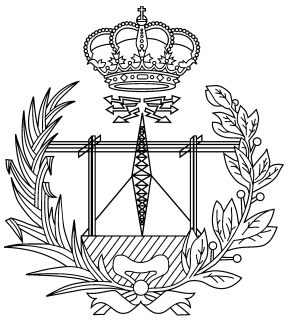
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Grado

COMPORTAMIENTO DE PROTOCOLOS DE TRANSPORTE EN ENTORNOS IIOT

(On the behavior of transport protocols over IIoT environments)

Para acceder al Título de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Neco Villegas Saiz

Septiembre- 2021

UNIVERSIDAD DE CANTABRIA

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

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Neco Villegas Saiz
Directores del TFG: Luis Francisco Diez Fernández, Ramón Agüero Calvo
Título: "COMPORTAMIENTO DE PROTOCOLOS DE TRANSPORTE EN ENTORNOS IIOT"
Title: "On the behavior of transport protocols over IIoT environments"

Presentado a examen el día:

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del tribunal:
Presidente (Apellidos, Nombre): García Arranz, Marta
Secretario (Apellidos, Nombre): Diez Fernández, Luis Francisco
Vocal (Apellidos, Nombre): Conde Portilla, Olga

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

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFG (solo si es distinto del Secretario)
V^{o} B^{o} del Subdirector	Trabajo Fin de Grado N^{o} (a asignar por Secretaría)

Índice general

Índice de figuras

Índice de tablas

Acrónimos

AEAD Authenticated Encryption with Associated Data.

AIMD Additive Increase/Multiplicative Decrease.

BBR Bottleneck Bandwidth and Round-trip time.

BDP Bandwidth-Delay Product.

BIC Binary Increase Congestion control.

ECN Explicit Congestion Notification.

HOL blocking Head-of-line blocking.

HTTP Hypertext Transfer Protocol.

IIoT Industrial Internet of Things.

M2M Machine To Machine.

MQTT Message Queue Telemetry Transport.

MSS Maximum Segment Size.

QoS Quality of Service.

QUIC Quick UDP Internet Connections.

RTO Retransmission Time-Out.

RTT Round-Trip Time.

SPDY SPeeDy.

SSL Secure Sockets Layer.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

Resumen Ejecutivo

QUIC es un protocolo de transporte encriptado, multiplexado y de baja latencia promovido por Google y diseñado desde cero para abordar las limitaciones del protocolo TCP, que es la solución que se ha venido utilizando de manera más habitual. Sin embargo, QUIC ha sido diseñado principalmente para servicios web, por lo que su uso en entornos IIoT no ha sido suficientemente analizado, resultando particularmente interesante caracterizar su rendimiento cuando se utiliza en estos escenarios.

En este proyecto se analiza el desempeño de QUIC como alternativa de transporte para IIoT, utilizando el protocolo MQTT, así como el papel de los algoritmos de control de congestión, que implementan un conjunto de técnicas para evitar situaciones de saturación en la red y recuperar datos en caso de pérdida. En ese sentido, Google propuso recientemente BBR para reemplazar los algoritmos tradicionales de control de congestión basados en pérdidas, como NewReno o CUBIC, con el objetivo de incrementar el rendimiento y reducir la latencia. Sin embargo, muchos estudios han puesto de manifiesto problemas de rendimiento de BBR, como la poco equidad en el reparto de capacidad entre flujos con diferentes RTT, o al compartir el canal con algoritmos basados en pérdidas como CUBIC.

En la memoria se realiza, en primer lugar, una revisión del estado del arte, en la que se analiza el funcionamiento de los protocolos implicados. A continuación se ha empleado el simulador *ns-3* para comparar el desempeño de QUIC con el observado al hacer uso del esquema tradicional TCP/TLS, así como las diferencias entre los algoritmos de control de congestión. Se han simulado diferentes tecnologías de red, y los resultados obtenidos ponen de manifiesto que QUIC ofrece un buen rendimiento, lo que permitiría su adopción en escenarios IIoT.

Executive Abstract

QUIC is an encrypted, multiplexed, and low-latency transport protocol promoted by Google and designed from the ground up to tackle the limitations of the traditional TCP. However, QUIC use in IIoT environments is not widespread, and it is therefore interesting to characterize its performance when in over such scenarios.

In this project, the performance of QUIC as a transport alternative for IIoT based on MQTT is analyzed, as well as the role of congestion control algorithms, which propose a set of techniques to avoid saturation situations in the network and recover from data losses. In that sense, Google recently proposed BBR to replace traditional loss-based congestion control algorithms, such as NewReno or CUBIC, to achieve high performance and low latency. However, many studies have reported performance issues in BBR operation, such as unfairness between flows with different RTTs or sharing the channel with loss-based algorithms.

In order to introduce the information in a clear way, an analysis of the state of the art is first carried out in which the operation of the protocols involved is briefly analyzed and explained. The *ns-3* simulator is then used to compare the performance of QUIC with that exhibited by the traditional TCP/TLS scheme, as well as the differences between the congestion control algorithms. Different network technologies are simulated, evincing that QUIC yields good performances in configurations resembling IIoT scenarios.

1 Introducción

1.1. Objetivos

En este trabajo se pretende analizar el comportamiento de diferentes protocolos de transporte y soluciones de control de congestión en entornos Industrial Internet of Things (IIoT). Para ello se hará uso del simulador *ns-3* y de una implementación del protocolo Quick UDP Internet Connections (QUIC). Se comparará el comportamiento de esta solución con el del protocolo de transporte Transmission Control Protocol (TCP), utilizando diferentes mecanismos de control de congestión: NewReno, CUBIC, Bottleneck Bandwidth and Round-trip time (BBR).

Se hará uso del protocolo de aplicación Message Queue Telemetry Transport (MQTT), cuyo uso está muy extendido en los escenarios IIoT. Como no se dispone de una implementación del mismo en el marco del simulador *ns-3*, un primer objetivo del TFG será la implementación de una aplicación que emule el comportamiento de MQTT, con la presencia de los tres roles que contempla: publisher, subscriber y broker.

Una vez establecido el entorno de análisis, se llevará a cabo una campaña de experimentos para estudiar el comportamiento de las diferentes combinaciones protocolo de transporte/mecanismo de control de congestión, bajo diferentes supuestos, teniendo en cuenta las características de las tecnologías subyacentes y los patrones de tráfico.

