ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Mecanismos de control de congestión y errores en el protocolo QUIC

(Congestion and error control mechanism in the QUIC protocol)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Fátima Fernández Pérez

Octubre - 2019

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MASTER

Realizado por: Fátima Fernández Pérez.
Director del TFM: Ramón Agüero Calvo.
Título: "Mecanismos de control de congestión y errores en el protocolo QUIC"
Title: "Congestion and error control mechanism in the QUIC protocol"

Presentado a examen el día: 21 de octubre del 2019.

para acceder al Título de

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Pablo Pedro Sánchez Espeso
Secretario (Apellidos, Nombre): Luis Francisco Díez Fernández
Vocal (Apellidos, Nombre): Roberto Sanz Gil

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFM
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Máster Nº
(a asignar por Secretaría)

Agradecimientos

En primer lugar, me gustaría agradecer a Ramón Agüero por el apoyo y, sobretodo, por la confianza que siempre ha depositado en mi. También, darle las gracias a Pablo Garrido y a Mihail Zverev por la disposición y ayuda. Además, agradecer al Grupo de Ingeniería Telemática, en especial al Laboratorio de Redes y Servicios, por hacer más amenos mis días.

Por otro lado, quiero hacer un reconocimiento a mis compañeros de clase, y en concreto, al grupo del café, que sin ellos las mañanas se hubieran hecho mucho más largas. También a mis amigos, por las horas de desconexión necesarias que hacen que una vea las cosas de mejor manera.

Por último, hacer una mención especial a mi familia, que sin ella no hubiera sido posible llegar a donde estoy, en especial a mis padres y a mi hermana, que a cada meta llegamos juntas. Y a Fernando, por el apoyo incondicional y los ánimos que siempre me ha dado.

Resumen

Actualmente, con la evolución e integración de las redes inalámbricas, surge la necesidad de buscar protocolos de transporte que aporten menos latencia y un mejor aprovechamiento del ancho de banda disponible. Hoy en día, el principal protocolo de transporte en Internet es TCP, pero la continua evolució de las redes hace que este se quede obsoleto. QUIC aparece en la comunidad investigadora para suplir sus carencias, ya que tiene como principal objetivo reducir los tiempos de ida y vuelta.

El funcionamiento de los protocolos de transporte sobre redes inalámbricas no es óptimo, siendo el tiempo de recuperación de paquetes uno de los factores más perjudiciales. En este trabajo se recoge un estudio sobre la implementación de técnicas de corrección de errores para la recuperación de paquetes en el nuevo protocolo de transporte QUIC: rQUIC y QUIC-FEC. El objetivo principal de esta integración es la disminución de latencia, suprimiendo retransmisiones innecesarias cuando aparece un evento de pérdida de información.

Para llevar a cabo el análisis de este protocolo, se hace uso de un escenario de simulación, así como los cambios necesarios para modificar las alternativas en cuanto a la implementación de técnicas FEC. Se maneja el entorno de los contenedores LXC de Linux, el simulador de eventos discreto *ns-3*, y se trabaja con el lenguaje de programación GO.

Palabras clave: QUIC, rQUIC, QUIC-FEC, latencia, ancho de banda, control de congestión, *Forward Error Correction*, tiempo de ida y vuelta (RTT).

Abstract

Nowadays, with the evolution and integration of wireless networks, it becomes fundamental to look for transport protocols that provide short connection latencies and better use of available bandwidth. Currently, the most widespread transport protocol on the Internet is TCP, but the continuous network evolution makes it obsolete. QUIC appears in the research community to address its shortcomings, as its main goal is to reduce the round-trip time.

The behaviour of the transport protocols over wireless networks is suboptimal, with the packet recovery time as one of the most damaging factors. This work includes a study of the implementation of error correction techniques for packet recovery in QUIC: rQUIC and QUIC-FEC. The main objective of this integration is to decrease latency, avoiding unnecesary retransmissions when an information loss event appears.

In order to carry out the analysis of QUIC, a simulation scenario is used. In addition, we also tackle the necessary changes to modify the different solutions that integrate FEC techniques. The environment is based on the Linux LXC containers, the discrete-event network simulator *ns-3*, and the programming language is GO.

Key words: QUIC, rQUIC, QUIC-FEC, latency, bandwidth, congestion control, *Forward Error Correction*, round-trip time (RTT).

Índice

Índice de figuras

Índice de tablas

Acrónimos

QUIC Quick UDP Internet Connections

TCP Transmission Control Protocol

UDP User Datagram Protocol

BBR Bottleneck Bandwidth and Round-trip time

TLS Transport Layer Security

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

DTLS Datagram Transport Layer Security

AEAD Authenticated Encryption and Associated Data

FEC Forward Error Correction

DCT Download Completion Time

RTT Round-Trip Time

BW Bandwidth

1 Introducción

El protocolo de transporte por excelencia en Internet en los últimos años ha sido TCP, pero la evolución de las redes hace que este se quede obsoleto. Esto se debe a la complejidad que supone hacer cambios en su *kernel* y a la variedad de *middle-boxes*[1] que se encuentran en la red. Además, el compromiso que surge entre el aumento del ancho de banda y la reducción de la latencia en las comunicaciones hace que se busquen otras alternativas.

Quick UDP Internet Connections (QUIC) [1] [2] es un protocolo de capa de transporte cuyo funcionamiento se sustenta directamente sobre UDP. El diseño de este protocolo se focaliza en la reducción de la latencia, disminuyendo el número de retransmisiones, y, en contraste con TLS, dota a la comunicación de una mayor seguridad, encriptando la información útil y parte de la de control que se intercambia de extremo a extremo de la red. El uso de QUIC viene motivado por la fácil implementación y actualización en el sistema operativo de los nodos de Internet, siendo una ventaja fundamental sobre el resto de alternativas. QUIC tiene la capacidad de multiplexar una gran cantidad de flujos entre un cliente y un servidor, reduciendo así el número de conexiones que se generarían con TCP. Esto hace que el tráfico transmitido se unifique y, por lo tanto, la cantidad de respuestas sobre información del canal disminuyan (menos sobrecarga en la red), o que los protocolos de nivel superior (como SPDY[2]) compriman la transmisión de datos redundantes. En definitiva, QUIC reune algunas de las funcionalidades de HTTP/s, TLS

[1] Dispositivo que se encuentra en la red y sirve para filtrar, manipular, transformar e inspeccionar el tráfico que se envía.

[2] Protocolo de nivel de sesión desarrollado por Google. Aporta mayor velocidad en la descarga de páginas web mediante la realización de más peticiones en una única sesión TCP.

y TCP. Actualmente, aproximadamente el 7 % del tráfico de Internet se sustenta sobre QUIC, del cual más del 30 % pertenece a Google[2].

Un punto de vista importante es el que se refiere a las pérdidas que se ocasionan en el canal de comunicaciones. Los protocolos de congestión que utiliza TCP asumen estas pérdidas como congestión de la red, haciendo que para las redes inalámbricas (utilizadas en una gran variedad de campos como es *Internet of Things*, IoT) sean subóptimos, principalmente las técnicas de detección y recuperación de pérdidas. Para solventar esta insuficiencia se implementa Forward Error Correction (FEC) que es una técnica de corrección de errores que se basa en la transmisión de datos redundantes para recuperar las tramas perdidas en recepción, idónea para protocolos de transporte donde no se garantiza la entrega de paquetes, como es el caso de UDP. La utilización de este mecanismo hace que no se realicen retransmisiones y, por lo tanto, no haya tiempos de espera adicionales. Debido a la ventaja que ocasiona utilizar FEC en cuanto a rendimiento, han aparecido distintas implementaciones en QUIC que hace que sea interesante hacer una equiparación de todas ellas[3][4].

Actualmente, el protocolo de congestión que utiliza QUIC por defecto es TCP CUBIC. La ventaja que presenta este mecanismo es que la ventana de congestión varía de una forma más escalable y más estable que los mecanismos previos de TCP. Aunque el rendimiento de TCP CUBIC es mejor que el de TCP New Reno, se propone incluir este último para comprobar el comportamiento de QUIC al utilizarlo. Debido a la buena respuesta que puede ofrecer QUIC en redes inalámbricas, otro mecanismo de congestión interesante es Bottleneck Bandwidth and Round-trip time (BBR). TCP Reno y TCP CUBIC erróneamente asumen la pérdida de paquetes como congestión. La congestión se produce cuando la cantidad de información que hay en la red supera el Bandwidth Delay Product (BDP). BBR propone no tratar a la pérdida de paquetes como sinónimo de congestión, de tal manera que el ancho de banda se mantendría estable y la latencia se reduciría.

1.1. Objetivos

Tras plantear una introducción a las bases del proyecto, se plantean los objetivos que se han fijado en este Trabajo Fin de Máster. Para realizar los diferentes análisis, se ha utilizado la herramienta de simulación Network Simulator-3 (NS3). NS3 aporta las herramientas necesarias para simular distintos escenarios y así modificar las características de la red.

El principal objetivo es el estudio del protocolo de transporte QUIC con la inclusión de técnicas FEC en su funcionamiento, y la comparativa de las diferentes implementaciones que han surgido con las pertinentes modificaciones en el código. Por otro lado, se pretende estudiar la integración de otros protocolos de congestión (TCP New Reno y BBR).

Todo este estudio viene regido por la continua evolución de las redes inalámbricas y sus aplicaciones.

1.2. Estructura del documento

A continuación, se explica brevemente el contenido de la memoria y de qué forma se ha dividido el proyecto. Se resumirán los puntos principales de cada capítulo.

- **Capítulo 1. Introducción**

  En el actual capítulo se introducen las ventajas que tiene el protocolo de transporte QUIC sobre el resto de alternativas. Además, se citan nuevos mecanismos de corrección de errores y de congestión que se pueden añadir a QUIC nativo. Por otro lado, se fijan los objetivos del trabajo, así como la disposición de la memoria.

- **Capítulo 2. Estado del Arte**

  En el Capítulo 2 se hace una revisión teórica de todos los aspectos que se han tratado en el trabajo. Se explica en profundidad el protocolo QUIC, describiendo los distintos mecanismos de su funcionamiento y las diferentes alternativas en la inclusión de las técnicas FEC para la corrección de errores que se han estudiado. Adicionalmente, se repasan los protocolos de congestión en los que se ha centrado el trabajo como son TCP Reno, TCP CUBIC y BBR.

- **Capítulo 3. Entorno y desarrollo de simulación**

  En el tercer capítulo se expone el entorno de simulación, basado en una red simulada en NS-3, en contenedores linux (LXC) y el lenguaje de programación GO. Además, se detalla el proceso para hacer las medidas pertinentes con el protocolo QUIC, así como las modificaciones que se le han hecho para generar otros experimentos, como es la generación de una tasa fija de FEC o la inclusión de un nuevo mecanismo de congestión.

- **Capítulo 4. Análisis de resultados**

  En este capítulo, se explican inicialmente los resultados de la comparativa de las implementaciones de las técnicas FEC en QUIC nativo.

- **Capítulo 5. Conclusiones y Lineas futuras**

  Por último, en el quinto capítulo se detallan las conclusiones que se han obtenido analizando los resultados, así como las líneas futuras que surgen tras la realización de este trabajo.

2

Estado del Arte

El desarrollo de QUIC [2][5][6] viene dado por la necesidad de hacer un protocolo que sea flexible y fácil de implementar a través de Internet, es decir, que sea transparente a la red y que su inclusión y actualización no altere la programación de *middle-boxes* ni los equipos de los clientes. El potencial resultado que se puede obtener con este protocolo es la reducción de la latencia: se minimiza el tiempo de ida y vuelta (RTT) en el inicio de la comunicación, y mediante técnicas FEC se consigue reducir el retardo originado por la pérdida de paquetes. Por otro lado, se consigue la garantía de privacidad semejante a TLS, y una reducción en el ancho de banda, unificando el tráfico de señalización del canal a través de la multiplexación de *streams*. Estas ventajas de QUIC, junto a otras, hace que sea un protocolo idóneo para las redes inalámbricas.

En este capítulo se expone la base teórica de este protocolo, así como las alternativas que aparecen con la implementación de técnicas FEC (QUIC-FEC y rQUIC) y los posibles protocolos de congestión combinables con QUIC (TCP Reno, TCP CUBIC y BBR).

2.1. Quick UDP Internet Connections (QUIC)

Quick UDP Internet Connections (QUIC) es un protocolo a nivel de capa de transporte desarrollado por Google en 2016 y que actualmente se encuentra bajo estandarización por el IETF. Su principal implementación ha sido en servidores de Google, sustentando un 7 % del tráfico actual de Internet. QUIC ofrece multiplexación de *streams*, encriptado de

la información y baja latencia en la comunicación, para la mejora del rendimiento a nivel de transporte para el tráfico HTTPs.

El objetivo de QUIC es aunar algunas de las funcionalidades que se tenían hasta ahora con la pila de protocolos HTTP/2, TLS y TCP, e implementarlas sobre el protocolo de transporte UDP, Figura 2.1. La implementacion de los mecanismos de congestión se hace diréctamente sobre el protocolo QUIC, y se utiliza UDP para permitir que los paquetes atraviesen los puntos intermedios de la red. Además, sustituye la seguridad que ofrece TLS integrando encripatado y autenticación de extremo a extremo para que la información no se vea alterada. También se encripta el establecimiento de la conexión (*handshake*), lo cual hace que la latencia en el inicio de la comunicación se reduzca notablemente, debido a que se reutilizan las credenciales del servidor para conexiones redundantes y se elimina el *overhead* que resulta en este punto de la comunicación. La característica que adquiere QUIC del protocolo HTTP/2 es la multiplexación de flujos en una misma conexión. Esto hace que la pérdida de un solo paquete único bloquee flujos de datos que contengan ese paquete.

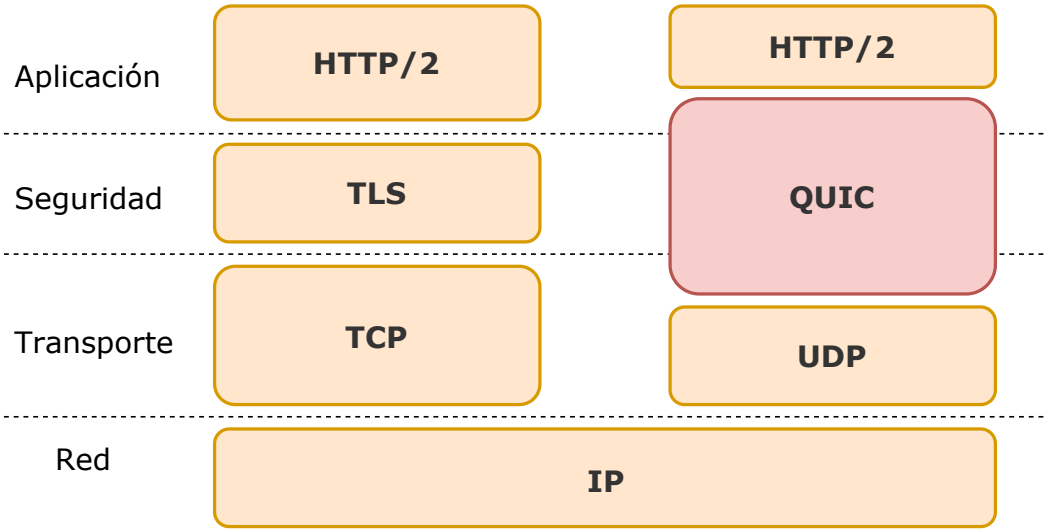


Figura 2.1: Arquitectura de la red con QUIC.

La motivación de fomentar este nuevo protocolo también viene dada por la utilización de SPDY [5]. SPDY es un protocolo de capa de sesión creado por Google, que opera sobre TCP y se encarga de multiplexar flujos. La forma en la que opera para reducir la latencia es mandar todas las peticiones sin necesidad de esperar a que peticiones anteriores se hayan resuelto, y así es capaz de reducir el ancho de banda comprimiendo tráfico redundante. Aunque SPDY aporta muchas ventajas, existe un compromiso perjudicial entre el uso de recursos y la reducción de la latencia.

