiamos como este se encuentra diseñado y como varios de los mecanismos complejos de HTTP/2 ahora son simplemente delegados a QUIC, pues en este ya se encuentran resueltos.

En base a todo lo mencionado, podemos ver a QUIC como el eventual reemplazo de TCP. Si esto último resulta ser cierto, solo lo sabremos con el paso del tiempo. La adopción de un protocolo involucra muchos factores como la velocidad, su facilidad de implementación, costo, y muchos otros más. Adicionalmente, será importante seguir de cerca la evolución de las herramientas para administrar y controlar el tráfico QUIC, ya que los mismos mecanismos de seguridad que garantizan la privacidad de nuestros datos (y reducen el acoplamiento en los middleboxes) son aquellos que dificultan securizar una red ante este tipo de tráfico. Por el momento, sabemos que la adopción de QUIC dependerá casi exclusivamente de la adopción del protocolo HTTP/3. Mientras esto sucede, será muy común encontrarse con implementaciones mixtas. En el caso de HTTP, sabemos que el descubrimiento de versiones del protocolo por ahora se debe realizar usando las versiones previas, y por lo tanto depende de TCP. Esto resulta en la necesidad de un alto grado de adopción para poder usar HTTP/3 *libremente*. En este trabajo buscamos obtener datos sobre el funcionamiento de QUIC y HTTP/3 en ambientes productivos y si actualmente podemos observar una mejora con respecto a sus contrapartes más antiguas. Creemos que esto puede resultar de utilidad para entender qué tan rápido, o con cuánta facilidad, estos protocolos pueden dar valor a alguien que esté considerando utilizarlos. En la sección 5 experimentamos con distintos sitios que implementan HTTP/3. En general los resultados

obtenidos mostraron que el rendimiento de HTTP/2 y HTTP/3 puede ser bastante similar, dependiendo la implementación utilizada. No obstante, se observan ventajas por parte de HTTP/2 en la mayoría de los casos. Por otro lado, también se realizaron mediciones que intenten dar lugar a distintas funcionalidad de QUIC y HTTP/3. En la sección 5.3 fue posible observar mejoras de latencia para HTTP/3 cuando podemos aprovechar el multiplexado de QUIC. A su vez, durante el análisis realizado en la sección 5.3.1, también se encontraron mejoras de latencia con respecto a HTTP 1 y 2 cuando forzamos el recreado de conexiones, dando lugar al uso de las funcionalidades de session resumption y 0-RTT handshake. Estos resultados nos llevan a concluir que para estos casos la implementación de HTTP/3 no es tan directa si queremos ver resultados concretos en latencia. En especial considerando los casos en donde nos referimos a algunos de los servicios web más usados y optimizados del mundo. Adicionalmente, es importante elegir la implementación de QUIC y HTTP/3. A pesar de implementar los mismos protocolos, observamos como los resultados pueden variar considerablemente. Más aún, cada biblioteca tiene sus propias formas de configurarse, lo cual implica un esfuerzo distinto de implementación para cada lenguaje.

Otro punto importante a considerar es que el uso de CDNs y caches puede estar resultando en la diferencia entre las versiones del protocolo sean menores y aumenta la varianza de las muestras tomadas. Esta varianza se pudo observar fuertemente en las pruebas de multiplexado. No obstante, obtuvimos resultados mayormente positivos para HTTP/3 y QUIC. Estos resultados nos llevaron a alejarnos un poco de nuestra idea original y probar con un servidor de pruebas como lo es quic.aiortc.org, dado que este no cuenta con ninguna CDN. En este caso vimos latencias considerablemente mejores para QUIC con respecto a sus predecesores. No obstante, nos encontramos con un caso peculiar dado por el paralelismo de los requests HTTP.

Concluyendo, a pesar de las numerosas mejoras introducidas en QUIC y HTTP/3, su implementación no es una garantía de mejor rendimiento. En especial en arquitecturas muy optimizadas que cuentan con redes de contenido. En estos casos no necesariamente detectaremos con una mejora inmediata y se va a requerir de un esfuerzo para optimizar la implementación. No obstante, la presencia de funcionalidades como 0-RTT handshake y multiplexado a nivel transporte, abren las posibilidades a mejoras de rendimiento, siempre y cuando podamos generar las configuraciones y situaciones correctas. Esperamos que a medida que la adopción de QUIC y HTTP/3 continúe creciendo, también lo hagan las bibliotecas necesarias para su implementación, resultando en mejoras más notorias, o al menos resultados menos variados entre cada alternativa.