1.2. Estructura

En esta sección se explican brevemente los principales puntos tratados en los capítulos posteriores.

Capítulo 2. Antecedentes

Se realiza una revisión teórica de todos los aspectos que se han contemplado durante el proyecto. Primero, se realiza un introducción al protocolo MQTT y al modelo publish-subscribe y, a continuación, se explica el protocolo de transporte QUIC, centrándose en las ventajas que puede aportar en entornos IIoT. Además, se describen los algoritmos de control de congestión empleados en las simulaciones. Estos son:

- **NewReno.** Una de las principales alternativas empleadas por TCP y en la que se basan la mayoría de los demás algoritmos de control de congestión.

- **CUBIC.** Una solución desarrollada para mejorar las prestaciones de TCP en redes actuales, las cuales disponen cada vez de mayor ancho de banda.
- **BBR.** Un nuevo enfoque que rompe con el control de la congestión basado en la pérdida de paquetes.

Por último, se comentan diferentes aspectos relativos al simulador *ns-3* y que sirven de introducción al siguiente capítulo.

Capítulo 3. Entorno de simulación

Se describe el entorno de simulación en *ns-3*, incluyendo las aplicaciones desarrolladas, los diferentes escenarios y las soluciones adoptadas para lograr los objetivos propuestos en el proyecto.

Capítulo 4. Resultados

Se realiza un análisis de los resultados obtenidos en las simulaciones, justificando así lo descrito en capítulos anteriores. Se compara el comportamiento de TCP y QUIC en varios escenarios y se analiza el papel que desempeñan los algoritmos de control de congestión estudiados.

Capítulo 5. Conclusiones

Se resumen las conclusiones obtenidas, tanto de la parte teórica como de las simulaciones realizadas, así como las posibles líneas futuras que surgen al finalizar este proyecto.

2 Antecendentes

2.1. MQTT

MQTT es un protocolo de comunicación Machine To Machine (M2M) creado en 1999 y basado en el modelo publish-subscribe. Aunque inicialmente se trataba de un protocolo propietario, en 2010 fue liberado y, en 2014, la versión 3.11 fue estandarizada por la Organización Internacional de Normalización (ISO) y la Organización para el Avance de Estándares de Información Estructurada (OASIS). Actualmente MQTT se encuentra en su versión 5 [1].

A modo de ejemplo, en la Figura 2.1 se muestra el modelo de comunicaciones de MQTT aplicado a IIoT con sensores de temperatura. En MQTT existen tres tipos de dispositivos: publisher, subscriber y broker. Los publishers suelen ser pequeños sensores, que generan la información y la publican en un broker común en temas específicos. Por otro lado, un subscriber recibe todos los mensajes de un tema concreto, producidos por cualquiera de los publishers. Tanto los publishers como los subscribers pueden verse como clientes en la topología de red correspondiente. El elemento clave de MQTT es el servidor intermedio, conocido como broker, que es quien se encarga de gestionar las suscripciones. Todos los mensajes publicados en la red se envían al broker, que se encarga de distribuir la información a los subscribers correspondientes. Aunque los clientes solo interactúan con un broker, el sistema podría contemplar un cluster de brokers, que intercambian datos según los temas de sus subscribers actuales. El broker también considera los distintos niveles de Quality of Service (QoS) para los clientes y, por lo tanto, las retransmisiones potenciales. MQTT presenta tres niveles QoS:

- **QoS 0 unacknowledged (at most one).** El mensaje se envía una única vez.
- **QoS 1 acknowledged (at least one).** El mensaje se envía hasta que se garantiza la entrega. El suscriber puede recibir mensajes duplicados.
- **QoS 2 assured (exactly one).** Se garantiza que cada mensaje se entrega al suscriber únicamente una vez.

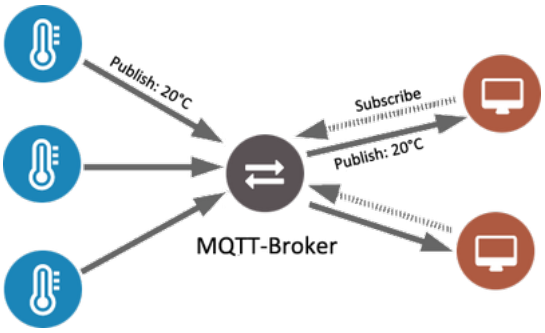


Figura 2.1: Modelo de comunicación IIoT MQTT.

MQTT dispone de distintas medidas de seguridad, incluyendo transporte Secure Sockets Layer (SSL)/Transport Layer Security (TLS), y autentificación por usuario y contraseña o mediante certificado. Uno de los aspectos fuertes de MQTT es el desacoplamiento entre los dispositivos, gracias al paradigma publish-subscribe que, junto a la sencillez y ligereza del protocolo, facilita la escalabilidad y la implementación en dispositivos de bajo cálculo. Otra ventaja es que el publisher puede enviar contenido nuevo siempre que esté disponible debido al asincronismo entre el interés de un nodo y la publicación de la información.

2.2. QUIC

QUIC [2] es un protocolo desarrollado por Google con el objetivo de proporcionar un transporte de extremo a extremo, seguro y confiable, de baja latencia. Google agregó soporte para Chrome en 2013 y, en 2017, todos los usuarios de la aplicación YouTube de Chrome y Android ya usaban este protocolo de transporte. La idea de diseño inicial de QUIC fue proporcionada por SPeeDy (SPDY), que luego se estandarizó como Hypertext Transfer Protocol (HTTP)/2 [3]. En la Figura 2.2 se muestra la integración de QUIC en la pila de protocolos TCP/IP.

QUIC proporciona multiplexación de streams sin los inconvenientes del Head-of-line blocking (HOL blocking) de TCP, mejora los tiempos de establecimiento de la conexión, al combinar la negociación de parámetros criptográficos y de transporte, y proporciona handshake de 1-Round-Trip Time (RTT) para la primera conexión y de 0-RTT para conexiones posteriores, usando TLS 1.3 [4]. QUIC autentica completamente cada paquete y cifra el mayor porcentaje posible. Los paquetes QUIC se transportan en datagramas UDP para facilitar la implementación en sistemas y redes existentes. Véase en la Figura 2.4 la estructura de los paquetes en QUIC.