1. Con SPDY, el retraso o pérdida de un solo paquete produce un bloqueo en todos los

flujos (*head-of-line blocking*) debido a que TCP solo proporciona una interfaz. Esto hace que el retardo de un solo paquete afecte al resto de flujos. Con QUIC se consigue que la pérdida de un paquete solo afecte al flujo al que pertenece.

2. Una conexión hecha con SPDY sustituye múltiples conexiones TCP, pero el protocolo de congestión no se utiliza de forma eficaz. La pérdida de un solo paquete SPDY hace que se reduzca el ancho de banda hasta un 50 % (ventana de congestión de TCP CUBIC), porque perjudica a todas las conexiones. Sin embargo, si se establecieran las conexiones TCP de manera independiente, la pérdida de un paquete solo afectaría a la conexión donde se encuentra dicho paquete, reduciendo el ancho de banda mucho menos que con el uso de SPDY.

3. El uso de TLS provoca un retraso de al menos 1 RTT en el establecimiento de la comunicación. Además, requiere que los paquetes se desencripten ordenadamente. Esto se soluciona en la versión 1.1 y en DTLS, donde se genera un vector de inicialización añadiendo más bytes al paquete TLS. Con QUIC, el inicio de la comunicación se haría de forma segura (autenticación y encriptación) en menos tiempo e invirtiendo menos recursos en la desencriptación de los paquetes.

El diseño e implementación del protocolo se basa en la reducción de la latencia durante el establecimiento de la comunicación, especialmente durante la reanudación de la conexión. También se centra en asegurar una eficiencia y baja latencia durante el intercambio de información entre dos extremos, o cuando el canal está vacío pero la conexión es estable.

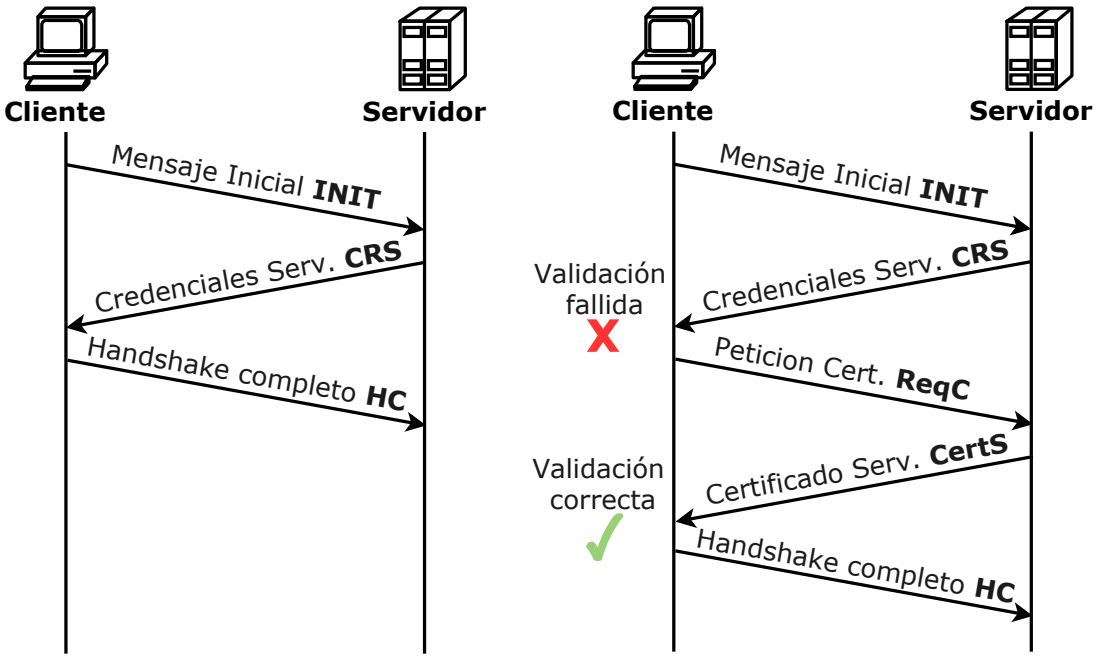
Establecimiento de la conexión

En una primera instancia, el establecimiento de la comunicación se basa en un intercambio criptográfico para establecer un canal seguro. El cliente almacena las credenciales del servidor para que en próximas conexiones se mantenga una comunicación encriptada y no se tengan que intercambiar mensajes innecesarios que harían que se aumentara el número de RTTs. El cliente asume que las credenciales del servidor (pareja de clave pública y privada) son las mismas, y así la fase inicial se reduzca a 0-RTT.

En esta fase, se forman varios escenarios en referencia a las conexiones iniciales de QUIC. Aparecen distintos mecanismos de *handshake*, en base a la cantidad de RTTs que se utilicen.

- ***Handshake* inicial**: En este escenario, el cliente no tiene ninguna información almacenada sobre el servidor. El cliente manda un mensaje aleatorio inicial (**INIT**) para empezar la sesión de negociación. En el otro lado, el servidor responde con la siguiente información (**CRS**): un certificado con su clave pública, la información sobre

la configuración del servidor, la cadena de autenticación del certificado, un *hash*[3] o firma de la cadena de autenticación y un bloque de autenticación (*token*) con el control de la dirección IP pública y puerto del cliente, además de un *timestamp*, que será utilizado por el cliente en próximas conexiones. Una vez realizado este intercambio de mensajes, el cliente verifica los parámetros del servidor a través del *hash* y envía su clave pública (**HC**), Figura 2.2a. Si no se valida correctamente dicha información, el cliente aumenta a 2-RTT y requiere un nuevo mensaje con la cadena de autenticación del certificado del servidor (**ReqC**), Figura 2.2b.



(a) *Handshake* inicial completo, 1-RTT. (b) *Handshake* inicial fallido, 2-RTT.

Figura 2.2: Escenario inicial.

Una vez completada esta fase inicial de 'descubrimiento', en las próximas conexiones se entrará en un escenario de 0-RTT debido a que el reconocimiento entre el cliente y servidor será instantáneo, reduciendo así la latencia en el establecimiento de la conexión.

- ***Handshake* repetido**: En una conexión repetida el cliente se asegura que el servidor siga manteniendo las credenciales validadas anteriormente. A continuación, se manda un mensaje de '*handshake* criptográfico' al servidor, (**ReqCr**), con la clave de sesión para intercambiar información encriptada (mecanismo similar a TLS: *TLS Session*

[3]Es una secuencia alfanumérica que se obtiene al aplicar a un mensaje x técnicas de codificación para obtener un 'resumen', y, único del mismo.

Master Secret[4]), la prueba del control del cliente sobre su IP y puerto, e información negociable como es el mecanismo de congestión que se va utilizar. A partir de este punto, el cliente puede mandar mensajes encriptados y autenticados al servidor antes de recibir su aceptación consiguiendo una latencia de 0-RTT. Si la clave pública del servidor previamente validada, ya no está en uso, o las condiciones expuestas por el cliente no las puede soportar el servidor, este último puede rechazar el contenido del paquete y asociarlo al caso explicado en la Figura 2.3. También se descartarán todos los paquetes de datos cifrados con las claves iniciales enviados desde el cliente después del '*handshake* criptográfico'.

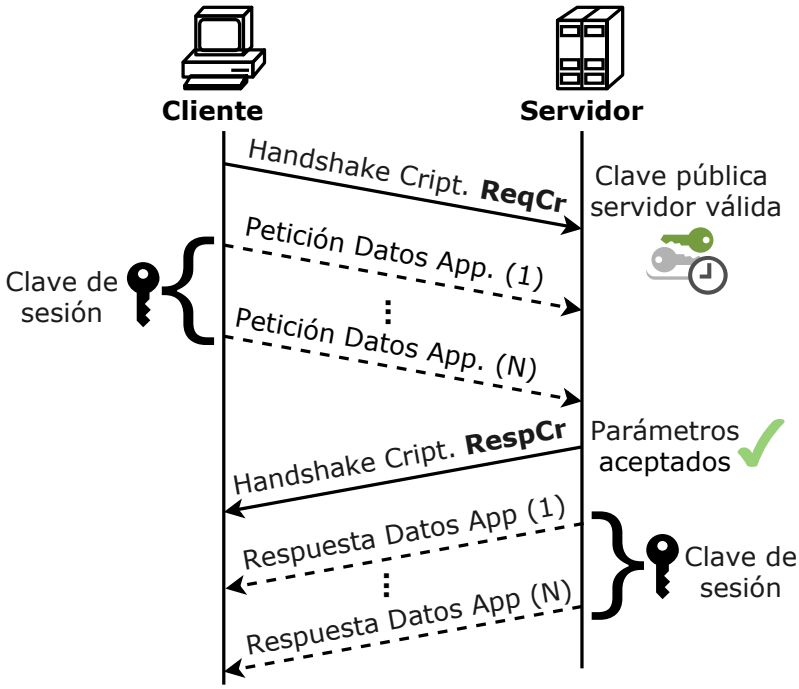


Figura 2.3: *Handshake* repetido, 0-RTT.

En el caso de que la clave pública del servidor siga siendo correcta y los parámetros propuestos sean aceptables el servidor, (**RespCr**), la prueba de control del puerto y dirección IP del cliente[5] es suficiente para poder descifrar y procesar los siguientes mensajes con datos de aplicación, Figura 2.3. Después de que el cliente reciba la

[4]Proceso que sigue el protocolo TLS para crear la clave de sesión con la que se cifrarán los mensajes de aplicación. El cliente genera una clave aleatoria mediante un mecanismo de clave simétrica, y se la manda al servidor cifrándola con la clave pública de este. Cada extremo tiene su pareja de claves pública-privada generada a partir de mecanismos de criptografía asimétrica.

[5]Durante una conexión, un servidor QUIC puede crear y transmitir una declaración de que un cliente está utilizando una dirección y un puerto específicos en un momento determinado. Esta declaración incluye una MAC que será suficiente para autenticarse y entrar en el escenario de 0-RTT. Esta declaración se puede actualizar en una misma conexión con distintas marcas temporales.

aprobación del servidor, ambos extremos podrán crear la clave final para encriptar el resto de información a partir de un valor generado por el servidor.

El cliente almacena en su caché la configuración del servidor con el que se está comunicando, incluido el *token* de dirección IP del cliente. Como se ha descrito anteriormente, este *token* será suficiente para iniciar una conexión repetida con datos cifrados. Si dicho *token* expira durante la comunicación o el servidor cambia sus certificados, la comunicación falla y se volvería al caso inicial de la Figura 2.2b.

Estructura de un paquete QUIC

Toda la información se fragmenta en bloques y se emplaza en paquetes UDP. Todos los paquetes QUIC se forman a partir de una sección de cabecera y otra de datos, y cada parte de datos se forma a partir de una secuencia de *frames*. Los bloques de datos son bloques encriptados a partir del mecanismo AEAD (*Authenticated Encryption and Associated Data*).

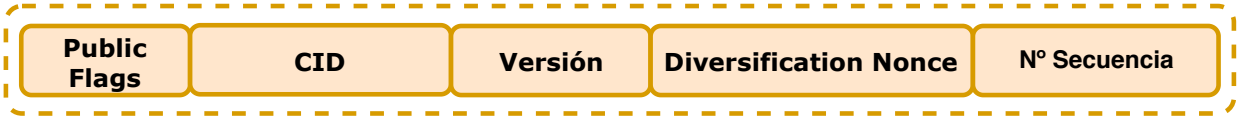


Figura 2.4: Campos de la cabecera QUIC.

La estructura básica de la **cabecera**, Figura 2.4, de cada paquete QUIC se conforma a partir de los siguientes campos: *Public Flags*, *Connection Identifier* (CID), la versión de QUIC, *Diversification Nonce* y el número de secuencia del paquete. Esta parte solo se autentica.

Public Flags: su tamaño máximo es de 1 byte, y detalla el diseño del resto de la cabecera de forma más compacta.

Connection Identifier (CID): es un identificador único de cada conexión y su longitud es de 64 bits. En una conexión puede variar el puerto de origen debido al servicio que ofrecen los NAT[6], siendo insuficientes una IP y puerto origen para definir una conexión. El CID se utiliza para definir una conexión durante todo el tiempo en el que se esté utilizando, de forma que la probabilidad de colisión con otras disminuya. Este parámetro es un valor aleatorio de 64 bits y suele ser definido en el primer paquete UDP que manda el cliente al servidor, estando presente en el resto de manera implícita o explícita. Posteriormente, el servidor puede utilizar el CID para identificar una conexión específica dentro de las que

[6] *Network Address Translation*: mecanismo utilizado en la red para posibilitar la conexión entre direcciones IP privadas y el resto de Internet.

puede tener activas (aproximadamente 2^{32} conexiones simultáneas) o para el mecanismo AEAD.

Versión de QUIC: este campo solo se utiliza en el primer paquete dirigido al servidor para que este pueda comprender la versión de QUIC que se está empleando. Su tamaño es de 32 bits. Para evitar redundancia, en el resto de paquetes el valor en *public flags* de este campo será 0.

Diversification Nonce: este valor se representa con 256 bits y se utiliza para el algoritmo de cifrado y autenticación.

Número de secuencia del paquete: este parámetro es importante en el procesado de paquetes para determinar aquellos que estén duplicados o falten, y para la encriptación. Este número forma parte de la base del vector inicial que se utiliza para desencriptar cada paquete. El número de bits que se utilizan para determinar el número de secuencia se fija en las *public flags* y puede ser de 8, 16, 32 o 48 bits.

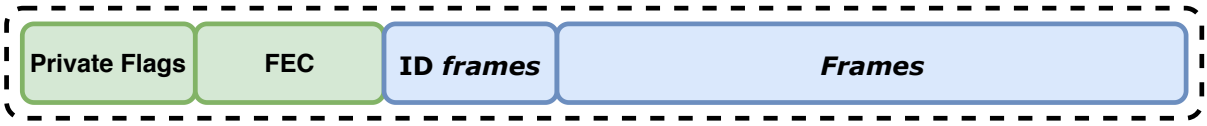


Figura 2.5: Payload de QUIC.

La parte de **datos**, Figura 2.5, en un paquete QUIC siempre se cifra y autentica mediante el algoritmo AEAD. En este apartado, se encuentra un bloque compuesto por bits de redundancia para la autenticación, concatenados con un *string* del tamaño de la carga útil de los datos. Este bloque consiste de las siguientes partes: *Private Flags*, número del grupo FEC e indentificador de *frames*.

Private Flags: su extensión máxima es de 1 byte y se denominan como 'privadas' porque van encriptadas. El primer bit es el bit de entropía y el segundo representa el tamaño del número del grupo FEC y por lo tanto, el offset donde empiezan los *frames* de datos.

- Bit de entropía: este bit se selecciona aleatoriamente por el emisor y principalmente se utiliza para combatir el ataque *Optimistic Ack Attacks*[7]. El receptor en los paquetes de reconocimiento debe incluir el *hash* de los bits de entropía recibidos hasta ese momento, para que el servidor pueda corroborarlo.

- Número del grupo FEC: se utiliza para identificar el último paquete en un grupo FEC y por lo tanto donde comienzan los bloques de datos útiles. Es un paquete que contiene

[7]Es un ataque basado en la transmisión de varios ACKs por parte del cliente para inhabilitar al servidor. Hace que aumente la velocidad de transmisión del servidor y por lo tanto, se quede sin ancho de banda para atender otras solicitudes.

la XOR de toda la carga útil en el grupo FEC que se esté tratando.

Número del grupo FEC: este campo de 1 byte se habilita en las *Private Flags*. Indica el número del primer paquete en un grupo FEC. Para obtener el número del grupo FEC se restará el offset que representa este valor al número de secuencia del paquete.

Indentificador de *frames*: este parámetro identifica el tipo de datos que contiene un paquete mediante un byte. Es importante debido a que especifica el formato y el contenido de los datos en cada bloque de un paquete. Hay gran variedad de tipos de *frames* implícitos en cada paquete de datos: bloques de reconocimiento (*ACK frame*), bloques de control de congestión (*Congestion control frame*), bloques para indicar el cierre de conexión (*Connection close frame*), etc.