6.1. Trabajo Futuro

Esta tesis busca dar una primera idea de las mejoras inmediatas que pueden ofrecer QUIC y HTTP/3 en un ambiente productivo. No obstante, hay muchos aspectos interesantes para investigar sobre estos nuevos protocolos.

Actualmente, existen una gran cantidad de implementaciones de QUIC y HTTP/3, y ese número continua creciendo aún. Durante este trabajo solo tuvimos la oportunidad de estudiar tres bibliotecas. Una continuación de este trabajo puede ser continuar explorando las distintas implementaciones que existen. Adicionalmente, hay un concepto muy interesante que surge de la variedad de implementaciones de QUIC que es la sinergia entre bibliotecas. Es decir, estudiar diversas combinaciones de bibliotecas que generen resultados de mayor rendimiento, o en contraparte, encontrar combinaciones que puedan generar peores resultados.

En la sección 4.1 planteamos la utilización de la latencia de punta a punta como medida para comparar los protocolos. Aunque esta es útil para un primer análisis, sería interesante trabajar con otras formas de comparar los protocolos. Adicionalmente, otro punto para profundizar podría ser explorar el rendimiento de QUIC y HTTP/3 en distintas condiciones de red. Para esto, una vez más, tendríamos que alejarnos de las implementaciones reales y trabajar con simuladores que nos permitan generar y recrear esos escenarios.

Apéndice A

Configuración de Red

A continuación se adjunta la configuración de red de la interfaz utilizada durante la experimentación de esta tesis. Las direcciones MAC e IP se encuentran ofuscadas por privacidad.

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
     options=400<CHANNEL_IO>
     ether 3c:xx:xx:xx:xx:xx
     inet6 2800::xxx:xxxx:xxxx:2333%en0 prefixlen 64 secured scopeid 0xb
     inet6 2800:xxxx:xxxx:xxx:xxx:xxxx:xxxx:c6f2 prefixlen 64 autoconf secured
     inet6 2800:xxxx:xxxx:xxx:xxxx:xxxx:xxxx:493d prefixlen 64 autoconf temporary
     inet6 2800:xxxx:xxxx:xxx::ca24 prefixlen 64 dynamic
     inet 192.168.0.168 netmask 0xffffff00 broadcast 192.168.0.255
     nd6 options=201<PERFORMNUD,DAD>
     media: autoselect
     status: active
```

Apéndice B

Resultados con *outliers*

En el capítulo 5, se presentaron los resultados de la experimentación realizada en esta tesis. Como se mencionó en ese capítulo, con el propósito de facilitar la visualización e interpretación de los resultados en las secciones 5.1 y 5.2, se presentaron sin incluir los datos identificados como outliers. Sin embargo, es importante destacar que las distribuciones fueron construidas utilizando todos los datos recopilados.

B.1. Método de Detección

Para la identificación de datos considerados outliers, se emplea una medida estadística comúnmente conocida como el rango intercuartil (IQR en inglés). [48] Esta medida se calcula de la siguiente manera:

$$IQR = Q_3 - Q_1$$

Donde Q_1 y Q_3 representan los cuartiles 1 y 3, equivalentes a los percentiles 25 y 75, respectivamente. En base a esto, una medida es clasificada como outlier si esta es menor que $Q_1 - \frac{3}{2}IQR$ o mayor que $Q_3 + \frac{3}{2}IQR$

B.2. Resultados

A continuación, se presentan los resultados de las mediciones realizadas en las secciones 5.1 y 5.2, incluyendo los outliers detectados mediante el método descrito en la sección B.1.