Esta sección describe las principales funcionalidades de este protocolo, y justifica su adopción en el contexto de IIoT.

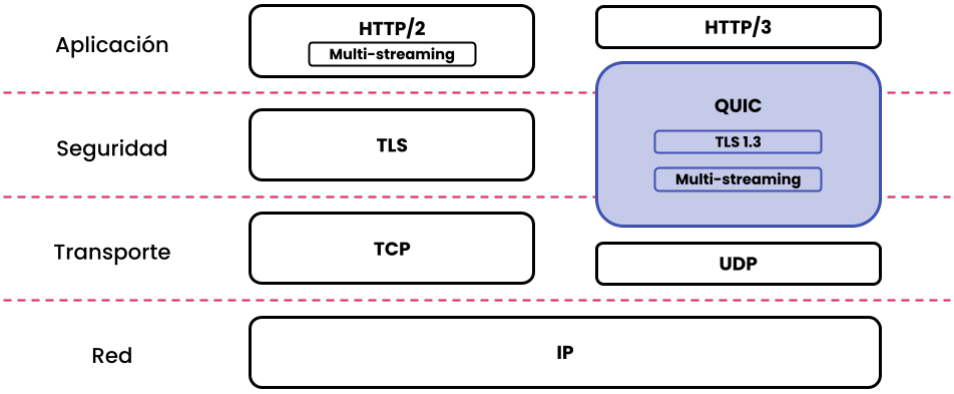


Figura 2.2: Arquitectura de QUIC.

Seguridad

En aplicaciones de IIoT es esencial el uso de protocolos criptográficos que aseguren el intercambio de datos de un extremo a otro a través de la capa de transporte. TLS es el protocolo criptográfico cliente-servidor más común. El cifrado simétrico en TLS permite comunicación autenticada, confidencial y con integridad entre dos dispositivos. La clave para este cifrado simétrico se genera durante el handshake TLS, y es única por conexión. Dado que las conexiones pueden interrumpirse debido a fenómenos como las fases de suspensión, la migración de la conexión y la pérdida de paquetes, el establecimiento de conexiones seguras puede suponer una sobrecarga elevada.

QUIC asume la responsabilidad de asegurar la confidencialidad e integridad de los paquetes. Para ello, utiliza claves derivadas de un handshake TLS, pero en lugar de llevar records TLS a través de QUIC (como sucede con TCP), el handshake TLS y los mensajes de alerta se transmiten directamente a través del transporte QUIC. Además, a diferencia de la operación de TLS sobre TCP, las aplicaciones QUIC que desean enviar datos no lo hacen mediante records TLS de datos de aplicación, sino como tramas QUIC STREAM u otro tipo de tramas, que luego se transportan en paquetes QUIC.

QUIC transporta datos de handshake TLS en tramas del tipo CRYPTO, cada una de las cuales consta de un bloque contiguo de datos de handshake, identificados por un offset y una longitud. Esas tramas viajan en paquetes QUIC y se cifran con el nivel correspondiente (que depende del número de secuencia o Packet Number del paquete, el cual hace que pertenezca a un Packet Number Space u otro).

La función de Authenticated Encryption with Associated Data (AEAD) utilizada para la protección de paquetes QUIC es la AEAD que se negocia para la conexión TLS. Por ejemplo, si la cipher suite acordada entre cliente y servidor es TLS_AES_128_GCM_SHA256, se utiliza la función AEAD_AES_128_GCM.

Establecimiento de la conexión

QUIC combina la negociación de parámetros criptográficos y de transporte para minimizar la sobrecarga y la latencia del establecimiento de la conexión. Durante esta fase pueden emplearse distintos mecanismos de handshake en base a la cantidad de RTTs, como puede verse en la Figura 2.3. QUIC integra el handshake TLS, como se describe con más detalle en [5]. En la Tabla 2.1 se muestra la mejora

que aporta QUIC frente a TCP/TLS en la reducción de los tiempos durante el establecimiento de la conexión.

Tabla 2.1: Comparación de tiempos en el establecimiento de conexión entre TCP y QUIC.

	Tiempo [RTT]	
	Sin conexión previa	**Con conexión previa**
TCP + TLS 1.2	3	2
TCP + TLS 1.3	2	1
QUIC	1	0

- **Handshake inicial (1-RTT).** El cliente no tiene ninguna información previa sobre el servidor. Como se observa en la Figura 2.3a, el cliente envía un primer mensaje denominado *Inchoate CHLO* (equivalente a un Client Hello, pero incompleto). Este mensaje contiene un identificador de conexión generado aleatoriamente (CID), que es utilizado por ambas partes para hacer referencia a esta conexión, y que permite migrar a una nueva dirección IP y/o puerto de un endpoint. A continuación, el servidor responde con un mensaje Reject (REJ). Este mensaje contiene entre otros:

  - La configuración del servidor, incluido el valor público Diffie-Hellmann de larga duración del servidor (PUBS).
  - Una cadena de certificados que autentica al servidor (CRT).
  - Una firma con la clave privada del servidor.
  - Un identificador de la configuración del servidor (SCID).
  - Un token de la dirección de origen.

  Una vez recibido el mensaje REJ, el cliente comprueba la firma y la cadena de confianza, y calcula las claves iniciales como un valor compartido, a partir del PUBS del servidor y su propia clave privada del Diffie-Hellman efímero. Las claves iniciales se pueden utilizar para enviar solicitudes cifradas al servidor antes de que este responda (aunque los datos cifrados con las claves iniciales no proporcionan Forward Secrecy). Tras esto, el cliente envía el CHLO completo, que contiene la clave pública efímera del algoritmo Diffie-Hellman, y también el primer paquete de datos encriptados. El servidor, después de recibir ambos paquetes, puede calcular la clave segura final, basándose en el valor público efímero de Diffie Hellman (KeyShare) del cliente, contenido en el CHLO, y responde con un SHLO que incluye, entre otros, un nuevo token, el KeyShare del servidor y el CID. Además, envía el ACK del primer paquete de datos recibido.