Elementos criptográficos y multiplexación de *streams*

Todos los paquetes QUIC son autenticados y por lo general, la parte de carga útil está totalmente cifrada.

Como se ha mencionado, para la autenticación y encriptado de los datos en el paquete de QUIC se utiliza ***Authenticated Encryption and Associated Data***. Con este mecanismo se consigue evitar la dependencia existente en QUIC de decodificar siempre y cuando los paquetes lleguen en orden. Esto se consigue mediante un vector inicial derivado del número de secuencia de cada paquete. Este valor de la cabecera no se encripta, para admitir la decodificación no serializada. Por otro lado, el parámetro *Diversification Nonce* genera entropía para la generación de las claves de sesión. Este vector es propuesto por el servidor en el *handshake* inicial.

Para evitar el bloqueo debido a la pérdida de paquetes iniciales (*head-of-line blocking*[8]) se utiliza el multiplexado de *streams* en una sola conexión. El ancho de banda se debe repartir entre el número total de *streams* creados. Cada *stream* se identifica con un identificador estático, siendo las impares generadas por el cliente y las pares por el servidor, para evitar colisiones. La generación de flujos surgen de manera implícita en el inicio de la comunicación al enviar los primeros bytes, y se destruyen mediante la configuración del último paquete, indicando en un bit que no se transmitirán más. Puede haber instantes en los que haya *streams* que no son necesarios, y se finalicen sin interrumpir la ejecución del resto. La multiplexación de flujos en QUIC se basa en el encapsulado de datos en uno o más *frames*, por lo tanto, un solo paquete puede llevar datos de múltiples *streams*.

[8]Bloqueo en la entrada de los buffers debido a la pérdida de paquetes iniciales necesarios para la decodificación serializada de la información.

Forward Error Correction

Para la recuperación de paquetes sin la necesidad de utilizar retransmisiones como en TCP, QUIC implementa en una primera instancia, un esquema FEC sencillo basado en la operación XOR. Esta técnica es interesante utilizarla en redes inalámbricas, donde la latencia y la probabilidad de pérdida de información es alta.

Un paquete FEC contiene la paridad de los paquetes pertenecientes a un mismo grupo FEC[9]. En el caso de que se pierda un paquete de un grupo, su información se puede obtener a partir del paquete FEC. El servidor encargado de aportar la información puede configurar la cantidad de paquetes FEC que se mandan durante una conexión para optimizar cada escenario de comunicación. La transmisión de estos paquetes se habilita en el primer bloque de la parte de datos (Figura 2.5).

El hecho de implementar esta técnica de corrección de errores para recuperar tramas perdidas, hace que surjan distintas variantes del protocolo QUIC. En las Secciones 2.3 y 2.4 se explican dos trabajos de investigación, y las técnicas FEC que han implementado, para una posterior comparación entre ellos.

2.2. Protocolos de control de congestión

Los protocolos de congestión más relevantes actualmente son TCP New Reno y CUBIC. QUIC implementa, como mecanismo de control de la congestión, CUBIC, siendo posible combinar otros, como TCP New Reno y BBR.

Slow Start y Congestion Avoidance

Antes de entrar en profundidad con TCP CUBIC, es importante explicar dos algoritmos de control de congestión importantes en TCP Reno [7][8]: *Slow Start* y *Congestion Avoidance*.

Ambos mecanismos deben ser implementados en la parte de transmisión para controlar la cantidad de información que se inyecta a la red. La ventana de congestión en el transmisor (**cwnd**) es importante para limitar la cantidad de información que se envía antes de recibir un reconocimiento (**ACK**), mientras que la ventana de congestión en recepción (**rwnd**) regula la cantidad de datos que se reciben. Otra variable importante, denominada *slow start threshold* (**ssthresh**), es un umbral que determina el funcionamiento de *slow*

[9]Un grupo FEC se compone por la transmisión de X paquetes de datos acompañados por uno formado por la paridad (XOR) de dichos paquetes.

start y *congestion avoidance*.

Cuando se inicia la transmisión, se desconoce el estado de la red, y TCP debe comprobar la capacidad disponible de esta, para enviar datos y no congestionar el canal. El algoritmo *slow start* se utiliza para esta fase inicial y para el reinicio de la transmisión cuando se ha producido la pérdida de algún segmento después de un tiempo estipulado.

El valor inicial de **cwnd** se fija en función del tamaño máximo del paquete que puede mandar el transmisor (**SMSS**). El umbral **ssthresh** se establece a un valor inicial lo suficientemente alto, que disminuirá en función de la congestión que pueda aparecer. Por lo tanto, si el valor de **cwnd** es menor que el umbral **ssthresh**, se utiliza el algoritmo *slow start*, en cambio, si **cwnd** es mayor que **ssthresh** se inicia *congestion avoidance*. En el caso de que la ventana de congestión y el umbral tengan el mismo valor, se utilizará un mecanismo indistintamente.

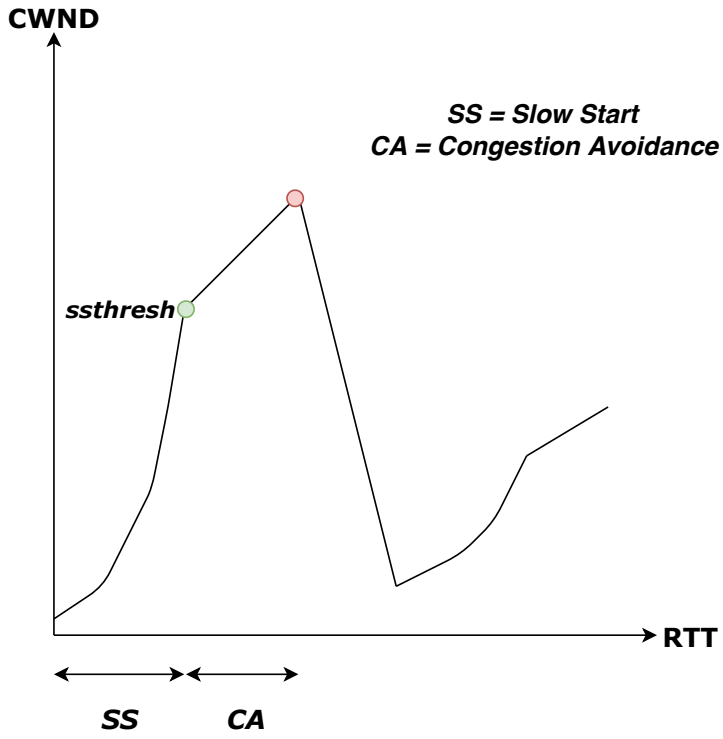


Figura 2.6: Esquema del funcionamiento de *slow start* y *congestion avoidance*.

Durante la fase de *slow start* la ventana de congestión **cwnd** aumentará doblando su valor en función de los **ACK** recibidos hasta el umbral **ssthresh** o hasta que se congestione la red. Una vez se pasa a la etapa del algoritmo *congestion avoidance*, el valor de **cwnd** aumenta en uno su valor por cada RTT. Esto sucede hasta que se produce congestión en la red, donde el cual reduce su valor hasta la mitad.

TCP CUBIC

El protocolo de control de congestión TCP CUBIC (basado en TCP BIC [7]) es el algoritmo por defecto que se utiliza en Linux y, como se ha mencionado, en QUIC. Con CUBIC [9] se pretende que el crecimiento lineal de la ventana de congestión en otros algoritmos TCP, crezca a un ritmo que dependa de una función cúbica. El principal objetivo de este cambio es obtener un protocolo que haga escalable a TCP en redes de larga distancia y velocidades altas. Por otro lado, se busca que la evolución de la ventana de congestión no dependa directamente del RTT, lo que hará que en flujos con distintos valores de RTT la ventana de congestión varíe simultáneamente.

La configuración de la ventana de congestión en TCP CUBIC (W_{CUBIC}) sigue la función cúbica que se representa en la Ecuación 2.1. En este caso, C es un parámetro prefijado de CUBIC, Δ es el tiempo que ha transcurrido desde la última vez que se produjo congestión en la red y, por lo tanto, un decremento de W_{CUBIC}. W_{max} es el valor de la ventana de congestión justo antes de que se produzca el último evento de pérdida de información y K, que se calcula a partir de la Ecuación 2.2, es el periodo de tiempo que toma W_{CUBIC} hasta adquirir el valor de W_{max}.

$$W_{CUBIC} = C \cdot (\Delta - K) + W_{max} \tag{2.1}$$

$$K = \sqrt[3]{\frac{\beta \cdot W_{max}}{C}} \tag{2.2}$$

Inicialmente, la ventana de congestión (W_{CUBIC}) evoluciona rápidamente, similar al crecimiento en *slow start*, hasta que se aproxima al valor de W_{max}, que empieza a tener un cambio más conservador. En una primera fase, el tamaño inicial de W_{max} se fija al valor que tenía la ventana de congestión justo antes de que se produjera la primera pérdida de información. En una segunda fase, después de que el tamaño de W_{CUBIC} se reduzca debido al evento de congestión, su crecimiento sigue hasta el umbral W_{max}. Por último, si se detecta una pérdida antes de alcanzar el tamaño de W_{max}, este umbral se reduce al valor que tenga la ventana de congestión en ese instante. En esta última fase, el tamaño de W_{CUBIC} varía en función de la Ecuación 2.1 (Figura 50 de [7]).

Además, TCP CUBIC garantiza tener un comportamiento mejor que el control de congestión estándar, TCP Reno. Este mecanismo consiste en asegurar que la ventana de congestión siempre esté por encima de la de Reno, por lo tanto, se calcula, aproximadamente, la ventana de congestión que se obtendría con TCP Reno, W_{Reno}. Para ello, hay que resaltar que una de las diferencias en el cálculo de ambas ventanas es que el factor β es distinto en cada protocolo de congestión. Esto hace que la ventana de congestión de TCP Reno aumente s (Ecuación 2.3) por cada RTT.

$$s = 3 \times \frac{\beta - 1}{\beta + 1} \tag{2.3}$$

Si CUBIC detectase que el tamaño de la ventana de congestión de Reno es mayor, aproximaría el valor de W_{CUBIC} a W_{Reno}.

Bottleneck Bandwidth and Round-trip time (BBR)

La diferencia principal de TCP CUBIC y TCP Reno es la forma en la que ambos mecanismos calculan la ventana de congestión, siendo el primero un algoritmo más escalable que el segundo. Aunque se observen cambios sustanciales al utilizar CUBIC, sigue apareciendo el problema de como tratar los eventos de congestión de la red. En otros protocolos se activan los mecanismos de control de congestión cuando se producen pérdidas de paquetes. En cambio, con el protocolo BBR se pretende diferenciar entre la pérdida de paquetes y la aparición de congestión en la red [10][11]. La pérdida de paquetes puede ocurrir antes de que se sature la red y no haya suficiente capacidad, sin embargo, la congestión se detecta cuando la red está ya desbordada, produciendo grandes retardos.

BBR es un protocolo de congestión desarrollado por *Google*. El objetivo principal del algoritmo BBR es asegurar que el cuello de botella[10] no se congestione, alcanzando así el máximo rendimiento con un retardo mínimo. Por lo tanto, BBR hace una estimación de la cantidad de datos que puede haber en la red (BDP) a partir de la capacidad disponible (cuello de botella) y el retardo mínimo.

El control de la transmisión de información se hace a partir de la estimación del ancho de banda disponible en cada instante. No se utiliza una ventana de congestión o el recuento de ACKs para saber la disponibilidad de la red, pero existe un límite en la cantidad de datos que pueden estar fluctuando de un extremo a extremo: 2·BDP.

El comportamiento del algoritmo BBR consta de cuatro fases diferentes [11]:

- *Startup*: esta fase adapta el comportamiento inicial en TCP CUBIC, doblando la tasa de transmisión por cada RTT. Una vez se mide el ancho de banda y este no crece más, BBR asume que se ha llegado al límite de la capacidad y se genera un retraso de un RTT, además de una cola de datos.
- *Drain*: en esta segunda fase, BBR procura drenar dicha cola para poder calcular el ancho de banda disponible inicial y el retardo mínimo, RTT_{min}, en las dos fases siguientes.

[10] Mínimo ancho de banda disponible en la red. Limita el throughput de la misma.

- *Probe Bandwidth*: en este estado se prueba a encontrar más ancho de banda disponible en varios ciclos. Se aumenta la tasa de transmisión en un periodo de tiempo, y disminuye hasta vaciar la cola.
- *Probe RTT*: una vez estimada la capacidad disponible y que no varíe el retardo durante 10 segundos, se entra en esta última fase. Se inicia la búsqueda del RTT_{min} reduciéndose el ancho de banda conseguido y drenando la cola generada. Si el retardo que se mide es igual al que se tenía previamente, hay más ancho de banda disponible que se puede utilizar. En el caso de que el retardo medido aumente respecto al que se tenía, es posible que se esté produciendo congestión en la cola de datos.

2.3. QUIC-FEC

Los autores de [4], amplían el protocolo QUIC, incluyendo distintas técnicas FEC para la recuperación de paquetes. Proponen tres contribuciones a QUIC:

- Incluir una extensión que posibilita utilizar tres técnicas FEC distintas. Además, es posible distinguir entre paquetes recibidos correctamente o aquellos que se hayan recuperado mediante el uso de FEC.
- Se aporta una implementación de las diferentes técnicas FEC en QUIC.
- Se demuestra la ventaja de utilizar técnicas FEC sobre QUIC en las transferencias de datos.

En este caso, se define el mecanismo FEC como la transmisión de redundancia *Repair Symbols*, junto con los datos para la recuperación de paquetes, *Source Symbols*. Los esquemas FEC que se recogen en este estudio son los que se basan en **códigos bloque y convolucionales** para la corrección de errores.

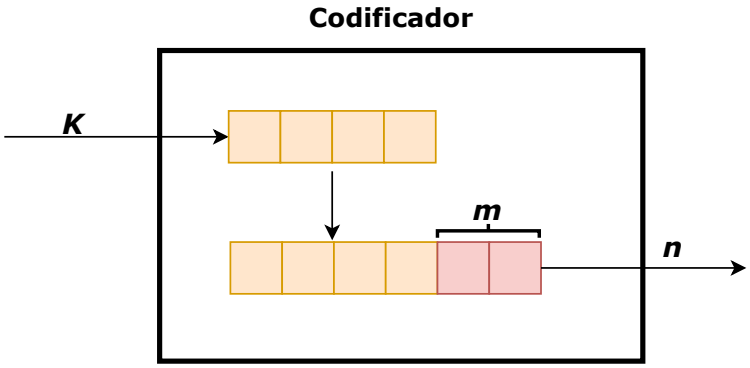


Figura 2.7: Esquema básico de los códigos bloque.

Se definen los **código bloque** como la transmisión de k *Source Symbols* y $m = n - k$ *Repair Symbols*. Se agrega redundancia a cada bloque de símbolos, siendo $n{>}k$ (Figura 2.7). Por ejemplo, un código (6, 4) define que por cada bloque de 4 *Source Symbols* se protegen con 2 *Repair Symbols*. En QUIC-FEC, se trabaja con dos tipos de códigos bloque: **XOR** y **Reed-Solomon**.

- Esquema XOR: A los *Source Symbols* se les aplica diréctamente la operación XOR, generando así los bits de redundancia necesarios (*Repair Symbols*). Su implementanción es sencilla, pero su algoritmo solo permite recuperar un solo *Source Symbol* de cada bloque. Por lo tanto, se utiliza el intercalado, que consiste en transmitir paquetes sucesivamente en diferentes bloques FEC haciendo que la XOR funcione eficientemente.
- Reed-Solomon: Es un tipo de códigos bloque que aporta una mayor velocidad de codificación y son capaces de corregir más de un error en una secuencia binaria. Este mecanismo permite recuperar más de un paquete en el mismo bloque FEC, aunque computacionalmente sea más complejo.