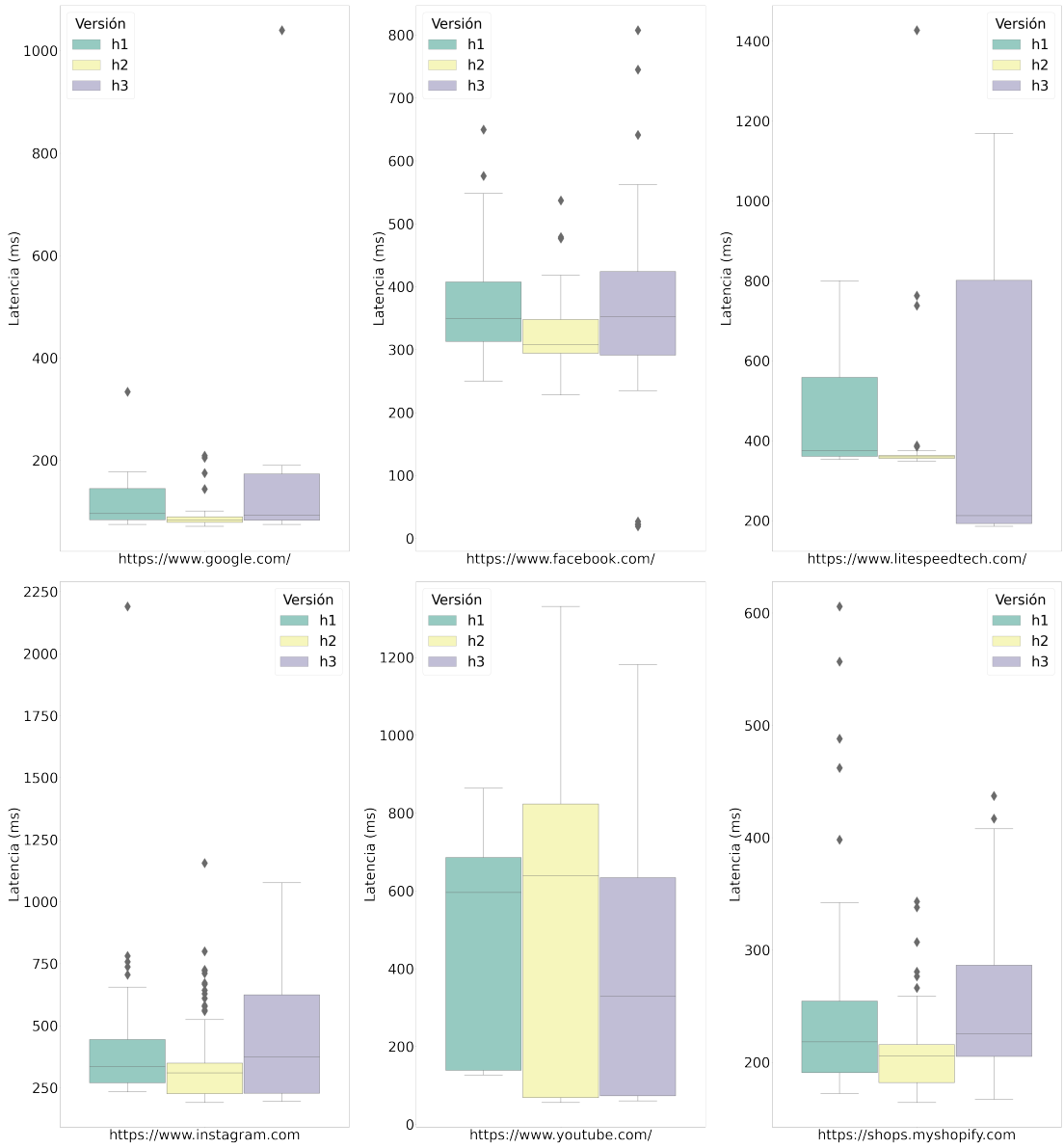


Figura B.1: Comparación de latencia por cada servidor y versión del protocolo HTTP de las tres implementaciones utilizadas. Corresponde a la figura 5.1.