- **Handshake repetido (0-RTT).** El cliente sigue el mismo protocolo que en el escenario anterior a partir del CHLO, como se muestra en la Figura 2.3b. Para ello, el cliente necesita almacenar en caché la configuración del servidor y el último token de la dirección de origen. Esto permite al cliente retomar la conexión con el servidor sin necesidad de enviar nuevamente un mensaje *Inchoate CHLO*, consiguiendo de esta forma enviar un paquete de datos en el primer intercambio con el

servidor. Sin embargo, TLS 1.3 establece algunas limitaciones en el uso de esta funcionalidad, principalmente para evitar posibles ataques por repetición.

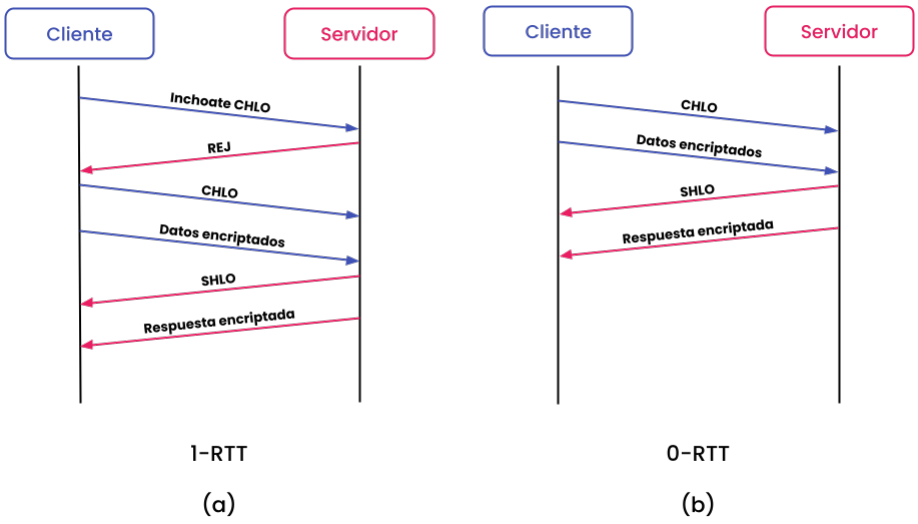


Figura 2.3: Handshakes QUIC.

Multiplexación de streams

QUIC es más eficiente en la compresión de encabezados de la capa de transporte gracias al uso de multiplexación. En lugar de abrir múltiples conexiones desde el mismo cliente, QUIC abre varios streams multiplexados en una única conexión. Además, no sufre el problema del HOL de TCP, al implementar la multiplexación en la capa de transporte (en HTTP/2 la multiplexación es a nivel de aplicación)

Los streams pueden ser unidireccionales o bidireccionales. Se identifican dentro de una conexión mediante un valor numérico, denominado stream ID, que consiste en un número entero de 62 bits (0 a $2^{62} - 1$) que es único para todos los streams de una conexión. El bit menos significativo (0x01) del stream ID identifica al iniciador del mismo. Las transmisiones iniciadas por el cliente tienen ID de transmisión par (con el bit establecido en 0) y las transmisiones iniciadas por el servidor tienen ID de transmisión impar (con el bit establecido en 1). El segundo bit menos significativo (0x02) del stream ID distingue entre streams bidireccionales (con el bit puesto a 0) y streams unidireccionales (con el bit puesto a 1).

Un stream QUIC puede formar un mensaje de aplicación de hasta 2^{64} bytes. Además, en el caso de pérdida de paquetes, no se impide que la aplicación procese paquetes posteriores. La multiplexación es útil en aplicaciones de IIoT, donde se requiere una gran cantidad de transferencia de datos por transacción. Esta característica mejora el rendimiento, por ejemplo, de las actualizaciones remotas y la automatización industrial.

Migración de conexiones

Las conexiones QUIC no están estrictamente vinculadas a una única ruta de red. Los clientes pueden utilizar los identificadores de conexión para transferir las conexiones a una nueva ruta de red. Este diseño

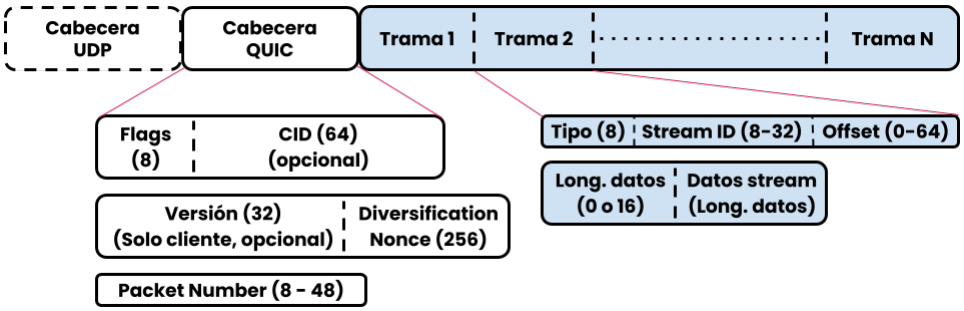


Figura 2.4: Estructura de los paquetes QUIC. Las partes encriptadas aparecen sombreadas y la línea continua indica autenticación.

permite que las conexiones continúen después de cambios en la topología de la red, o nuevas asignaciones de direcciones, como las que pueden ser causadas por NAT-Rebinding.

Cada conexión está identificada por un CID único de 64 bits generado aleatoriamente y que se incluye en la cabecera (parte no cifrada). El uso de CIDs habilita también el multipath probando un nueva ruta de conexión. Este proceso se denomina *path validation*.

Control de flujo y congestión

Similar a TCP, QUIC implementa un mecanismo de control de flujo para evitar que el buffer del receptor se inunde con datos. Emplea un esquema basado en límites en el que un receptor anuncia el límite de bytes totales que está preparado para recibir en un stream determinado o para toda la conexión. Esto conduce a dos niveles de control del flujo de datos en QUIC:

- **Control de flujo a nivel de stream**. Limita el buffer a nivel de stream. Un receptor QUIC comunica su capacidad de recibir datos mediante la publicación periódica del desplazamiento absoluto de bytes por stream en las tramas de actualización de la ventana, para los paquetes enviados, recibidos y entregados.
- **Control de flujo a nivel de conexión**. Evita que los transmisores excedan la capacidad del buffer de un receptor para una conexión, al limitar el total de bytes de datos enviados en todos los streams.