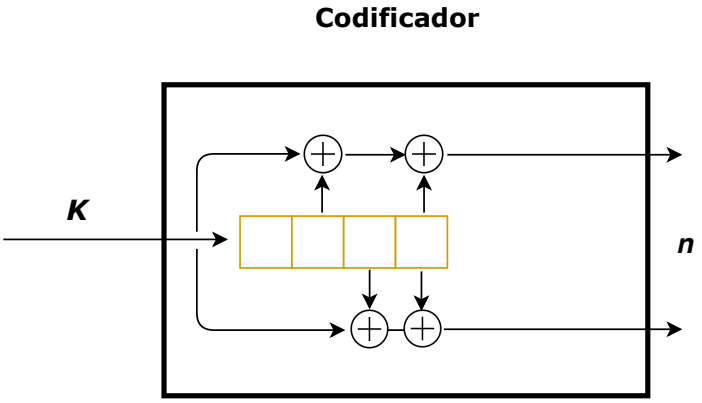


Figura 2.8: Esquema básico de los códigos convolucionales).

Los **códigos convolucionales** (Figura 2.8) tienen el mismo objetivo que los códigos bloque, con la característica adicional de que tienen memoria, es decir, la codificación actual depende de los datos enviados en ese instante y en los anteriores. En este tipo de codificación se definen tres parámetros distintos: $m{=}n - k$ *Repair Symbols* se transmiten por cada k *Source Symbols*. Los símbolos redundantes se transmiten para proteger los c *Source Symbolos* anteriores. Por ejemplo, si se tiene un código convolucional (3, 2, 4), se envía un *Repair Symbol* por cada dos *Source Symbols*. Esto significa que los símbolos de redundancia se mandan intercalados con los símbolos útiles de datos, protegiendo así los cuatro *Source Symbolos* anteriores. En este caso, se utiliza el **Random Linear Codes** como código convolucional.

- Random Linear Codes (RLC): es un tipo de código convolucional y tiene como ventaja la baja latencia que se obtiene en el momento de la recuperación de paquetes. La extensión que se utiliza en QUIC-FEC propone aplicar ecuaciones lineales con los *Source Symbols* perdidos.

El mecanismo que se utiliza en este estudio en referencia a la manipulación de los paquetes cuando se produce una pérdida o congestión es la siguiente:

- El paquete no se protegió mediante técnicas FEC: en este caso, si este paquete se pierde no se recuperará. Los paquetes FEC (paquetes de redundancia) son los que se tratan en este caso.
- El paquete ha sido protegido mediante FEC, pero no ha podido recuperarse: el emisor se da cuenta de la pérdida y retransmite el *frame* correspondiente.
- El paquete fue protegido mediante FEC y recuperado: la confirmación de la recuperación de estos paquetes puede producir la omisión de la señal de congestión al ocupar más ancho de banda. Esto se puede evitar de tres maneras distintas:
  - No mandar un *ACK* cuando se recupera el paquete. Esto producirá que el paquete recuperado se trate como perdido y el transmisor utilice retransmisiones innecesarias.
  - Distinguir los paquetes recuperados por pérdidas en el canal y los perdidos por congestión. La pérdida por congestión produce un aumento del RTT, por lo tanto, si las pérdidas ocasionadas por el canal se tratan igual, la latencia aumenta considerablemente. En cualquier caso, independientemente del tipo de pérdida, se produce un ajuste en el ancho de banda.
  - Notificar explícitamene al emisor de la recuperación de un paquete. Es la solución que se implementa en QUIC-FEC. El paquete se confirma con un paquete QUIC ACK y se señala que ha sido recuperado. El transmisor adapta la ventana de congestión y elimina el paquete en cuestión de la cola de retransmisión.

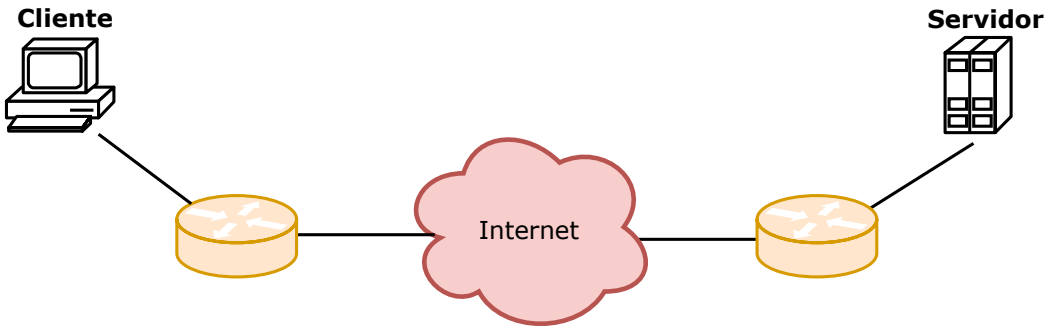


Figura 2.9: Topología de red emulada de los experimentos de QUIC-FEC.

Los experimentos realizados se basan en analizar la implementación y las ventajas obtenidas con el uso de técnicas FEC en el protocolo QUIC. Para esto, se ha obtenido el tiempo total de descarga (*Download Completion Time*, DCT) requerido para completar una transferencia HTTP. Las medidas se realizan con diferentes modelos de pérdidas y se utiliza un modelo *Gilbert-Elliott* para representar ráfagas de paquetes perdidos. El modelo Gilbert Elliott se basa en una cadena de Markov de dos estados: *Good* y *Bad*. En el estado *Good*, la probabilidad de que se transmita un paquete es k y en el estado *Bad* la probabilidad de emitir un paquete es h. La probabilidad de transición del estado *Good* al *Bad* es p y del estado *Bad* al *Good* es r.

La topología de red que se utiliza en la red simulada que han expuesto es la de la (Figura 2.9). Además, los parámetros (Tabla 2.1) de dicha red se basan en dos tecnologías: *Direct Air-to-Ground Communications* (DA2GC)[11] y *Mobile Satellite Service* (MSS)[12] [12].

Tabla 2.1: Parámetros de las redes emuladas.

Tecnología	Ancho de banda (Mbps)	RTT (ms)	Probabilidad de pérdida (%)
DA2GC	0.468	262	3.3
MSS	1.89	761	6

Los resultados iniciales se basan en la comparación de QUIC nativo con QUIC-FEC con un modelo de pérdidas uniforme y probando los mecanismos de recuperación de paquetes Reed-Solomon (30, 20) y RLC (3, 2, 20). Los tamaños de los ficheros que se van a transmitir para medir el DCT de una transferencia HTTP son de 1 kB, 10 kB, 50 kB y 1 MB.

Para los parámetros de la tecnología DA2GC se observa que para la transferencia de ficheros de tamaño 1 kB, el uso de FEC implica una leve o nula mejora del DCT respecto a QUIC puro. Esto se debe a que la cantidad de datos que se transmite no son suficientes para saturar la ventana de congestión del transmisor. Sin embargo, para ficheros de tamaños mayores la utilización de técnicas de corrección de errores perjudica el parámetro DCT, debido a que el poco ancho de banda usado en esta tecnología no es suficiente para transmitir datos de redundancia. Los experimentos con los parámetros de la tecnología MSS muestran como resultado que, con la transferencia de ficheros de pequeño tamaño y la aplicación de técnicas FEC, se reduce el DCT total. El RTT y la probabilidad de error son lo suficientemente altos para perjudicar gravemente el DCT, pero con FEC se utilizan los beneficios que aporta un ancho de banda mayor para mandar los bits de redundancia necesarios.

En conclusión, los resultados para DA2GC aportan un beneficio para el 25 % de los

[11] Es la implementación de tecnología Wi-Fi en aviones.
[12] Servicio móvil por satélite.

experimentos realizados en ficheros de 1 kB aplicando FEC. En cambio, para la tecnología MSS se observa ventaja en un 75 % de los experimentos con ficheros de 50 kB. El uso de técnicas FEC aporta ventajas respecto al DCT si se utiliza un ancho de banda amplio para la transmisión de paquetes de redundancia y así no limitar la ventana de congestión del transmisor.

Otros resultados obtenidos por este grupo han sido basados en la configuración de la red con otros parámetros, basándose en el modelo *Gilbert-Elliott* y aplicando las técnicas FEC. Se sigue observando el compromiso existente entre el patrón de pérdidas y la necesidad de tener un ancho de banda que pueda soportar la redundancia necesaria para la recuperación de paquetes.

Por último, se comparan las diferentes técnicas FEC implementadas en el código QUIC-FEC con la transferencia de ficheros de distinto tamaño. Se han hecho los experimentos utilizando como código bloque *Reed-Solomon* y como código convolucional RLC. Se puede ver como el uso de *RLC* para ficheros de 1 MB da lugar a un mejor resultado que con la técnica *Reed-Solomon*. El DCT disminuye con el *RLC* debido a la forma en que se transmite los paquetes *Repair Symbols*. *Reed-Solomon* manda los *frames* FEC después del bloque entero de información, en cambio *RLC* envía los paquetes FEC intercalados con los paquetes de datos de un mismo bloque. Es decir, si los parámetros para *Reed-Solomon* son (30,20), cada 20 paquetes de datos protegidos por FEC, se mandan 10 paquetes *Repair Symbols*: si se pierde el primer paquete de un bloque, hay que esperar hasta la transmisión de 19 paquetes más para poder recibir los paquetes FEC y recuperar dicho paquete. Por otro lado, con los parámetros de *RLC* de (3,2,20), cada transmisión de 2 paquetes de información se manda uno FEC, siendo así más rápida la recuperación de un paquete perdido.

Debido a que las pérdidas no son constantes y a que el ancho de banda no es siempre el mismo, surge la necesidad de hacer un FEC adaptativo y así no perjudicar el DCT. Se propone variar la tasa FEC, es decir, cada cuantos paquetes de datos se mandan paquetes FEC. El problema que surge es la aparición de sobrecarga en la transmisión excesiva de paquetes FEC. Por otro lado, en [4] aconsejan el uso de estas técnicas de corrección de errores para transmisiones cortas de información y grandes anchos de banda.

2.4. rQUIC

En [3] se diseña e implementa rQUIC: integración de técnicas FEC en el protocolo QUIC, para reducir el tiempo de recuperación de paquetes en las comunicaciones inalámbricas. Se evalúa dicha solución sobre diferentes tipos de redes inalámbricas y aplicaciones. Las principales contribuciones de este documento se presentan a continuación:

- La introducción de *Forward Error Correction* en el protocolo QUIC para comunicaciones inalámbricas robustas. Para demostrar las ventajas que aporta rQUIC, se diseña e implementa un algoritmo FEC: XOR adaptativo. De esta forma se pretende respetar la ventana de congestión.
- Para examinar los beneficios que aporta rQUIC respecto a QUIC, se configurarán diferentes parámetros de redes: ancho de banda, probabilidad de pérdida y RTT. Se cogerán muestras a partir de dos aplicaciones distintas: transferencia de información web y transferencia de ficheros.
- Se diseña un *testbed* inalámbrico para complementar las emulaciones con experimentos en mundo real y así realizar análisis de rQUIC en redes WiFi (IEEE802.11) y LTE.
- Para seguir con el desarrollo y evolucionar su algoritmo, se aporta el código fuente de rQUIC. Se basa en una implementación abierta de QUIC en *go*[13].

Para reducir la latencia y que sea transparente su implementación en QUIC, rQUIC introduce un algoritmo basado en la operación XOR. Al paquete QUIC nativo se le añade una cabecera más, donde se activa la técnica FEC.

Es importante destacar que rQUIC inicialmente encripta la parte de información útil del paquete QUIC, y posteriormente codifica con FEC. Una sesión QUIC produce una única conexión extremo a extremo. Si ambos lados permiten la transmisión de información, la sesión QUIC genera paquetes QUIC, que serán encriptados y autenticados. Los paquetes encriptados son introducidos al módulo rQUIC, produciendo el paquete rQUIC añadiendo el campo de *cabecera FEC* y aplicando el algoritmo FEC correspondiente dependiendo del tipo de paquete (Figura 2 de [3]).

La cabecera rQUIC se puede activar o desactivar mediante las *flags* de la cabecera QUIC o negociar en el establecimiento inicial de la conexión. La cabecera FEC se compone de 4 bytes:

- Tipo: se representa mediante 1 byte. Identifica si el paquete está protegido o no por FEC, o si es directamente un paquete FEC. Los valores que adquiere son los siguientes:
  - 0x80: El paquete está protegido. El módulo FEC protege paquetes que al menos tenga un bloque de datos.
  - 0x00: El paquete no está protegido. Los paquetes que no son respaldados por FEC son aquellos que llevan información de control: por ejemplo paquetes de confirmación de tramas (ACK), que son mandados periódicamente.

[13]Lenguaje de programación desarrollado por Google.

- 0xC0: Es un paquete FEC. Se utiliza para la recuperación de paquetes.

- Bloque de identificación: se utiliza 1 byte para representarlo. Identifica aquellos paquetes que van protegidos por el mismo paquete FEC, es decir, paquetes que pertenecen al mismo bloque de datos.

- *FEC Ratio*: es un parámetro importante para el algoritmo que se expone en este documento, y se utilizan 8 bits para declararlo. El paquete FEC se genera después de *FEC Ratio* paquetes QUIC.

- *Count*: si es un paquete protegido, este campo indica el orden del paquete en el bloque FEC. Se representa con el último byte de la cabecera FEC.

Es importante el método que se utilice para codificar o decodificar, debido a que esta acción implica un aumento de la latencia en una conexión, lo cual quiere reducir QUIC. Por lo tanto, el diseño de un algoritmo FEC se debe adaptar a las características de la conexión, proporcionando un compromiso entre el ancho de banda disponible y el incremento de *overhead*. El algoritmo FEC es adaptativo, y se basa en la operación XOR.

Algoritmo FEC: La principal operación que se realiza se basa en la operación lógica XOR. Su funcionamiento consiste en introducir por cada n paquetes de datos un paquete extra de redundancia, siendo este paquete extra la XOR de los n paquetes de datos anteriores. La ventaja que se obtiene con este método es una baja complejidad computacional y, por lo tanto el tiempo que se invierte en codificar y decodificar es mínimo. Por otro lado, la desventaja de aplicar un algoritmo tan sencillo es que solo se puede recuperar un paquete por bloque. En el caso de que hubiera múltiples pérdidas, el FEC XOR no impide el mecanismo de recuperación de paquetes clásico, consumiendo una gran cantidad de recursos. El hecho de sufrir este defecto, se estudia la adaptación de la tasa FEC, es decir, la cantidad de paquetes que se mandan en cada bloque FEC en función de las retransmisiones necesarias.

FEC adaptativo: El principal objetivo de aplicar una tasa adaptativa del FEC es que se reduzca el *overhead* cuando no haya pérdidas y se incremente la redundancia cuando sucedan.

El algoritmo propuesto se basa en las pérdidas residuales, es decir, pérdidas que se producen debido al fallo del FEC. Para calcular las pérdidas residuales (ϵ) se hacen a partir de los paquetes transmitidos y retransmitidos en un periodo de tiempo i y de duración T:

$$\epsilon_i = \frac{\text{retransmisiones}}{\text{transmisiones - retransmisiones}} \tag{2.4}$$

Las medidas de pérdidas residuales se representan como la media en un periodo N:

$$\overline{\epsilon_i} = \frac{\sum_{i=1}^{N} \epsilon_i}{N} \tag{2.5}$$

A partir de las pérdidas residuales se calcula la tasa FEC adaptativa que se utiliza durante una conexión. Dicha tasa sigue el siguiente algoritmo:

r = r_{init}
if $\overline{\epsilon} > \gamma$ **then**
| r = r × (1 - δ)
else
| r = r × (1 + δ)
end

Algoritmo 1: Cálculo de la tasa FEC adaptativa

El valor de la tasa FEC, r, depende del valor de las pérdidas residuales. Si las pérdidas residuales, $\overline{\epsilon_i}$, adquieren un valor mayor que el de un umbral γ, la tasa FEC incrementa en función de un parámetro de correción, δ. Estos parámetros son configurables y determinan la tolerancia de las técnicas FEC en cuanto a la recuperación de tramas.