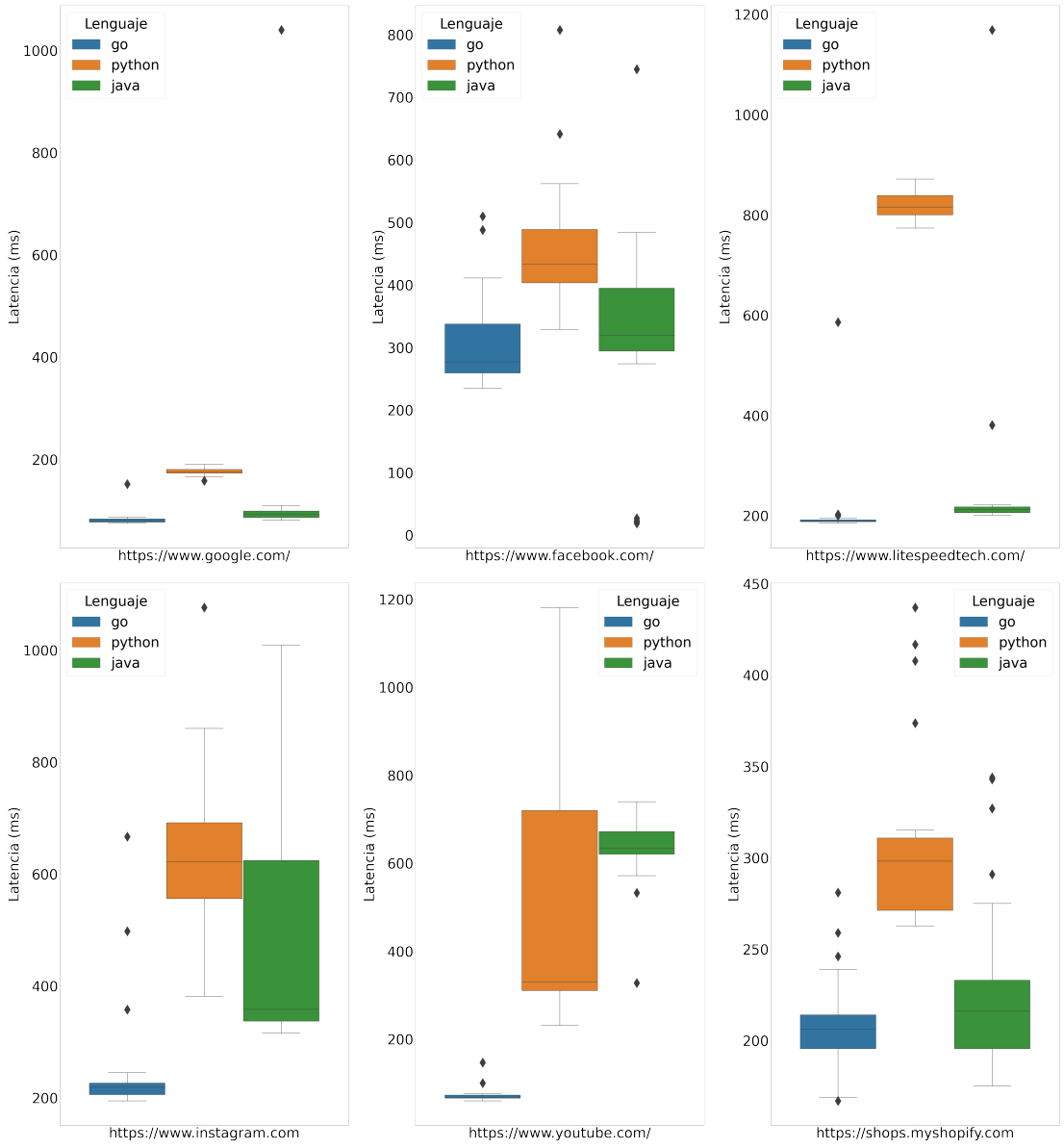


Figura B.2: Comparación de latencia por cada servidor e implementación de QUIC y HTTP/3. Corresponde a la figura 5.2.