Un receptor establece límites iniciales para todos los streams a través de los parámetros de transporte durante el handshake. Posteriormente, un receptor puede enviar tramas `MAX_STREAM_DATA`, con el correspondiente stream ID, o tramas `MAX_DATA`, para anunciar límites más grandes.

QUIC proporciona la realimentación necesaria para implementar un control de congestión y entrega confiable. En la sección 6 de la RFC 9002 [6] se describe un algoritmo para detectar, y recuperar, la pérdida de datos. En la Sección 7 se describe un ejemplo de algoritmo de control de la congestión utilizado por QUIC.

TCP sufre un problema de ambigüedad en sus retransmisiones. Cuando un paquete no llega a su destino en un intervalo de tiempo razonable, se retransmite con el mismo número de secuencia. Esto

provoca que, tras la recepción del posterior ACK, no se pueda distinguir si el reconocimiento corresponde al paquete original o al reenviado. Este problema no afecta demasiado a la comunicación, pero solucionarlo ayudaría en la detección de las pérdidas. Por ello, QUIC utiliza números de secuencia (Packet Numbers) únicos, incluso en los paquetes retransmitidos. Además, un ACK de QUIC incluye explícitamente el retraso sufrido desde la recepción de un paquete y el propio reconocimiento del paquete por parte del receptor. Esto permite, junto con los crecientes números de secuencia, un cálculo más preciso del RTT, mejorando todavía más la detección de paquetes perdidos.

Las señales que proporciona QUIC para el control de la congestión son genéricas y están diseñadas para admitir diferentes algoritmos. Un transmisor puede elegir unilateralmente un algoritmo diferente para usar, como pueden ser New Reno, CUBIC o BBR. Al igual que en TCP, los paquetes que contienen solo tramas ACK no están controlados por la congestión, sin embargo, a diferencia de TCP, QUIC puede detectar la pérdida de estos paquetes, y usar esa información para ajustar el control de la congestión o la tasa de envío de paquetes ACK.

2.3. Algoritmos de control de congestión

TCP introduce el concepto de "ventana" para lograr establecer un control de flujo de tráfico y gestionar las conexiones entre dos dispositivos. Así, la implementación básica (ventana deslizante) establece un tamaño de ventana por conexión(*receive window*, rwnd), que representa la cantidad de paquetes que el emisor puede enviar al receptor sin esperar reconocimiento.

A mediados de los años 80, con la popularización de Internet, el protocolo de ventana deslizante que gestiona la conexión en función de las capacidades del buffer del receptor empieza a no ser suficiente, frente a los problemas de congestión asociados con la red. Para adecuar la transmisión dependiendo del nivel de congestión, TCP introduce la ventana de congestión o cwnd, con la que se busca regular la cantidad de paquetes enviados en función de la congestión.

Los primeros protocolos TCP están basados en la pérdida de paquetes como indicador de congestión, usando mecanismos Additive Increase/Multiplicative Decrease (AIMD). El ejemplo más conocido en la actualidad es New Reno, el cual ha servido de referencia para la creación de otros algoritmos, como es el caso de TCP CUBIC, evolución de Binary Increase Congestion control (BIC). Existen otros esquemas que introducen modificaciones a los protocolos de control de congestión tradicionales, y que proponen un nuevo enfoque para la detección de la congestión, destacando, BBR [7].

2.3.1. NewReno

La implementación de TCP Reno contiene cuatro fases: *Slow Start*, *Congestion Avoidance*, *Fast Retransmit* y *Fast Recovery*. Cuando se inicia una conexión, primero se entra en la fase de *Slow Start*, según la que la ventana de congestión (cwnd) se inicializa a uno, y se establece un umbral (ssthresh). Cada vez que se recibe un ACK, la ventana de congestión aumenta en uno, lo que implica duplicar su valor en cada RTT. Una vez que la ventana de congestión llega al ssthresh, se entra en la fase de *Congestion*

Avoidance. Ahora, la ventana de congestión aumenta en un paquete con cada RTT. Si se reciben 3 ACK duplicados, lo que se interpreta como la pérdida de un paquete, TCP emplea el algoritmo *Fast Retransmit* para retransmitir el paquete inmediatamente. Después, mediante *Fast Recovery*, la ventana de congestión y el ssthresh se reducen, hasta la mitad de la ventana de congestión actual. Si se recibe un nuevo ACK, significa que el paquete perdido se ha retransmitido con éxito, y, por tanto, TCP sale del *Fast Retransmit* y entra en el *Congestion Avoidance*. Existe una segunda forma de detectar la congestión, cuando el contador Retransmission Time-Out (RTO) llega a cero (timeout). En este segundo escenario la posibilidad de congestión real en la red se supone más alta y, por ello, se aplican medidas más agresivas, como son reducir el ssthresh a la mitad del cwnd, establecer este último a 1 y entrar en *Slow Start*. En la Figura 2.5 se muestra un ejemplo de este comportamiento.

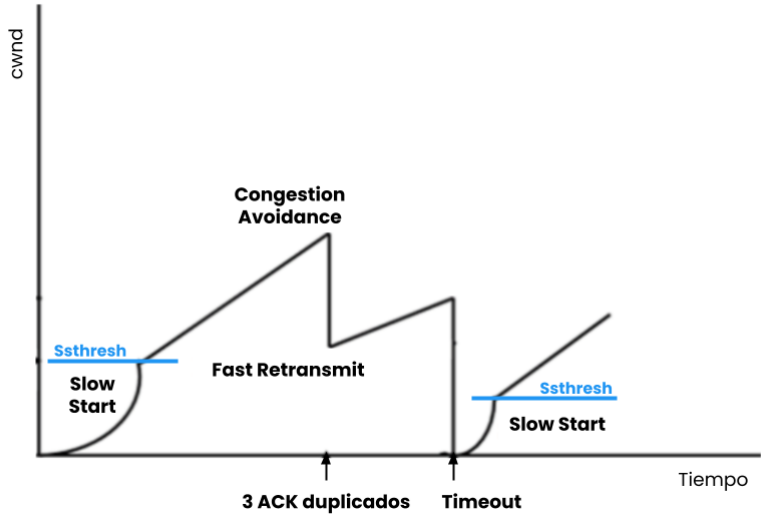


Figura 2.5: Slow Start, Congestion Avoidance, Fast Retransmit y Fast Recovery en TCP.