La implementación de rQUIC se basa en una implementación de código abierto en *go* (v0.7.0). Aunque QUIC siga la especificación del IETF, en este trabajo se utiliza la última versión estable de Google. rQUIC utiliza los paquetes QUIC encriptados y les añade la cabecera FEC y, si son *streams* de datos, los codifica con el algoritmo seleccionado. Por lo tanto, los paquetes generados pueden ser paquetes QUIC no codificados, paquetes protegidos por FEC (codificados) o paquetes FEC donde se aloja la redundancia para la recuperación de paquetes. Estos últimos también son tratados en el protocolo de congestión, por lo que rQUIC respeta la tasa de envío en función de la ventana de congestión.

Como algunos paquetes van codificados, en el lado del receptor debe haber un decodificador FEC. Si un paquete protegido se pierde, el resto que pertenezcan al mismo bloque (mismo número del grupo FEC en la cabecera), se detendrán hasta que se produzca uno de los siguientes eventos:

1. Si se pierde otro paquete del mismo bloque FEC: El receptor reportará un fallo en la decodificación y todos los paquetes correspondientes a dicho bloque son reenviados a la capa de sesión de QUIC.
2. No hay más pérdidas en el mismo bloque, y se recibe el paquete FEC: El decodificador será capaz de recuperar el paquete extraviado y se mandarán todos los paquetes a la capa superior de sesión.
3. Se recibe un paquete de un bloque FEC distinto: Se mandan a la capa de sesión QUIC todos los paquetes retenidos y se asume un fallo en la decodificación. Esto evita retrasos o posibles retransmisiones de los paquetes parados.

Es importante recuperar el número de paquete para que el mecanismo de detección de pérdidas cuente los diferentes paquetes que se han recibido. Si un paquete falta y se recupera a través de rQUIC, el mecanismo de detección no lo percibe. Dicho mecanismo manda reconocimientos de todos los paquetes que le llegan, incluso si son FEC. Los paquetes FEC son percibidos por el control de congestión también como un paquete QUIC normal, por lo que la ventana de congestión aumentará o reducirá. El problema surge cuando un paquete FEC se pierde: dicho paquete no se retransmitirá, pero sí afectará a la ventana de congestión.

Para demostrar las ventajas de rQUIC se utiliza el simulador de redes ns-3. Con ns-3 se puede conectar el tráfico de información de las redes reales sobre una red simulada. En este caso, el entorno de simulación se basa en una aplicación cliente-servidor, donde ambos se ejecutan en contenedores Linux[14]. Dichos contenedores se conectarán entre sí a través de la red simulada en el ns-3. Los contenedores son vistos como nodos conectados a través de una red CSMA a un router (Figura 5 de [3]). La conexión entre los routers es punto a punto y la red se puede configurar con los parámetros de ancho de banda, RTT y probabilidad de pérdida. Esto permitirá la representación de distintas tecnologías de red: WiFi/LTE con un RTT de 25 ms y un acho de banda de 20Mbps, 2G/3G con un retraso de 50 ms y 10Mbps de ancho de banda y por último los enlaces satelitales con un RTT alto como es 400 ms y un ancho de banda bajo, 1.5Mbps. La probabilidad de pérdida varía entre el 0 y 5 % en todas las tecnologías.

Las aplicaciones con las que se va a caracterizar rQUIC son dos, la tranferencia masiva de datos y el tráfico en la web, usando HTTP/2, y se comparará el comportamiento de QUIC nativo frente a rQUIC. Para la transferencia masiva de información, el cliente descarga del sevidor ficheros de 20MB para redes como WiFi/LTE y ficheros de 5MB para los enlaces satelitales y 2G/3G. Para la transferencia de archivos web se utilizarán algunas páginas web, donde la compresión de datos se estima alrededor de 2KiB.

Las medidas que se efectuarán para comprobar las ventajas de rQUIC son las siguientes:

Completion Time Ratio (ξ): Este parámetro representa el tiempo total que se requiere para la descarga de información(DCT). El *ratio* de este parámetro define el tiempo total de descarga entre rQUIC y QUIC nativo. En la Ecuación 2.6 se calcula el *completion time ratio* como la división entre el tiempo requerido en rQUIC y la media de los tiempos obtenidos con QUIC nativo después de 100 iteraciones. Si ξ es menor que 1, se demuestra la ganancia que se obtiene con el uso de rQUIC sobre QUIC.

$$\xi = \frac{\text{Completion Time rQUIC}}{\text{Completion Time QUIC}} \tag{2.6}$$

[14]Se basa en el concepto de virtualización de redes. Los contenedores son máquinas virutales que comparten el *kernel* del sistema operativo donde se implementa.

Overhead: El *overhead* o sobrecarga, define la cantidad de datos redundantes (paquetes FEC) que son transmitidos en una sesión rQUIC. Se representa como la cantidad de datos transmitidos en una sesión.

En las medidas iniciales del ξ, se utilizaron varios valores de los parámetros del Algoritmo 1, $\delta \in \{0.25\,\%, 0.5\,\%, 1\,\%, 3\,\%\}$ y $\gamma \in \{0.25, 0.33, 0.5, 0.75\}$. Debido a que no se observan grandes cambios respecto al tiempo de descarga, y que para valores del parámetro de correción δ más altos se observa una mejor respuesta, se fijan $\gamma = 0.33$ y $\delta = 1\,\%$.

Los resultados experimentales del tiempo de descarga (DCT) y el *overhead* producido por paquetes FEC para la transferencia masiva de información se representan en la Figura 9 de [3]. Se puede ver como el hecho de ajustar dinámicamente la transmisión de paquetes FEC trae consigo un claro beneficio sobre el protocolo QUIC nativo. Para características de red similares a las de las tecnologías WiFi/LTE se observa una ganancia del 60 %, y un 50 % para características de enlaces satelitales. Por otro lado, a medida que se aumenta la probabilidad de error del canal, la cantidad de sobrecarga es mayor, debido a la adaptación de la tasa de FEC, que hace que aumente la transmisión de bits de redundancia para la recuperación de paquetes perdidos. Tal y como se suponía, para pérdidas del 0 %, no se observa ninguna ventaja de rQUIC respecto a QUIC, debido a que no entra en funcionamiento el algoritmo FEC.

Para el tráfico de internet, se puede observar los mismos resultados, Figura 10 de [3]. Este tráfico se caracteriza por la transmisión de objetos de pequeño tamaño. Para las características de redes móviles 2G/3G/LTE y tecnología WiFi, se puede ver una mejor respuesta con el uso de rQUIC. Para redes con un retardo notable, la utilización de FEC no trae consigo un beneficio claro, debido a mala respuesta que se observa en la fase de *slow start* del protocolo de congestión. Esto se debe a que la ventana de congestión crece rápidamente antes de que el transmisor vea la primera pérdida de paquetes. Respecto a la sobrecarga, se observa una respuesta similar a la aplicación anterior, pero con valores más pequeños. Esto se debe a que la mayoría de transmisiones HTTP/2 se producen en la fase de *slow start*, donde no se suceden pérdidas.

Aunque la implementación de rQUIC aporte grandes ventajas, hay aspectos que se han observado y son importantes tratar para la inclusión de otras técnicas FEC.

1. El valor de la tasa FEC está limitado por 2 y 255. En el caso de que sea 2 implica que solo se transmiten paquetes codificados y en el caso de adquirir el valor de 255 es porque está limitado al tamaño del campo FEC (1 byte) de la cabecera. Otro limitación que puede aparecer es la ventana de congestión: la tasa FEC no puede superar el valor de la ventana, debido a que de esta manera se previene el bloqueo de un mismo bloque FEC.

2. El valor inicial de la ventana de congestión es 32 paquetes y el umbral (*ssThreshold*) de la fase de *slow start* no está definido, por lo tanto este estado se mantiene hasta

que se produce la primera pérdida. En este estudio se observa el comportamiento de rQUIC en el inicio de la conexión, viendo que la ventana de congestión crece, pero la tasa FEC no se adapta adecuadamente, lo que hace que sea conveniente deshabilitar el FEC en la fase de *slow start*, ya que no se producen pérdidas hasta que finalice esta.

3. Si se recibe un paquete del bloque siguiente al que se está recibiendo sin completarse, se produce un fallo en la decodificación, y todos los paquetes que estaban retenidos se transmiten a la capa de sesión de QUIC desordenadamente.

3

Entorno y desarrollo de simulación

En el presente capítulo se describe el entorno de simulación, así como las herramientas que se han utilizado para desplegarlo. Es importante resaltar que la red simulada se ha realizado a través de la herramienta *ns-3* y los nodos se han creado mediante contenedores linux.

Por otro lado, los resultados que se obtienen en el Capítulo 4 se basan en el desarrollo de los repostiorios QUIC vistos en [3] y [4]. Por esto, se explicarán los cambios realizados en cada código.

3.1. Herramienta *ns-3* y contenedores Linux

El simulador *ns-3*[15] es un simulador de red basado en eventos y su principal uso se encuentra en entornos académicos y de investigación. Fue creado en 2006 y es un proyecto de código abierto basado en lenguaje C++.

ns-3 aporta distintos modelos de redes. Esta herramienta puede ser útil para el desarrollo de escenarios que no son factibles en entornos reales, y así estudiar su comportamiento de una forma controlada. Los modelos de redes disponibles en *ns-3* están enfocados principalmente en como funciona Internet y en los protocolos que utiliza, pero esta herramienta no se limita solo a entornos de Internet, y cada usuario puede desarrollar diferentes sistemas

[15] https://www.nsnam.org/docs/release/3.30/tutorial/ns-3-tutorial.pdf

para otro tipo de redes.

Una de las ventajas del diseño de *ns-3* respecto a otros simuladores es que se basa en una gran variedad de librerías, que pueden ser combinadas tanto entre ellas como con otras externas al simulador. Además, aporta distintas herramientas de animación, análisis de datos y visualización de los mismos. El usuario debe realizar la configuración a través de la línea de comandos y usar herramientas de desarrollo de software en código C++ o Python. Adicionalmente, este simulador puede operar en distintos sistemas operativos como son *Linux*, *macOS* o *Windows*, siempre y cuando este último tenga soporte para trabajar con el entorno de *Linux*.

Otras de las herramientas importantes con la que se trabaja son los contenedores Linux (LXC). Es una tecnología de virtualización[16] que permite a los usuarios crear y administrar contenedores de sistemas o aplicaciones. En este caso simulan nodos Linux. El objetivo de los contenedores LXC es la creación de un entorno lo más cercano posible a una instalación Linux estándar sin la necesidad de utilizar un *kernel* distinto.

Topología del escenario

Para la recogida de medidas el escenario que se ha dispuesto se basa en dos nodos utilizando los contenedores Linux (*source* y *receiver*) y una red simulada mediante *ns-3*. Para la creación de los dos nodos se instala la herramienta LXC en el equipo y se determinan los ficheros de configuración (*lxc-source.conf* y *lxc-receiver.conf*) donde se determinan los siguientes campos:

1. "*lxc.uts.name*": especifica el nombre del contenedor o nodo. En este caso *source* y *receiver*.
2. "*lxc.net.*[i].*type*": define el tipo de red virtual que usan los contenedores. En ambos nodos se fija a *veth*, es decir, una red ethernet virtual.
3. "*lxc.net.*[i].*flags*": sirve para activar la interfaz del nodo, se fija a *up*.
4. "*lxc.net.*[i].*link*": especifica la interfaz que se utiliza para el tráfico de la red. En este caso se utilizan los *bridges*[17] y se denominan *br-source* y *br-receiver*. Estos componentes se conectarán a los correspondientes TAPs[18] para que los paquetes se puedan mandar de un lado a otro de la red.
5. "*lxc.net.*[i].*ipv4.address*": se configura la dirección IPv4 de la interfaz virtual de cada contenedor. Estas direcciones son las mismas que se especifican en la red *ns-3*.

[16] Creación de una versión software virtual de recursos teconlógicos.
[17] Puentes de red para conectar ambos lados de la red.
[18] Componente en la red para separar la entrada y salida de datos.

Antes de generar ambos lados de la red, se crean y activan dichos *bridges* y se emparejan con cada TAP correspondiente. En la Tabla 3.1 se muestran los comandos ejecutados para la creación de los componentes intermedios de la red.

Tabla 3.1: Creación y conexión de *bridges* y TAPs.

Creación de los *bridges*
sudo brctl addbr **br-source**
sudo brctl addbr **br-receiver**
Creación y asignación de las IPs de los *TAPs*
sudo tunctl -t **tap-source**
sudo tunctl -t **tap-receiver**
sudo ifconfig **tap-source** 0.0.0.0 promisc up
sudo ifconfig **tap-receiver** 0.0.0.0 promisc up
Conexión de los *bridges* con su respectivo TAP
sudo brctl addif **br-source tap-source**
sudo ifconfig **br-source** up
sudo brctl addif **br-receiver tap-receiver**
sudo ifconfig **br-receiver** up

Por último, se crean los nodos que actuarán de servidor (*source*) y cliente (*receiver*) a través de los contenedores LXC. Para esto, se utilizan los ficheros de configuración descritos anteriormente y se ejecutan los comandos de la Tabla 3.2 tanto para la generación de los nodos y activación de los mismos.

Tabla 3.2: Creación de los nodos *source* y *receiver*.

Nodo servidor (*source*)
sudo lxc-create -f **lxc-source.conf** -t download -n **source** - - -d ubuntu -r trusty -a amd64
sudo chrrot /var/lib/lxc/**source**/rootfs/ passwd
sudo lxc-start -n **source** -d
sudo lxc-attach -n **source**
Nodo cliente (*receiver*)
sudo lxc-create -f **lxc-receiver.conf** -t download -n **receiver** - - -d ubuntu -r trusty -a amd64
sudo chrrot /var/lib/lxc/**receiver**/rootfs/ passwd
sudo lxc-start -n **receiver** -d
sudo lxc-attach -n **receiver**

Para que los nodos se puedan comunicar se necesita la red que se crea a través de *ns-3*. En la Figura 3.1 se representa el esquema. La programación de la red consiste en

generar tres redes distintas formadas por los siguientes dispositivos: entre el nodo servidor y el $router_0$, la red entre los $router_0$ y $router_1$, y la tercera red formada por el $router_1$ y el nodo cliente. En el nodo cliente y en el nodo servidor se instala la clase *tapBridge* en el modo *UseBridge*, para conectar con ambos contenedores Linux.