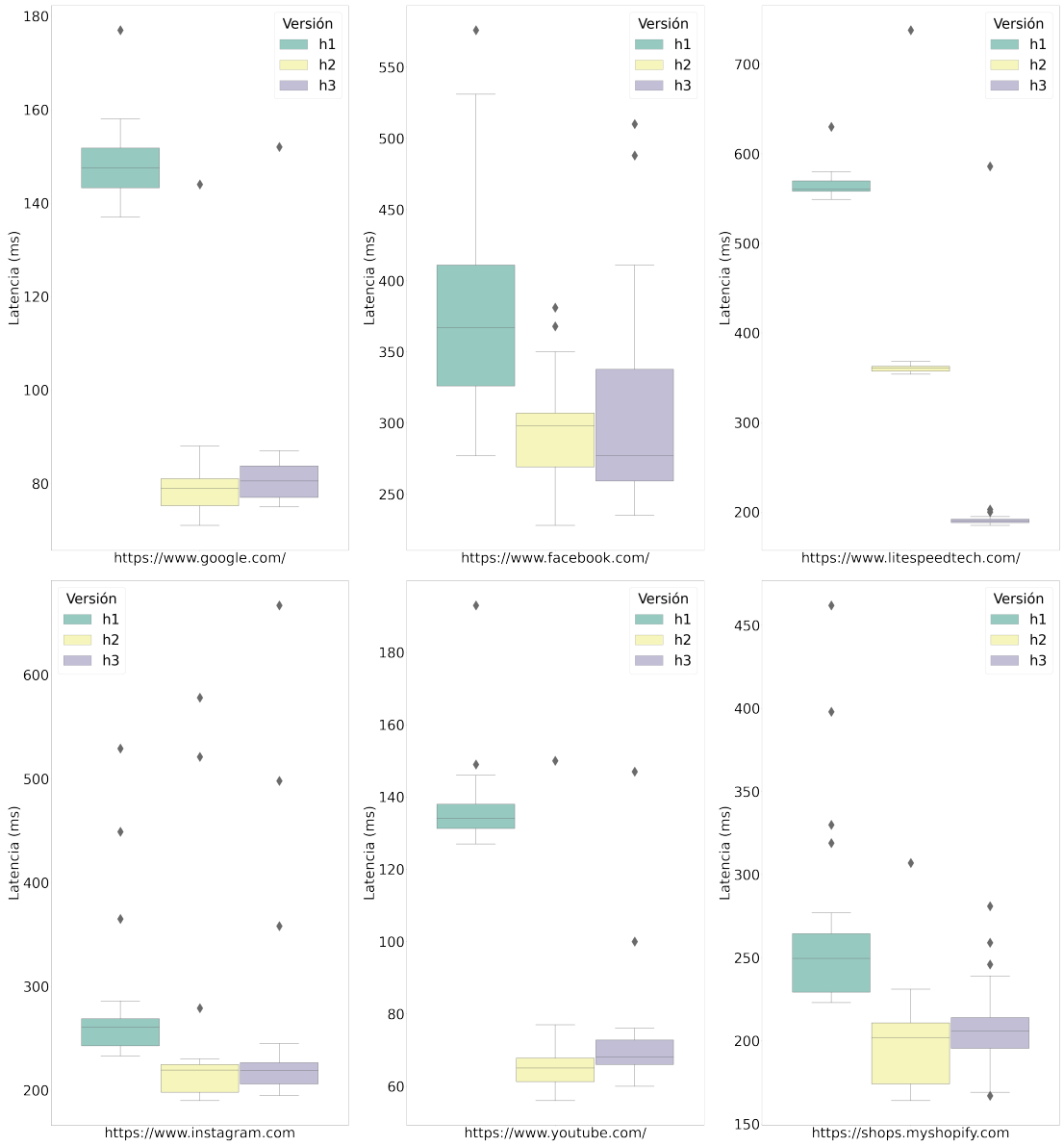


Figura B.3: Comparación de latencia por cada servidor y versión del protocolo HTTP únicamente para las mediciones usando Go Lang. Corresponde a la figura 5.3.

Apéndice C

Mediciones fallidas con quic.cloud

Quic.cloud es una empresa parte de Litespeedtech. Esta provee servicios de CDN para acelerar el rendimiento de servicios basados en WordPress. [1]Al igual que su empresa padre, quic.cloud soporta HTTP/3 y este forma parte de su propuesta de valor.

Durante el proceso de experimentación de esta tesis, se investigó sobre qué sitios tienen soporte para HTTP/3 y podían ser utilizados para realizar las mediciones tal como se explica en la sección 4.4. En este escenario consideró a quic.cloud como parte del conjunto inicial de sitios para medir el rendimiento de las versiones de HTTP.

A la hora de hacer mediciones sobre quic.cloud se obtuvieron resultados variados que dificultaron el análisis y la comparación con los otros sitios seleccionados. Durante las mediciones realizadas se observaron múltiples timeouts esporádicos. Más aún, para la implementación desarrollada en Java, no fue posible realizar requests hacia quic.cloud a causa de lo que aparenta ser un ciclo infinito de redirecciones, pues cada request realizado devolvió una reducción al mismo dominio. Este último fenómeno resulta especialmente raro pues solo sucedió para una de las implementaciones utilizadas. La suma de los timeouts y el problema de las redirecciones para Java resultó en que quic.cloud sea descartado del conjunto de sitios a medir durante la experimentación de esta tesis.