TCP NewReno [8] modifica el procedimiento *Fast Recovery* de TCP Reno. El *Fast Recovery* de TCP Reno mejora el rendimiento y hace más robusta la red ante la pérdida de un único paquete. En una red real, una vez que ocurre una situación de congestión, habrá muchas pérdidas de paquetes. TCP Reno considerará que hay múltiples congestiones de red y, por lo tanto, la ventana de congestión y el ssthresh se reducirán a la mitad varias veces, provocando una bajada en el rendimiento. Como Reno, NewReno también entra en Fast Retransmit cuando recibe 3 ACK duplicados, sin embargo se diferencia en que no sale de *Fast Recovery* hasta que todos los paquetes enviados en el momento en que se entró en *Fast Recovery* son reconocidos. Cuando NewReno entra en *Fast Recovery*, recuerda el número de secuencia más grande de los paquetes enviados y, al recibir un ACK, pueden darse dos casos:

- ***Partial ACK.*** El ACK recibido no incluye al paquete con el número de secuencia más alto. Se retransmite el primer segmento sin confirmar y se reduce cwnd en la cantidad de datos confirmados. Si confirma al menos un Maximum Segment Size (MSS) de datos nuevos, vuelve a incrementar cwnd en un MSS. Esta medida trata de garantizar que, cuando termine el *Fast Recovery*, haya aproximadamente una cantidad de datos igual al ssthresh pendiente de confirmar en la red. Si además es el primer *Partial ACK*, se reinicia el timer de retransmisión.

- ***Full ACK.*** Todos los paquetes que debían ser retransmitidos se han recibido correctamente, y ya es posible salir del *Fast Recovery*.

2.3.2. CUBIC

CUBIC [9] se ha utilizado en Linux desde la versión del kernel 2.6.19, reemplazando a su predecesor BIC. Desde entonces, se ha integrado en muchas implementaciones. En una red compuesta por enlaces con grandes anchos de banda, un algoritmo de control de congestión que incrementa lentamente la tasa de transmisión puede terminar por desperdiciar la capacidad disponible. CUBIC surge con la idea de tomar ventaja de los enlaces de comunicación actuales que disponen de cada vez mayores anchos de banda. Para ello, propone aumentos en los tamaños de las ventanas de congestión más agresivos, pero evitando llegar a sobrecargar la red.

$$W_{CUBIC} = C \cdot (\Delta - K)^3 + W_{max} \tag{2.1}$$

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

Los cambios en la ventana de congestión se modelan según la función cúbica que se muestra en la Ecuación 2.1, siendo C una constante de CUBIC (por defecto 0.4) usada para comparar el tiempo y el tamaño de la ventana, Δ el tiempo transcurrido desde la última caída de la ventana, y K el tiempo estimado para alcanzar el valor de la ventana cuando se ha producido la última pérdida, W_{max}. La expresión para calcular K depende de un parámetro β (por defecto 0.7), conocido como *CUBIC multiplication decrease factor*, utilizado para reducir la ventana en caso de detectar una pérdida.

Justo antes del evento de pérdida más reciente, CUBIC registra el tamaño máximo de ventana de congestión (W_{max}). Después del evento de pérdida, se reduce la ventana de congestión ($W_{CUBIC} = W_{max} \cdot \beta$). Inicialmente, como puede observarse en la Figura 2.6, el tamaño de la ventana crece muy rápido, en una región cóncava de la función CUBIC, sin embargo, a medida que se acerca al valor W_{max}, el crecimiento se ralentiza. En el momento en que alcanza W_{max}, la variación es casi nula. En el perfil convexo de la función, el crecimiento de la ventana es inicialmente lento. El tiempo transcurrido entre las regiones cóncava y convexa permite que la red se estabilice, ya que la cwnd no aumenta rápidamente con la alta utilización de la red. A medida que el cwnd se aleja del valor W_{max}, el crecimiento de la ventana se acelera, lo que mejora el rendimiento en redes con altos valores de ancho de banda y latencia, es decir, redes donde el producto de ambos o Bandwidth-Delay Product (BDP) sea alto.

Cuando un nuevo flujo se une a la red, los existentes deben ceder parte de su ancho de banda para permitir que el nuevo flujo tenga cierto margen de crecimiento. Para acelerar esta liberación de ancho de banda, CUBIC implementa un mecanismo denominado *Fast Convergence*. Cuando ocurre un evento de congestión, antes de la reducción de la ventana de congestión, cada flujo recuerda el último valor de W_{max}. Si su valor actual de es menor que el anterior ($W_{max_{-1}}$), significa que el punto de saturación experimentado por este flujo se está reduciendo, debido al cambio en el ancho de banda disponible. Ante esta situación, se actúa de acuerdo a la Ecuación 2.3, reduciendo aún más W_{max} y, de esta forma, facilitando que el resto de flujos aumenten el valor de su ventana.

$$W_{max} = W_{max} \cdot \frac{1+\beta}{2} \quad \text{si} \quad W_{max} < W_{max_{-1}} \tag{2.3}$$

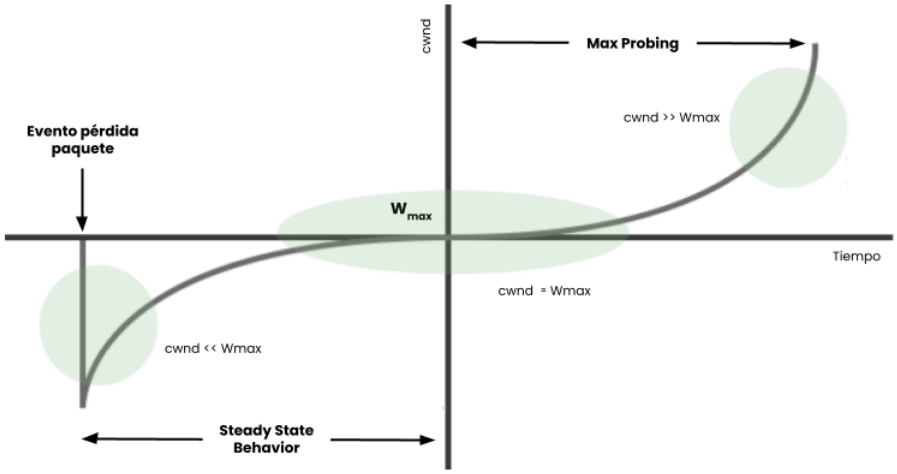


Figura 2.6: Función CUBIC.