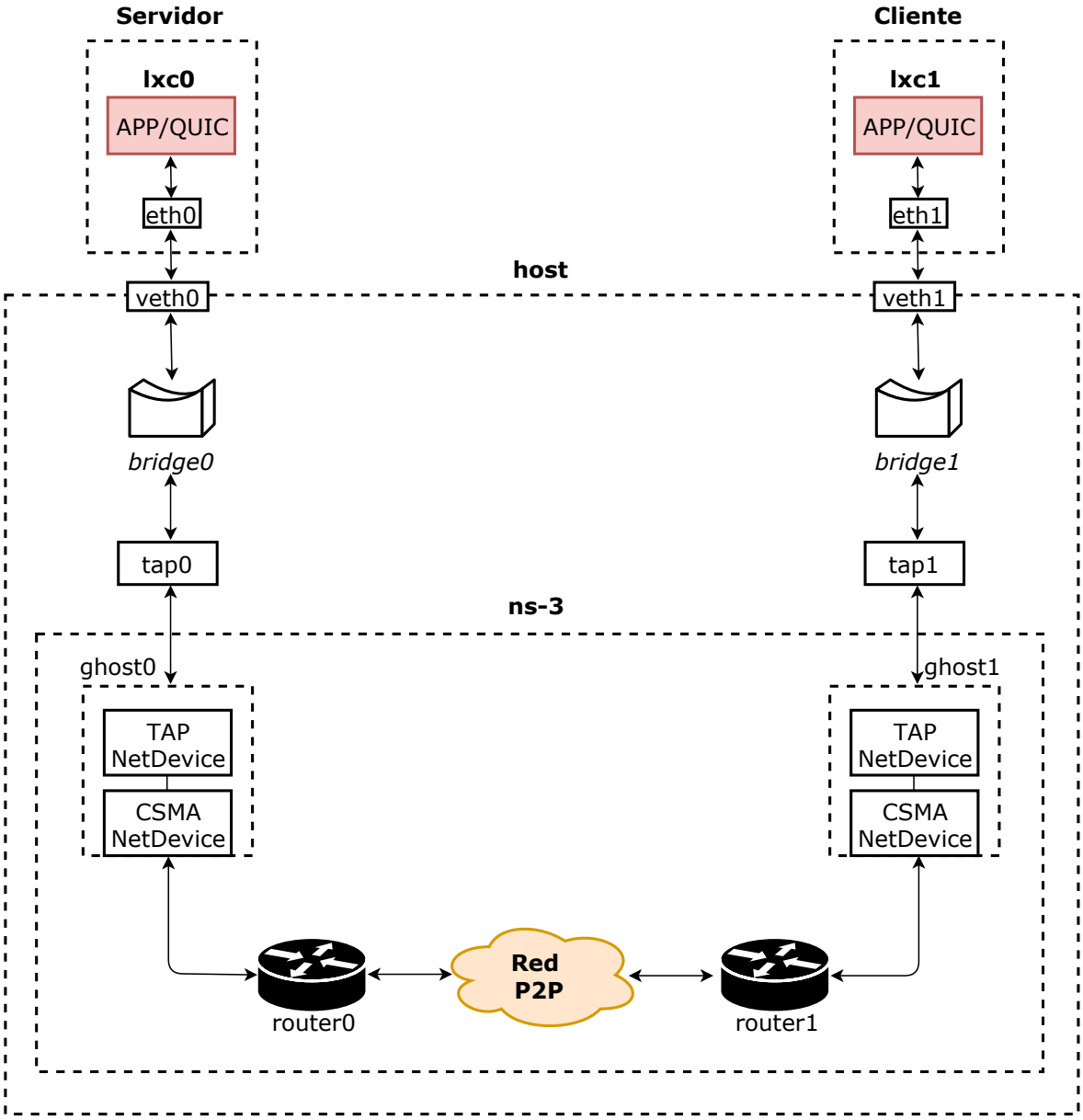


Figura 3.1: Topología de la red.

El motivo por el que se crean tres redes distintas es porque los *TAP-Bridge* solo se pueden conectar a redes CSMA[19], y si se conectasen el cliente y el servidor a partir de

[19]Protocolo de control de acceso al medio: el usuario verifica que no hay tráfico en la red para transmitir.

una red como esta, no se podrían modificar ancho de banda y latencia simultáneamente debido a que una depende de la otra en este tipo de despliegues. Para que se puedan configurar dichos parámetros, el servidor y el cliente se conectan al $router_0$ y al $router_1$ respectivamente, con una red CSMA, y ambos routers se acoplan a través de una red P2P[20] que permite configurar ancho de banda y latencia. Para que las redes CSMA sean transparentes al resto, se fija la capacidad a 10Gbps y el retardo a 0 ms, mientras que la red P2P aporta una capacidad bastante menor.

Por último, para que la topología de red sea lo más realista posible, se configuran la capacidad de los routers al *Bandwidth Delay Product* (BDP). Esto hace que la capacidad de los routers no sea infinita, pero que tampoco se produzca un cuello de botella debido a la congestión en los nodos. El valor del parámetro BDP se calcula a partir del producto de la capacidad del enlace (C) por el retardo del mismo (RTT), Ecuación 3.1.

$$BDP(\text{bits}) = C(\text{bits/s}) \cdot RTT(\text{s}) \tag{3.1}$$

La comunicación entre los contenedores LXC y el host principal se realiza a través de las carpetas compartidas de cada nodo. De esta manera se pueden mover archivos de forma sencilla.

3.2. Lenguaje GO

La implementanción de QUIC que se utiliza en este trabajo está realizada en el lenguaje de programación GO. Es un proyecto de código abierto y está desarrollado por *Google*. La instalación de este se realiza en el host principal y en los contenedores LXC de tal forma que se puedan ejecutar programas GO en ellos. Para la instalación del entorno de GO se introducen los comando de la Tabla 3.3.

Tabla 3.3: Instalación del entorno de GO.

Descarga de GO
> wget https://redirector.gvt1.com/edgedl/go/go1.9.2.linux-amd64.tar.gz > sudo tar -C /usr/local -xzf go1.9.2.linux-amd64.tar.gz
Añadir GO a la ruta en *.profile*
> echo "export PATH=$PATH:/usr/local/go/bin»> ~ /.profile > source ~ /.profile
Versión de GO
> go version

[20]Es una red que se crea a partir de nodos que se comportan iguales entre sí.

Como herramienta principal para el uso y desarrollo de QUIC nativo y sus variantes se utiliza *github*. En el host princial, en el directorio de GO, se ubica el repositorio donde está programado QUIC en su versión original y las variantes con la inclusión de técnicas FEC (rQUIC, QUIC-FEC y QUIC). Para realizar cambios, utilizar versiones específicas de la implementanción del protocolo u otras acciones, se utilizan los comandos *git* del entorno *github*[21].

3.3. rQUIC tasa FEC fija y QUIC-FEC

Para comparar las distintas implementaciones de técnicas FEC de [3] y [4] se consideran varios aspectos. Para que sea un análisis coherente, se modifica rQUIC para que la tasa FEC no se adapte en función del Algoritmo 1, y se configure a un valor fijo. Esta nueva configuración es práctica para realizar una comparativa justa entre QUIC-FEC, que inicialmente no establece una adaptación de la tasa FEC, y rQUIC.

Para la modificación del código y establecer una tasa fija de la transmisión de redundancia, se trabaja con el repositorio rQUIC[22]. El objetivo es que cuando se utilice una aplicación sobre QUIC, el periodo de transmisión de paquetes FEC sea constante. Para esto, en el programa principal donde se implementa el codificador FEC (*fec_encoder.go*), se suprime la línea de código donde se llama a la función que dinamiza la tasa FEC: *go f.dynamicRatio.StartTimer*. En esta función, *StartTimer*, se calculan las pérdidas residuales (ϵ) y, en función del valor de estas, se aplica el Algoritmo 1 a partir de la función *UpdateRatio*.

```
aux = 0
if CWND < RatioFEC then
    aux = CWND
    if aux < 2 then
        aux = 2
    end
else
    aux = RatioFEC
end
ChangeFecRatio(aux)
```

Algoritmo 2: Tasa FEC fija.

Por otro lado, debido a que en este estudio se respeta el valor de la ventana de congestión, es importante limitar el periodo de transmisión de paquetes FEC, para que no surja un comportamiento inestable en la comunicación y se bloquee la transmisión en mitad de un bloque FEC. En el programa de configuración del código base de QUIC *session.go*, don-

[21] github.com/dasdo/9ff71c5c0efa037441b6.
[22] github.com/pgOrtiz90/quic-go-fec/tree/quic-fec

de se comprueba si hay que transmitir paquetes FEC, se añade la condición de que la tasa FEC permanezca fija, siempre y cuando no se supere el valor de la ventana de congestión. Esto quiere decir que el periodo de transmisión de paquetes FEC debe adquirir siempre el valor mínimo entre la ventana de congestión (**CWND**) y la tasa FEC (**RatioFEC**). En el Algoritmo 2 se especifica el valor que adquiere la tasa FEC, *ChangFecRatio()*.

Para la obtención de resultados, en el repositorio QUIC-FEC[23] se implementa la aplicación *bulk* que consiste en la transmisión masiva de información entre un cliente y un servidor. En el análisis que se hace en [4], los autores determinan que la utilización de códigos convolucionales como técnica FEC es la más eficaz, pero debido a que en [3] solo está implementado el mecanismo XOR, se programa la configuración de QUIC para que utilicen el mismo. Para esto, en la parte del servidor se especifica el esquema FEC a seguir a través de la *flag fecSchemeFlag* y se pasa a la variable *fecSchemeArg*. Se establece el mecanismo XOR y se fija el parámetro de redundancia mediante las siguientes variables:

- **NUMBER_OF_SOURCE_SYMBOLS**: esta variable especifica la cantidad de símbolos se envían antes de mandar el paquete de redundancia. Si por ejemplo, se fijase a cuatro, se transmitirían cuatro símbolos de información antes de mandar el paquete FEC.
- **NUMBER_OF_REPAIR_SYMBOLS**: define la cantidad de paquetes FEC que se mandan. En este caso se fija a uno.
- **NUMBER_OF_INTERLEAVED_BLOCKS**: indica cuantos bloques protege el paquete FEC.

A continuacón, se especifíca en la variable *fs* la técnica FEC utilizada definida en el código de QUIC (XOR): *fs=quic.XORFECScheme*. Por último, se pasa a la configuración de QUIC (*config*) el esquema escogido: *config :=* & *quic.Config{FECScheme: fs}*.

3.4. Aplicación *bulk*

Las medidas que se utilizan para la caracterización de resultados se obtienen a partir de una aplicación *bulk*: consiste en la transmisión masiva de información entre un servidor y un cliente. Los ficheros son *bulk_server.go* y *sink_client.go* como servidor y cliente respectivamente, y están programados en GO. El funcionamiento de esta aplicación consiste en que el cliente recibe toda la información que le transmite el servidor y ambos calculan el rendimiento o *throughput* de la red.

[23] bitbucket.org/michelfra/quic-fec/src/networking_2019/

$$\text{Throughput (Mbps)} = \frac{\text{Información útil (bits)}}{\text{Tiempo (ms)}} \tag{3.2}$$

La aplicación servidor básica consiste en transmitir información masiva cuando el cliente se conecta a él. La información se transmite en paquetes con tamaño máximo de 1452 bytes. Para la configuración sencilla de la comunicación se hace uso de las siguientes *flags*, se definen los siguientes parámetros:

- **ip**: se especifica la dirección IP del contenedor donde se ubica el servidor y el puerto donde se mantiene escuchando la petición del cliente.
- **mb**: cantidad de información a transmitir en megabytes.

Inicialmente, el servidor genera paquetes con información aleatoria y se queda a la espera de que el cliente le mande una petición. A continuación, se puede elegir entre iniciar una conexión QUIC o una conexión TCP. Para la creación de la primera, se establece una configuración TLS y para la segunda una configuración TLS+TCP. Por último, se calcula el *throughput* aplicando la Ecuación 3.2: se utiliza la cantidad de bytes que se han transmitido al cliente, dividido entre el tiempo en milisegundos que ha durado la descarga de datos.

Al otro lado de la comunicación se encuentra el cliente, cuyo funcionamiento consiste en conectarse al servidor y recibir la información emitida por este. La programación del cliente es similar a la del servidor, debido a que se pueden configurar algunos parámetros iniciales a través de *flags*:

- **ip**: se fija a la dirección IP y puerto donde escucha el servidor para la conexión con el mismo.
- **tcp**: mediante esta opción se elige el tipo de conexión, TCP o QUIC.

Una vez escogido el tipo de conexión que se quiere (TCP o QUIC), se definen algunos parámetros y se empieza a recibir información. Por último, igual que en el servidor, se calcula la capacidad neta de transferencia de información sobre un enlace, *throughput*.

Cuando se implementa FEC en QUIC en [3], es importante resaltar los cambios que sufren ambos lados de la red. Aparecen otros parámetros de configuración mediante *flags*:

- **fecRatio**: define la cantidad de paquetes (símbolos) que se transmiten antes de mandar un paquete FEC.

- **N**: el periodo de medida (T) en el cual se calcula el valor de las pérdidas residuales, T=N·RTT.
- **delta**(δ): el valor de la tasa FEC dinámica varía en función de este parámetro.
- **target**(γ): valor umbral para el cálculo de la tasa de transmisión dinámica de paquetes FEC.

Por otro lado, es importante implementar el codificador y decodificador FEC en los extremos de la comunicación. En el servidor, se implementa el codificador y el decodificador, donde se especifican los parámetros de ambos, y se pasa la configuración de estos a la configuración de QUIC. En la parte del cliente, se realiza el mismo proceso pero solo implementando el decodificador.

4

Simulaciones y resultados

En este capítulo se detalla el proceso de medida para la obtención de resultados, así como la explicación de estos. Se explican los parámetros que se han modificado para las medidas y sus aspectos más significativos.

Los resultados representados se basan en el protocolo QUIC nativo y en la implementación de técnicas FEC en QUIC de [3] y de [4] con las modificaciones pertinentes explicadas en el Capítulo 3 (Sección 3.3).

4.1. Conceptos previos

El proceso de medida se realiza sobre la red representada en la Figura 3.1. Consta de dos contenedores que ejecutan la aplicación *bulk* sobre el protocolo de transporte QUIC. Se comunican a través de la topología simulada mediante *ns-3*, basada en una red CSMA para la conexión de los contenedores a los respectivos routers y en una red P2P para la conectividad de estos. La red P2P permite modificar los parámetros que caracterizan los resultados: ancho de banda, retardo, probabilidad de pérdida de paquetes y BDP:

- **Ancho de banda (BW)**: capacidad máxima de la red desplegada.
- **Retardo de ida y vuelta (RTT)**: tiempo requerido para que un paquete se transmita hasta el receptor y vuelva al transmisor.

- **Probabilidad de pérdida de paquetes (*loss rate*)**: es la tasa de pérdida de paquetes en el canal.
- **BDP**: como se ha explicado anteriormente, el *bandwidth delay product* es la cantidad máxima de información que está en la red en un tiempo dado. Se modifica este valor para que la capacidad de la red no sea infinita.

El hecho de poder modificar ciertos parámetros hace que se puedan simular o representar diferentes tipos de redes, por esto, los valores de ancho de banda y RTT que se programan son los que se representan en la Tabla 4.1. Por otro lado, la probabilidad de pérdida de paquetes toma los siguientes valores: 0 %, 1 %, 2 %, 3 % y 5 %.

Tabla 4.1: Caracterización de la red.

Tipo de red	**BW (Mbps)**	**RTT (ms)**	**BDP**
WiFi/LTE	20	25	45
2G/3G	10	100	85
Enlaces satelitales	1.5	400	55

Los resultados que se obtienen se basan en la utilización de QUIC nativo y la inclusión de técnicas FEC en los estudios de rQUIC [3] y QUIC-FEC [4]. Se hará una comparación entre la versión original de QUIC y rQUIC dinámico, así como la comparativa de rQUIC y QUIC-FEC con tasa de redundancia fija. El barrido de la tasa FEC (*FEC ratio*) adquiere los siguientes valores: 0, 5, 10, 15, 20, 25, 30 y 32. El FEC ratio se debe fijar a un valor que respete la ventana congestión prefijada en el código base de QUIC. En este caso se deja a 32, de ahí que la tasa FEC adquiera como mucho ese valor.

Los parámetros que se han analizado son el $\overline{\textbf{\textit{throughput}}}$ y el **tiempo de descarga de un fichero**. El $\overline{\textit{throughput}}$ o rendimiento medio de la red es la media de la cantidad efectiva de información que se transmite por la red. Se calcula a partir de la información recibida por el cliente y el tiempo que tarda en llegar. El tiempo de descarga se define como el tiempo necesario para la descarga de un fichero desde el servidor. Es importante recalcar que el tiempo de descarga se representará como la ganancia de un método sobre QUIC nativo (*Download Completion Time Ratio*, DCTR), es decir, si el valor de DCTR es menor que 1, significa que la alternativa a QUIC nativo presenta mejor comportamiento que este.

La cantidad de información que se ha transmitido regularmente entre servidor y cliente es de 20 MB, y los datos se han procesado con Matlab.

4.2. Análisis de resultados

A continuación se explicarán los resultados obtenidos con QUIC y sus alternativas, QUIC-FEC y rQUIC.

4.2.1. rQUIC tasa fija

En una primera campaña de resultados, se analiza el comportamiento que se obtiene de rQUIC manteniendo la tasa FEC fija. En la Figura 4.1 se representa el parámetro DCTR frente a la tasa FEC fijada barriendo la probabilidad de error. Los parámetros de ancho de banda y RTT que definen la red son 20 Mbps y 25 ms respectivamente.