[1]WordPress es un sistema de gestión de contenidos open source enfocado al desarrollo de páginas web sin necesidad de programar. Segun W3Techs, el 43,1 % de los sitios de Internet en 2022 se utilizan WordPress

Bibliografía

[1] 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). Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842

[2] Jim Roskind, Chromium. 2013. QUIC: Design Document and Specification Rationale https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?usp=sharing

[3] 1981. RFC793. Transmission Control Protocol. https://doi.org/10.17487/RFC0793

[4] Jonathan Rosenberg. 2008. UDP and TCP as the New Waist of the Internet Hourglass. https://datatracker.ietf.org/doc/html/draft-rosenberg-internet-waist-hourglass-00

[5] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is it still possible to extend TCP? In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC '11). Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/2068816.2068834

[6] Scott W. Brim and Brian E. Carpenter. 2002. RFC3234. Middleboxes: Taxonomy and Issues. https://doi.org/10.17487/RFC3234

[7] G. Papastergiou et al. 2017. "De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives,ïn IEEE Communications Surveys & Tutorials, vol. 19, no. 1, pp. 619-639, Firstquarter 2017, https://doi.org/10.1109/COMST.2016.2626780.

[8] InfoQ. 2015. Google Will Propose QUIC As IETF Standard. https://www.infoq.com/news/2015/04/google-quic-ietf-standard/

[9] Jana Iyengar and Martin Thomson. 2021. RFC9000. QUIC: A UDP-Based Multiplexed and Secure Transport. https://doi.org/10.17487/RFC9000

[10] Marx, Robin and Herbots, Joris and Lamotte, Wim and Quax, Peter. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. https://doi.org/10.1145/3405796.3405828

[11] Martino Trevisan, Danilo Giordano, Ali Safari Khatouni. Measuring HTTP/3: Adoption and Performance. 19th Mediterranean Communication and Computer Networking Conference. 2021. https://doi.org/10.1109/MedComNet52149.2021.9501274

[12] Yu, Alexander and Benson, Theophilus A. Dissecting Performance of Production QUIC. 2021. https://doi.org/10.1145/3442381.3450103

[13] Roy T. Fielding and Julian Reschke. 2014. RFC7230. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://doi.org/10.17487/RFC7230

[14] Mike Belshe and Roberto Peon and Martin Thomson. 2015. RFC7540. Hypertext Transfer Protocol Version 2 (HTTP/2). https://doi.org/10.17487/RFC7540

[15] Jana Iyengar and Ian Swett. 2021. RFC9002. QUIC Loss Detection and Congestion Control. https://doi.org/10.17487/RFC9002

[16] Martin Thomson and Sean Turner. 2021. RFC9001. Using TLS to Secure QUIC. https://doi.org/10.17487/RFC9001

[17] P. Karn and C. Partridge. 1987. Improving round-trip time estimates in reliable transport protocols. SIGCOMM Comput. Commun. Rev. 17, 5 (Oct./Nov. 1987), 2–7. https://doi.org/10.1145/55483.55484

[18] Mike Bishop. 2022. RFC9114. HTTP/3. https://doi.org/10.17487/RFC9114

[19] Roberto Peon and Herve Ruellan. 2015. RFC7541. HPACK: Header Compression for HTTP/2. https://doi.org/10.17487/RFC7541

[20] Cloudfare. 2016. HPACK: the silent killer (feature) of HTTP/2. https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/

[21] Charles 'Buck' Krasic and Mike Bishop and Alan Frindell. 2022. RFC9204. QPACK: Field Compression for HTTP/3. https://doi.org/10.17487/RFC9204

[22] John Dellaverson, Tianxiang Li, Yanrong Wang, Jana Iyengar, Alexander Afanasyev, Lixia Zhang. 2022. A Quick Look at QUIC. University of California, Los Angeles.