Como se ha visto, la agresividad de CUBIC depende principalmente del valor de W_{max} antes de una reducción de ventana. Por ello, CUBIC aumenta su ventana de congestión de manera menos agresiva cuanto menor sea el BDP de la red. En los casos en que la función cúbica de CUBIC aumente su ventana de congestión de forma menos agresiva que el control de congestión estándar de TCP, CUBIC simplemente sigue el tamaño de la ventana del TCP estándar para garantizar que se logre al menos el mismo rendimiento. A esta región en la que CUBIC se comporta como TCP estándar se la conoce como *TCP-friendly region*.

2.3.3. BBR

Punto de trabajo

Desde 1980 se ha interpretado la pérdida de paquetes como congestión. Sin embargo, en la actualidad, con tarjetas de red que trabajan en el orden de los Gbps y chips de memoria de GBs, esta relación entre la pérdida de paquetes y la congestión es cada vez menos acertada.

Al disponer buffers en el cuello de botella de gran tamaño, el control de congestión basado en pérdidas los mantiene llenos, lo que provoca bufferbloat. En el caso contrario, con buffers pequeños, se malinterpreta la pérdida como una señal de congestión, lo que conduce a un bajo rendimiento. La solución a estos problemas requiere una alternativa al control de congestión basado en pérdidas. BBR, desarrollado por Google en 2016, nace de esta idea, aportando una nueva forma de entender dónde y cómo se origina la congestión de la red.

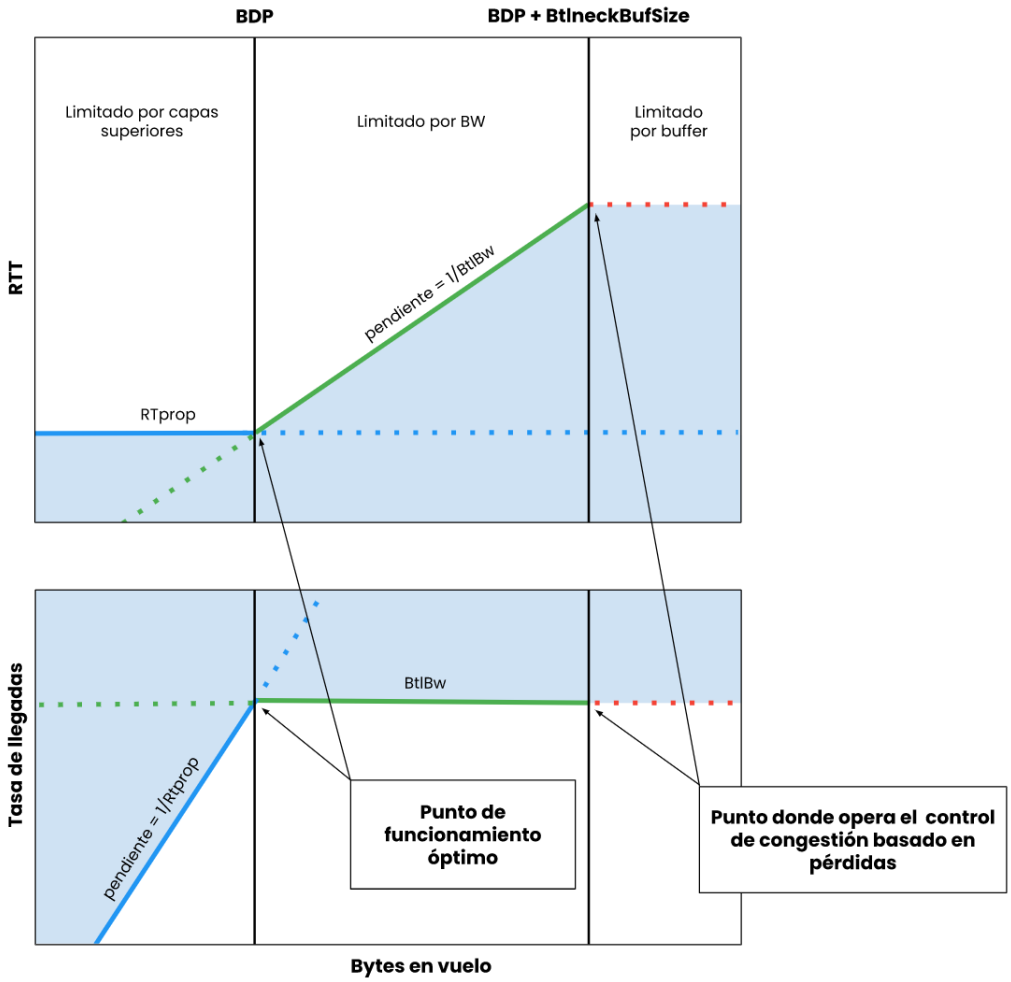


Figura 2.7: Puntos de trabajo control de congestión.

Existen dos restricciones físicas: *RTprop* (tiempo de propagación de ida y vuelta) y *BtlBw* (ancho de banda del cuello de botella). La Figura 2.7 muestra la variación de la tasa de llegadas y el RTT con la cantidad de datos en vuelo (datos enviados pero aún no reconocidos). La restricción *RTprop* son las líneas azules, la restricción *BtlBw* son las líneas verdes, y las líneas rojas representan el limite impuesto por el buffer del cuello de botella. La operación en las regiones sombreadas no es posible, porque violaría al menos una restricción. Se pueden observar tres regiones diferentes (limitada por aplicación, limitada por ancho de banda y limitada por buffer) con un comportamiento cualitativamente diferente.