Se puede comprobar como para un valor de FEC igual a 0, rQUIC y QUIC se comporta de una forma muy similar, debido a que no se inyecta ningún paquete FEC. Con valores de FEC superiores se representa una notable mejora en cuanto al tiempo de descarga de información, debido a que el DCTR se sitúa por debajo de 1 en la mayoría de los casos. Se ve una pequeña ventaja cuando el FEC es 5, es decir, cada 5 paquetes de información, se manda uno de redundancia. La inyección de paquetes de redundancia es favorable respecto a la recuperación de información, pero cuanto menor sea la tasa FEC fijada, mayor será la sobrecarga en la red. Por otro lado, el resultado para una probabilidad de pérdida del 0 % es el esperado, debido a que no se sufre la necesidad de recuperar información, siendo una respuesta relativamente constante para cualquier valor de tasa FEC.

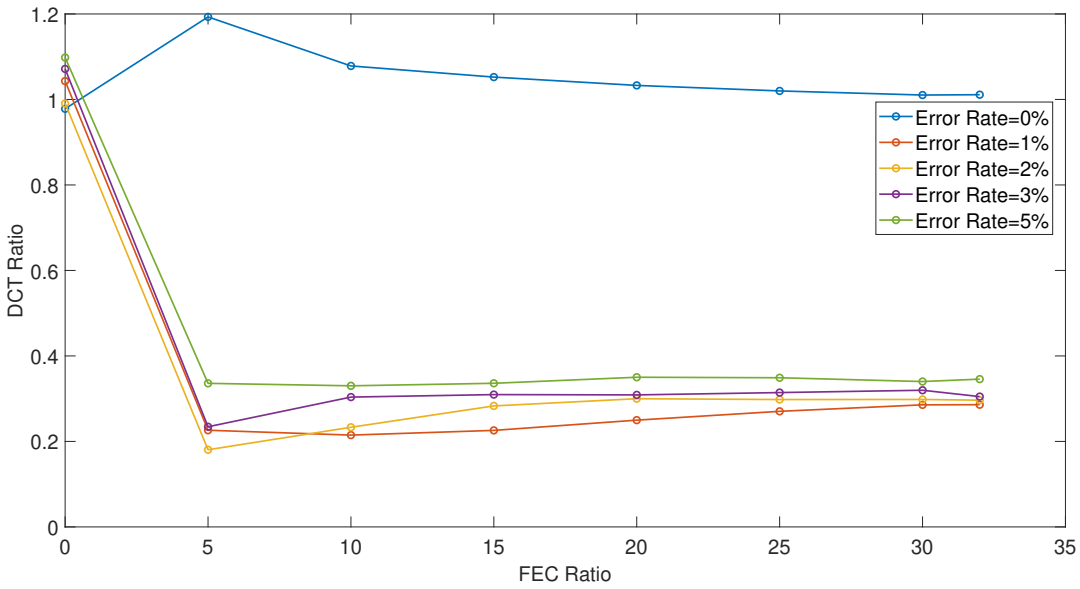


Figura 4.1: DCTR de rQUIC con tasa FEC fija vs QUIC.

La modificación de la tasa FEC también influye directamente en el $\overline{throughput}$ de la red. Con una probabilidad del 0 %, la utilización de redundancia con cualquier valor de tasa fija aporta un $\overline{throughput}$ medio de 17 Mbps. Por otro lado, con la implementación de QUIC nativo se obtiene los valores de rendimiento de la Tabla 4.2.

Tabla 4.2: $\overline{Throughput}$ utilizando QUIC nativo.

Prob. de pérdida	0 %	1 %	2 %	3 %	5 %
Thput de QUIC (Mbps)	18.2	3.4	2.1	1.6	1.2

En la Figura 4.2 se muestra el rendimiento medio de la red en función de la probabilidad de pérdida de paquetes y la tasa de redundancia. No se representa el valor que se obtiene con un 0 % de pérdida debido a que el uso de rQUIC fijo no presenta ninguna ventaja respecto a QUIC nativo. En comparación con la Tabla 4.2 se puede ver que para una tasa de pérdida y un FEC distinto a 0, el comportamiento de la red en cuanto el rendimiento, mejora notablemente. Se puede verificar en comparación con el FEC fijado a 0, ya que imita el comportamiento del protocolo puro.

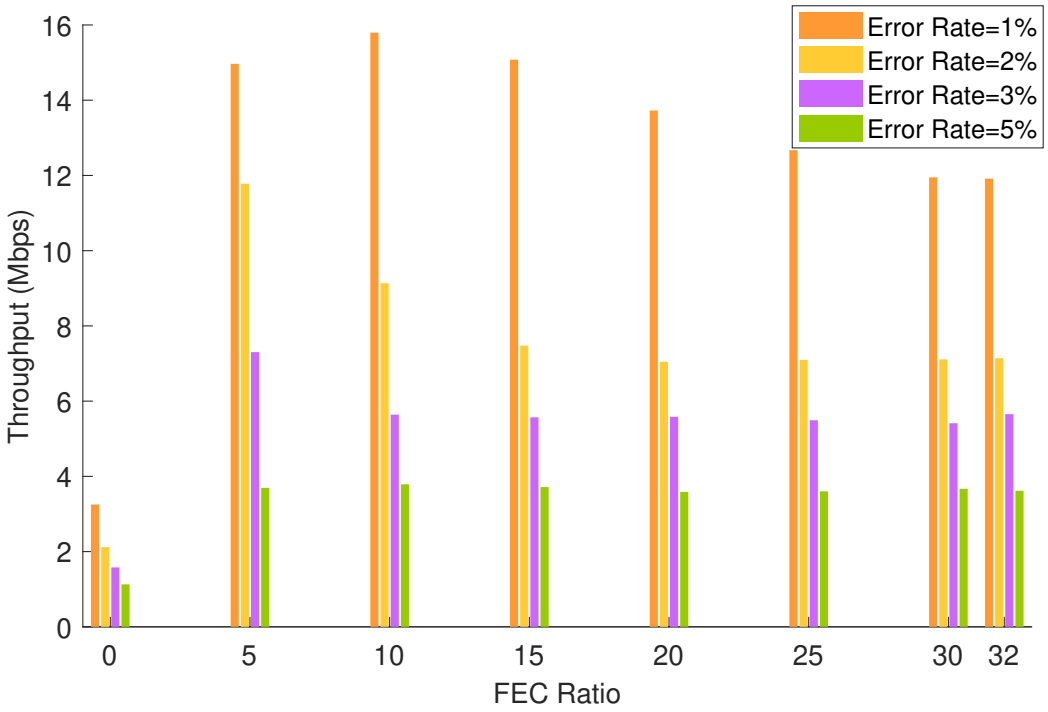


Figura 4.2: Representación del *throughput* obtenido con rQUIC con tasa FEC fija

4.2.2. QUIC-FEC

Las características de la red para realizar las medidas con esta alternativa es: ancho de banda de 20 Mbps y RTT 25 ms. Mismas condiciones que en rQUIC fijo. Cabe destacar que en [4] realizan sus experimentos con transferencias *bulk* de 1 MB como máximo, en cambio en este trabajo se evalúa con transferencias de 20 MB.

En una primera toma de contacto con esta alternativa, se compara QUIC-FEC con QUIC nativo. En la Figura 4.3 se comprueba como la utilización de técnicas de corrección de errores para la recuperación de información aporta un mejor comportamiento que utilizando QUIC en su estado natural: valor de DCTR está por debajo de 1.

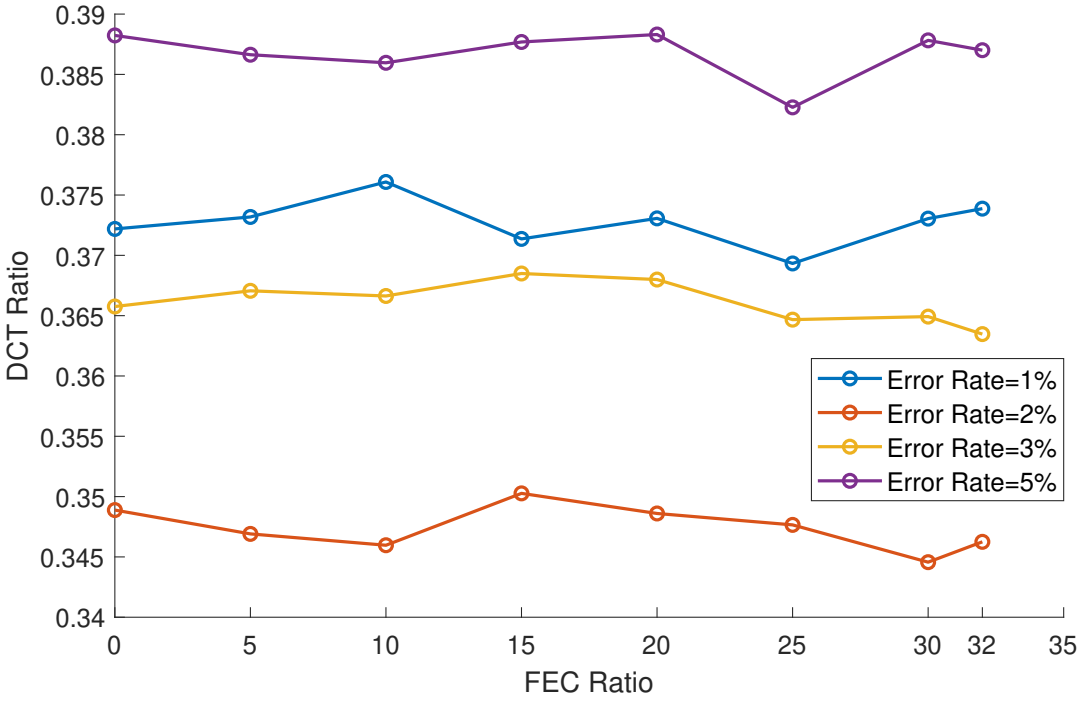


Figura 4.3: DCTR de QUIC-FEC vs QUIC.

Como se han estudiado las dos alternativas donde se incluye *Forward Error Correction*, es interesante realizar una comparativa entre ellas. En la Figura 4.4 se puede ver como la alternativa rQUIC basada en [3] ofrece un tiempo de descarga menor que QUIC-FEC (DCTR<1). Para una probabilidad de pérdida del 0 % se ofrece una mejor respuesta a medida que disminuye la tasa FEC debido a que no se inyectan tantos paquetes de redundancia. Para otros valores de tasa de pérdida de paquetes, ofrece más beneficio utilizar un ratio de FEC más pequeño, que aunque se sobrecargue mucho más la red, la recuperación de paquetes perdidos es mucho más rápida. Por otro lado, para una tasa FEC de 10 y un *error rate* de 1 %, se observa el valor más pequeño de DCTR, lo que se puede deducir

que existe un compromiso entre la rapidez de recuperación de paquetes (latencia) y la sobrecarga de la red.

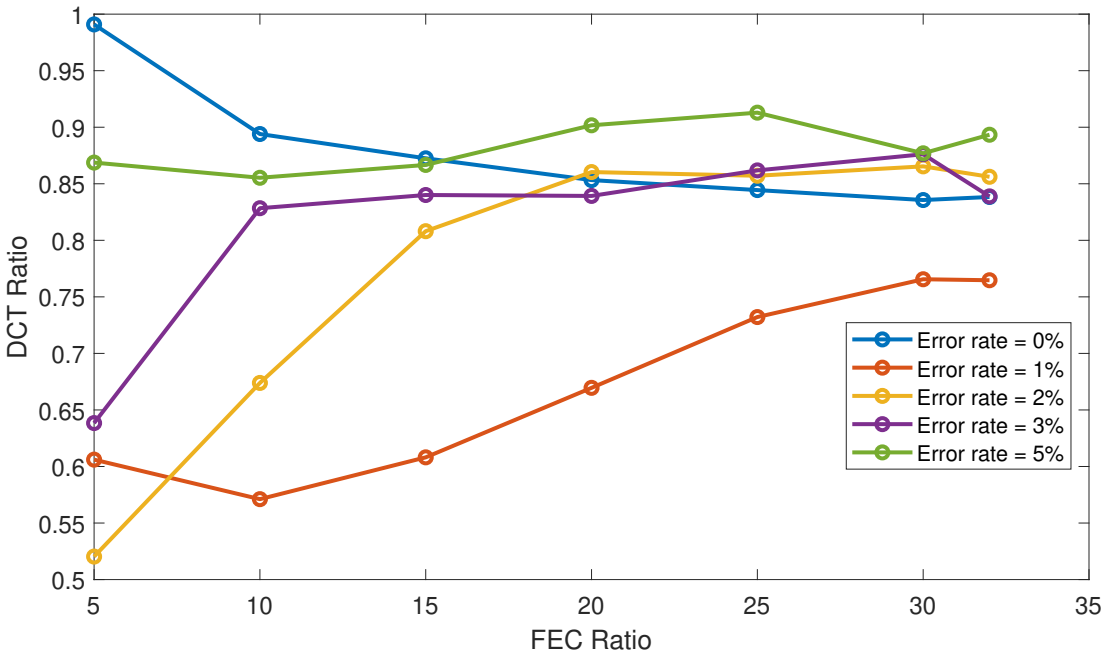


Figura 4.4: DCTR de rQUIC FEC fijo vs QUIC-FEC.

4.2.3. rQUIC

Por último, se verifica el beneficio que aporta [13] sobre QUIC nativo. rQUIC lo que busca es adaptar la tasa FEC en función del estado de la red, es decir, se adapta el ratio según la necesidad de retransmisión de la red. Por otro lado, el algoritmo FEC es el que se basa en una operación XOR, ideal para reducir el tiempo de codificación y decodificación y, por lo tanto, reducir la latencia en la red.

En la Figura 4.5 se comprueba como para las mismas condiciones del canal que se han implementando hasta ahora, emulando un sistema Wi-Fi o LTE (ancho de banda 20Mbps y RTT 25 ms), la utilización de rQUIC aporta una ganancia significativa en cuanto al tiempo de descarga sobre QUIC. Se puede ver que para tasas de error por encima de 0 %, ajustar la tasa FEC tiene mejor comportamiento que la implementación de QUIC nativo. Para una probabilidad de pérdida del 0 % se ve como rQUIC se comporta igual que QUIC debido a que no se aplica el algoritmo adaptativo, cancelándose así la tasa FEC. Por otro lado, también se ve un beneficio directo en el rendimiento de la red. En la Tabla 4.3 se recogen los valores de $\overline{throughput}$, donde se presentan en algunos casos casi cuatro veces más que el rendimiento de QUIC.

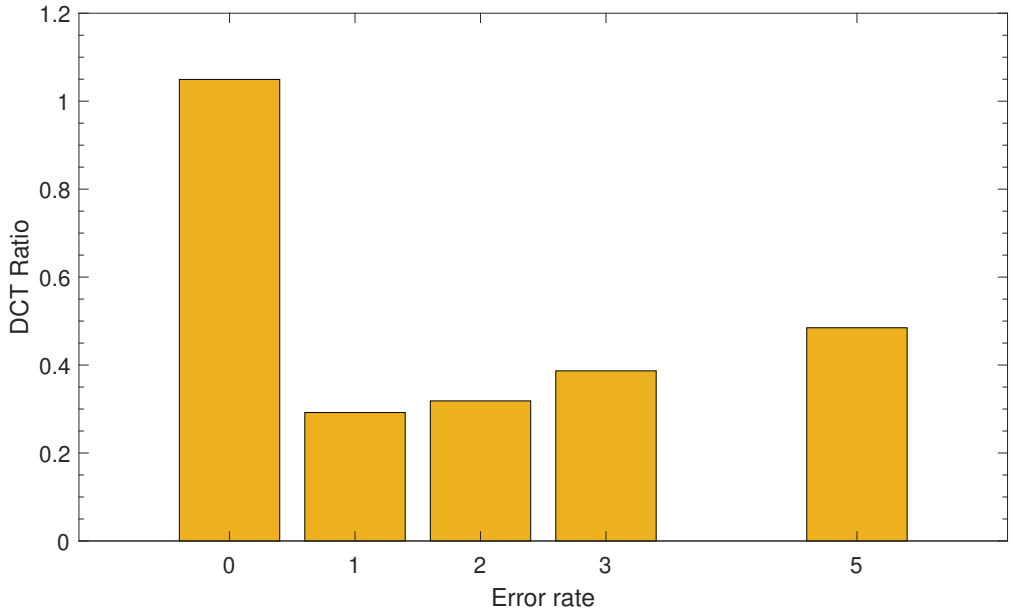


Figura 4.5: DCTR de rQUIC vs QUIC.