[23] Matt Sargent, Jerry Chu, Dr. Vern Paxson, Mark Allman. 2011. RFC6298. Computing TCP's Retransmission Timer. https://doi.org/10.17487/RFC6298

[24] Jana Iyengar, Ian Swett, Robin Marx. 2020. Tutorial on the QUIC Protocol. ACM SIGCOMM 2020. https://www.youtube.com/watch?v=31J8PoLW9iM

[25] QUIC Implementations. https://github.com/quicwg/base-drafts/wiki/Implementations

[26] Roy T. Fielding, Mark Nottingham, Julian Reschke. 2022. RFC9110. HTTP Semantics. https://doi.org/10.17487/RFC9110

[27] Mark Nottingham, Patrick McManus, Julian Reschke. 2016. RFC7838. HTTP Alternative Services. https://doi.org/10.17487/RFC7838

[28] D. A. Huffman. 1952. .ᴬ Method for the Construction of Minimum-Redundancy Codes,ïn Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, Sept. 1952, https://doi.org/10.1109/JRPROC.1952.273898

[29] Ethan Blanton and Dr. Vern Paxson and Mark Allman. 2009. RFC5681. TCP Congestion Control. https://doi.org/10.17487/RFC5681

[30] Alia Atlas and JR. Rivers and Naiming Shen and Ron Bonica and Carlos Pignataro. 2010. RFC5837. Extending ICMP for Interface and Next-Hop Identification. https://doi.org/10.17487/RFC5837

[31] Mathis, M. and J. Mahdavi. 1996. Forward acknowledgement: Refining TCP Congestion Control. ACM SIGCOMM Computer Communication Review, urlhttps://doi.org/10.1145/248157.248181.

[32] Ethan Blanton and Mark Allman and Lili Wang and Ilpo Järvinen and Markku Kojo and Yoshifumi Nishida. 2012. RFC6675. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. https://doi.org/10.17487/RFC6675

[33] Yuchung Cheng and Neal Cardwell and Nandita Dukkipati and Priyaranjan Jha. 2021. RFC6675. The RACK-TLP Loss Detection Algorithm for TCP. https://doi.org/10.17487/RC8985

[34] Gbur, Konrad and Tschorsch, Florian. 2021. A QUIC(K) Way Through Your Firewall?.

[35] Geoffrey A. Moore. 2014. Crossing the Chasm, 3rd Edition: Marketing and Selling Disruptive Products to Mainstream Customers. https://books.google.com.ar/books?id=wUEHAQAAQBAJ

[36] Eric Rescorla. 2018. RFC8446. The Transport Layer Security (TLS) Protocol Version 1.3. https://doi.org/10.17487/RFC8446

[37] Tommy Pauly and Eric Kinnear and David Schinazi. 2022. RFC9221. An Unreliable Datagram Extension to QUIC. https://doi.org/10.17487/RFC9221

[38] Martin Duke. 2022. QUIC Version 2. Draft 10. https://datatracker.ietf.org/doc/draft-ietf-quic-v2/10/

[39] Yanmei Liu and Yunfei Ma and Quentin De Coninck and Olivier Bonaventure and Christian Huitema and Mirja Kühlewind. 2022. Multipath Extension for QUIC. Draft 03. https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/03/

[40] Alan Ford and Costin Raiciu and Mark J. Handley and Olivier Bonaventure and Christoph Paasch. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. https://doi.org/10.17487/RFC8684

[41] Dr. Bernard D. Aboba and Gonzalo Salgueiro and Colin Perkins. 2023. RFC9443. Multiplexing Scheme Updates for QUIC. https://doi.org/10.17487/RFC9443

[42] David Benjamin. 2020. Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility. https://doi.org/10.17487/RFC8701

[43] Martin Thomson. 2022. Greasing the QUIC Bit. https://doi.org/10.17487/RFC9287

[44] Martin Duke and Nick Banks and Christian Huitema. 2022. QUIC-LB: Generating Routable QUIC Connection IDs. Draft 15. https://datatracker.ietf.org/doc/draft-ietf-quic-load-balancers/15/

[45] Stephan Friedl and Andrei Popov and Adam Langley and Stephan Emile. 2014. RFC7301. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. https://doi.org/10.17487/RFC7301

[46] Benjamin M. Schwartz and Mike Bishop and Erik Nygren. 2023. Service binding and parameter specification via the DNS (DNS SVCB and HTTPS RRs. Draft 12. https://datatracker.ietf.org/doc/draft-ietf-dnsop-svcb-https/12/

[47] McGill, R., Tukey, J.W. & Larsen, W.A. 1978. Variations of box plots. The American Statistician, 32(1), pp.12–16.

[48] Dekking, Frederik Michel; Kraaikamp, Cornelis; Lopuhaä, Hen Paul; Meester, Ludolf Erwin. 2005. A Modern Introduction to Probability and Statistics. Springer Texts in Statistics. https://doi.org/10.1007%2F1-84628-168-7.