Tabla 4.3: $\overline{Throughput}$ utilizando rQUIC y QUIC nativo para BW=20Mbps y RTT=25ms.

Prob. de pérdida	0 %	1 %	2 %	3 %	5 %
Thput de QUIC (Mbps)	18.1	3.4	2.1	1.6	1.2
Thput de rQUIC (Mbps)	17.3	11.6	6.6	4.4	2.6

Finalmente, para un análisis completo de esta alternativa a QUIC, se modifica los parámetros de la red de ancho de banda y RTT para emular redes 2G-3G y satelitales según la Tabla 4.1, Figura 4.6 y Figura 4.7 respectivamente. Para redes 2G-3G la respuesta de rQUIC sigue siendo beneficiosa aunque no se obtiene tanta ventaja debido a que a medida que se aumenta la tasa de error en el enlace, la sobrecarga de la red es mayor, tal y como se puede ver en los enlaces satelitales. Para RTT altos, el algoritmo adaptativo de rQUIC no presenta un comportamiento notable respecto al resultado que se obtiene con un RTT de 25ms. Por otro lado, tal y como se ha argumentado anteriormente, para tasas de error del 0 % rQUIC no ofrece ninguna ventaja respecto a QUIC.

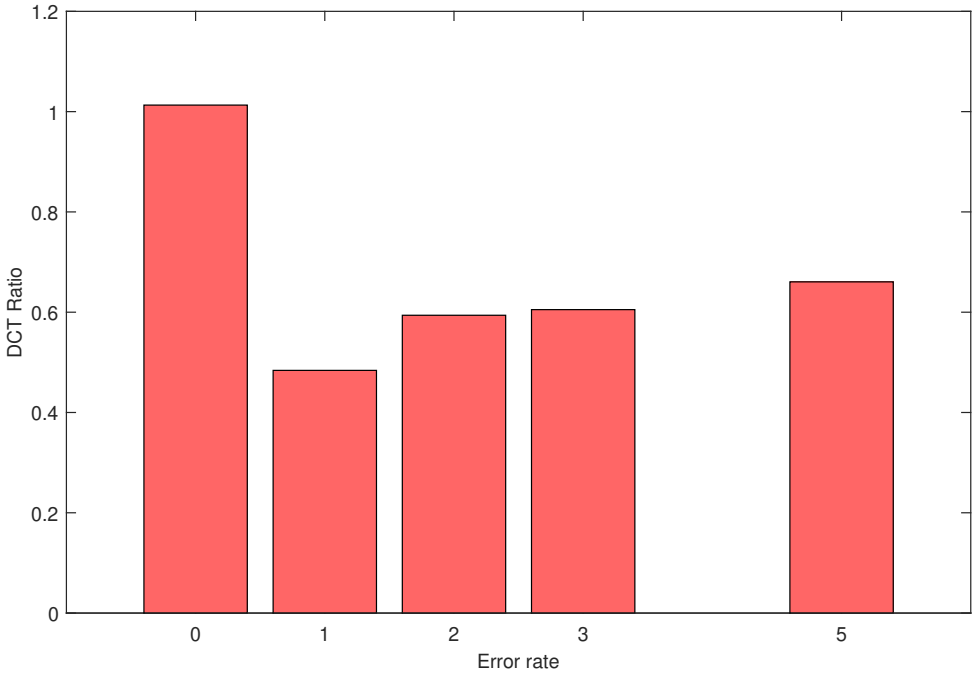


Figura 4.6: DCTR de rQUIC vs QUIC para BW=10Mbps y RTT 100ms.

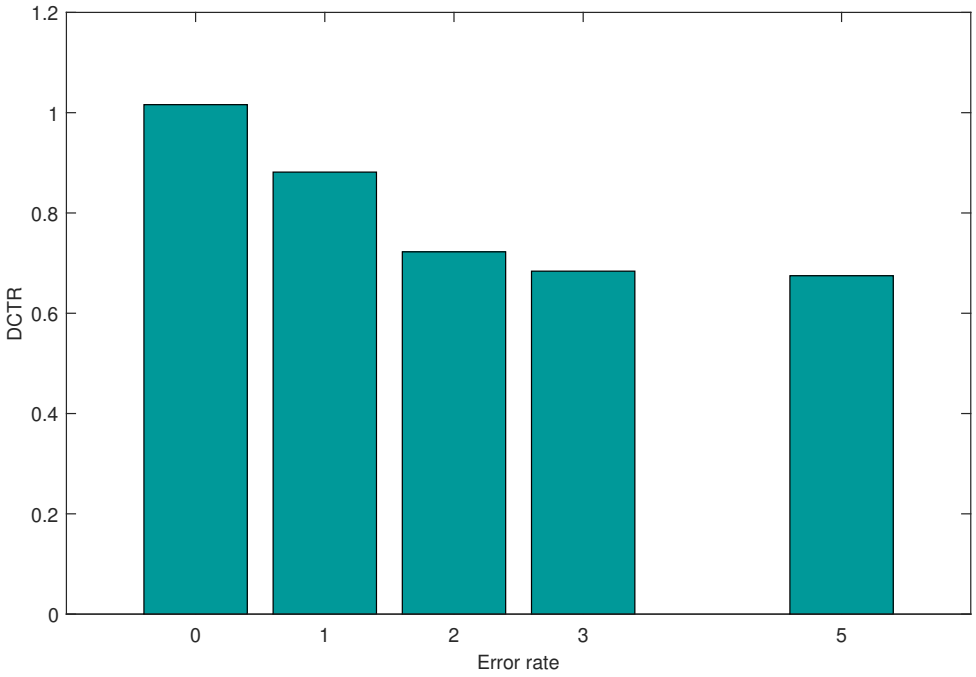


Figura 4.7: DCTR de rQUIC vs QUIC para BW=1.5Mbps y RTT 400ms.

4.2.4. Protocolo de congestión

En este trabajo se ha iniciado el cambio en el protocolo de congestión que utiliza QUIC nativo de forma transparente. Como se ha mencionado anteriormente, QUIC implementa TCP CUBIC como mecanismo de congestión y sería interesante estudiar el comportamiento de QUIC con otros protocolos como es TCP New Reno.

La implementación de QUIC que se ha utilizado en este proyecto es la que se realiza en el repositorio **github.com/lucas-clemente/quic-go/tree/v0.7.0**. La implementación del mecanismo de congestión se ubica en la carpeta *internal/congestion*. En el programa *cubic.go* se describe el funcionamiento CUBIC, y en *cubic_sender.go* se programa el transmisor (función *NewCubicSender*), y se hace la estimación de un parámetro importante para calcular la ventana de congestión de TCP Reno. Luego, en el directorio *internal/ackhandler* se programa *sent_packet_handler.go*, donde se elige el mecanismo de congestión. Esto se hace a través de la función *NewSentPacketHandler* donde se llama al transmisor del protocolo de interés:

```
congestion:=congestion.NewCubicSender(...)
```

Para implementar TCP New Reno se utiliza el código aportado en el repositorio **github.com/google/gvisor/blob/master/pkg/tcpip/transport/tcp/reno.go**. *reno.go* especifica el comportamiento de TCP New Reno. Luego, en el código *snd.go* se programa el transmisor "general" y mediante la *estructura sender* se definen parámetros importantes como es la ventana de congestión o el mecanismo de congestión que se pretende utilizar, *sndCwnd* y *cc congestionControl* respectivamente. En la función *initCongestionControl* se inicializa el protocolo de congestión deseado, así como la ventana de congestión y el umbral de la fase de *slow start*. Además, se define el sender con el mecanismo de congestión requerido.

Para que la integración de TCP New Reno sea lo más transparente posible, se identifica en el código *reno.go* y *snd.go* las interfaces (funciones y como se invocan) y se busca una correspondencia con la implementación de TCP CUBIC.

5

Conclusiones y líneas futuras

En este último capítulo se recogen las principales conclusiones que se han ido recopilando durante la realización del trabajo, haciendo especial hincapié en aquellas obtenidas tras analizar los resultados presentados en el Capítulo 4. Finalmente, se expondrán posibles líneas futuras de investigación que han quedado abiertas a raíz del estudio que se ha realizado.

5.1. Conclusiones

Como consecuencia del cambio tan sustancial que se ha producido en el mundo de las comunicaciones e Internet, la comunidad investigadora ha desarrollado propuestas con las que abordar la reducción de la latencia en la red, especialmente en las comunicaciones inalámbricas. Una importante limitación de las redes que hace que el tiempo de ida y vuelta no sea mínimo, son los procesos de en la recuperación de paquetes. El protocolo de transporte QUIC se desarrolla para que la latencia en las redes se vea reducida, y en este trabajo se recoge un estudio de las distintas alternativas que han aparecido, introduciendo técnicas de corrección de errores: rQUIC [3] y QUIC-FEC [4].

Una vez realizado el estudio del funcionamiento de QUIC, rQUIC y QUIC-FEC, se analiza su comportamiento mediante la ejecución de la aplicación *bulk* entre cliente y servidor.

Inicialmente, se compara rQUIC modificando su funcionamiento y dejando estático la tasa de transmisión de redundancia o tasa FEC, con QUIC. Se verifica que el hecho de implementar *Forward Error Correction* en QUIC aporta una reducción considerable de la latencia, así como un rendimiento mayor de la red. Es importante tener en cuenta la posible sobrecarga que se inyecta en la red, debido a la transmisión de redundancia, por eso, debe existir un compromiso entre la reducción de latencia y la frecuencia en la que se transmiten paquetes FEC, ya que también perjudicaría en el rendimiento.

Por otro lado, en QUIC-FEC se definen tres esquemas distintos de corrección de errores: XOR, *Reed-Solomon* y el código convolucional RLC. En este trabajo se utilizará el esquema XOR, debido a que rQUIC se basa en este. La comparativa de QUIC-FEC y QUIC corrobora que las técnicas de corrección de errores presenta un claro beneficio, ya que el tiempo de descarga disminuye notablemente. Cuando se compara rQUIC con el FEC estático y QUIC-FEC, se puede ver como el primero aporta mejores resultados. El tiempo de descarga, DCTR, es mayor en QUIC-FEC. Esto puede suceder debido a la manipulación que se hace en cuanto al control de flujo y la congestión: QUIC-FEC trabaja por encima del control de congestión y rQUIC con el FEC fijo lo considera, ya que de esta forma se evita que la red se congestione en medio de la transmisión de un bloque FEC.

La utilización de rQUIC con la tasa FEC dinámica hace que la transmisión de redundancia se adapte a las condiciones de la red. Esto produce que la sobrecarga se vea reducida en cuanto a las alternativas anteriores, ya que si no hay pérdida de paquetes la transmisión de redundancia se ve cancelada. rQUIC también aporta una ventaja sobre el tiempo de descarga y el *throughput*, independientemente al tipo de red.

Por último, debido a que el objetivo principal de QUIC es reducir la latencia en las comunicaciones, sería interesante implementar el mecanismo de congestión que aporta BBR. El protocolo BBR es una gran alternativa porque su principal objetivo es no entender la pérdida de paquetes como congestión, ya que la primera se produce antes que la segunda. De esta manera, el uso del mecanismo de congestión es menos agresivo porque no se basa en la adaptación de la ventana de congestión, sino en el análisis de la capacidad en cada instante y, por lo tanto, en el RTT de cada momento.

En conclusión, para una red de comunicaciones actual, donde el principal objetivo es reducir la latencia y aumentar el ancho de banda, el uso de rQUIC parece una gran alternativa: utilizar técnicas de corrección de errores hace que se vean suprimidas retransmisiones innecesarias, reduciendo también la sobrecarga de la red y aumentando el rendimiento. Por otro lado, la implementación que se realiza del FEC sobre QUIC es totalmente transparente, por lo tanto, su inclusión en sistemas reales no sería muy costosa. Por último, puede surgir un gran interés en modificar el protocolo de congestión que utiliza por defecto QUIC por BBR en redes que presentan mucho tráfico.

5.2. Líneas futuras

Además de las aportaciones que se exponen en este trabajo, es importante seguir en esta línea de investigación para la mejora de las prestaciones de las comunicaciones. A continuación, se detallan las distintas líneas futuras que se han concluido:

- Debido a que la utilización de técnicas de corrección de errores, como es *Forward Error Correction*, aporta una mejora notable en cuanto a latencia, sería interesante implementar el esquema de codificación *Network Coding*(NC)[13]. En las redes *multicast* se utiliza esta técnica de codificación para no realizar retransmisiones selectivas para la recuperación de paquetes. Implementar NC sobre QUIC podría dar un servicio aún más fiable y escalable en redes inalámbricas debido a que se puede producir menos sobrecarga en la red.

- Por otro lado, es interesante realizar un estudio del protocolo QUIC sobre redes *multipath*[14]. Este tipo de redes son útiles para redes inalámbricas debido a que permiten el uso de varios caminos para la retransmisión de información entre nodos. Evaluar el comportamiento de QUIC sobre estas redes podría verificar el buen comportamiento que tiene este protocolo en cuanto al retardo que aparece en aplicaciones cliente-servidor. Además, el hecho de que QUIC multiplexe *streams*, podría ser un beneficio adicional en este tipo de redes.

- Por último, el protocolo de congestión que utiliza QUIC por defecto es TCP CUBIC. Después de la investigación que se ha hecho en la posibilidad de implementar TCP New Reno para el análisis de su comportamiento, sería interesante implementar *Bottleneck Bandwidth and Round-trip time* (BBR). Como se ha explicado en el Capítulo 2, BBR aporta un mejor control de congestión, ideal para redes inalámbricas.

Bibliografía

[1] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A first look at QUIC in the wild. In *Passive and Active Measurement*, pages 255–268. Springer International Publishing, 2018.

[2] Adam Langley, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Alistair Riddoch, Wan-Teh Chang, Zhongyi Shi, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, and Ian Swett. The QUIC transport protocol. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*. ACM Press, 2017.

[3] Pablo Garrido, Isabel Sánchez, Simone Ferlin, Ramón Agüero, and Ozgü Alay. rquic: Integrating fec with quic for robust wireless communications.

[4] François Michel, Quentin De Coninck, and Olivier Bonaventure. Quic-fec: Bringing the benefits of forward erasure correction to quic. *arXiv preprint arXiv:1904.11326*, 2019.

[5] Jim Roskind. Quic: Design document and specification rational. *Tech. Rep.*, 2015.

[6] Ryan Hamilton, Janardhan Iyengar, Ian Swett, Alyssa Wilk, et al. Quic: A udp-based secure and reliable transport for http/2. *IETF, draft-tsvwg-quic-protocol-02*, 2016.

[7] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. Host-to-host congestion control for TCP. *IEEE Communications Surveys & Tutorials*, 12(3):304–342, 2010.

[8] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. Technical report, apr 1999.

[9] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, jul 2008.

[10] Mario Hock, Roland Bless, and Martina Zitterbart. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, oct 2017.

[11] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. Towards a deeper understanding of TCP BBR congestion control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, may 2018.

[12] John P. Rula, James Newman, Fabián E. Bustamante, Arash Molavi Kakhki, and David Choffnes. Mile high WiFi. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*. ACM Press, 2018.

[13] Pablo Garrido and Ramon Agüero. Caracterización experimental del comportamiento de network coding para comunicaciones multicast. In *Proceedings XIII Jornadas de Ingenieria Telematica - JITEL2017*. Universitat Politècnica València, sep 2017.

[14] Quentin De Coninck and Olivier Bonaventure. Multipath QUIC. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*. ACM Press, 2017.