Las líneas de restricción se cruzan en $Datos\,en\,vuelo = BtlBw \cdot RTprop$, es decir, en el BDP. Superar este punto crea una cola en el cuello de botella, lo que da como resultado la dependencia lineal del RTT con los datos en vuelo que se muestra en la Figura 2.7. Los paquetes se descartan cuando el exceso supera la capacidad del buffer. La congestión es solo una operación sostenida a la derecha de la línea BDP, y el control de la congestión es un esquema para limitar qué tan lejos a la derecha opera una conexión en promedio. El control de congestión basado en pérdidas (como New Reno y CUBIC) opera en el borde derecho de la segunda región (limitado por el ancho de banda), utilizando toda la capacidad del cuello de botella a costa de un alto *delay* y pérdida frecuente de paquetes. Por el contrario, BBR busca alcanzar el punto de funcionamiento óptimo de la conexión (transmitir a máxima velocidad con el mínimo RTT).

Para ello, los principales objetivos de BBR son la eliminación del efecto bufferbloat y el aprovechamiento total del canal. Gracias a la eliminación de colas intermedias se procura transmitir induciendo el menor retardo posible.

Funcionamiento

Dado que el estado de la conexión varía en el tiempo, se deben calcular constantemente una serie de parámetros. Los más destacados son *RTprop* y *RTprop*. Definidos en el apartado anterior, *RTprop* hace referencia al RTT mínimo de la comunicación y *RTprop* a los paquetes en vuelo. En las Figuras 2.8 y 2.9 se observan los diferentes estados que existen en BBR. A continuación, se detallan los objetivos y el funcionamiento de cada uno de ellos:

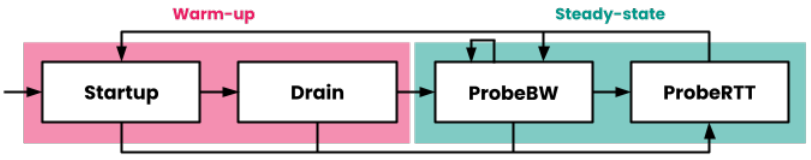


Figura 2.8: Diagrama de estados BBR.

- ***Startup.*** Estima el valor de *RTprop*. Para ello, dado que el rango a explorar puede ser muy amplio, BBR realiza una búsqueda exponencial, duplicando la tasa de envío en cada RTT. Esto encuentra el *RTprop* en $O(log_2(BDP))$ RTTs, pero a costa de crear un exceso de más de 2BDP en la cola.

- ***Drain.*** Elimina el exceso de carga que se haya podido inducir en las colas intermedias durante la fase anterior, reduciendo la cantidad de información a transmitir. El comportamiento inicial de BBR es similar al de CUBIC. Sin embargo, a diferencia de CUBIC, BBR logra drenar completamente la cola generada en la fase de *Startup*.

- ***ProbeBW.*** Sondea el ancho de banda mediante un proceso llamado *gain cycling*, el cual ayuda a que los flujos de BBR alcancen un alto *throughput*, bajo *delay* en la cola y converjan a un reparto justo del ancho de banda disponible. Consta de ocho estados cíclicos que buscan lograr transmitir aprovechando toda la capacidad del canal. En el primero de estos estados se comprueba si han cambiado las condiciones de la red, sobrecargándola con el envío del 125 % de su capacidad. En el segundo estado, se drena cualquier cola resultante del estado anterior, transmitiendo al 75 %. Si las condiciones de la red no han variado, durante los seis estados restantes se transmite al 100 % de la capacidad. Cada uno de los ocho estados dura aproximadamente el *RTprop* estimado. BBR se encuentra en *ProbeBW* durante la mayor parte de la conexión.

- ***ProbeRTT.*** En cualquier otro estado si la estimación de *RTprop* no se ha actualizado (obteniendo una medición de RTT más baja) durante más de 10 segundos, BBR pasa a este estado, en el que reduce el cwnd a un valor muy pequeño (cuatro paquetes). Después de mantener este número mínimo de paquetes en vuelo durante al menos 200 ms o un RTT, cambia al estado *ProbeBW* si se está utilizando todo el ancho de banda disponible o, en caso contrario, al estado *Startup*.

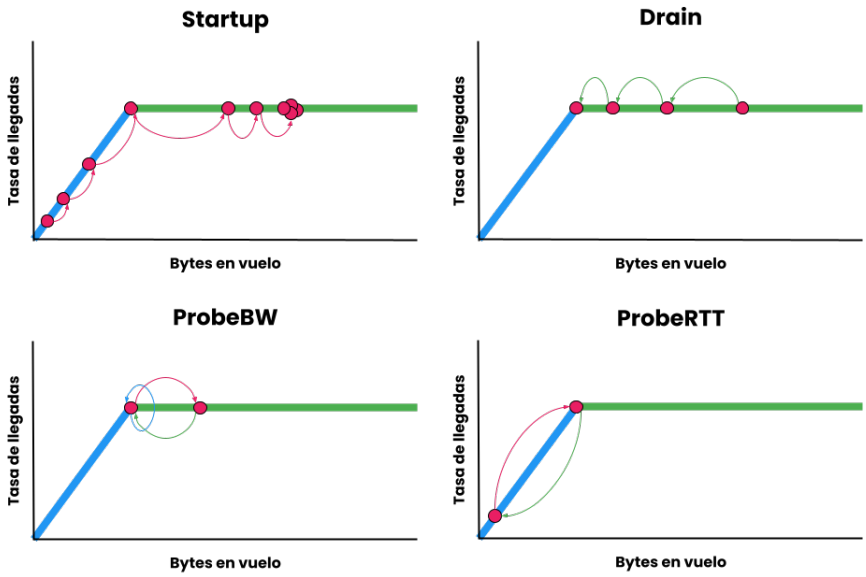


Figura 2.9: Estados BBR.

2.4. Simulación

2.4.1. Introducción al *ns-3* Network Simulator

Network Simulator (ns) es un simulador de redes basado en eventos discretos que comenzó a desarrollarse en 1989 como una variante del simulador de red REAL. La variante *ns-3* surge en el año 2005, siendo su primera versión (*ns-3.1*) de junio de 2008. *ns-3* se ha desarrollado para proporcionar una plataforma de simulación de red abierta y extensible, para la investigación y la docencia.

ns-3 utiliza C++ como lenguaje de programación, lo que proporciona varias ventajas, entre ellas, facilitar la inclusión de implementaciones basadas en C. Los elementos básicos de una simulación vienen representados por clases C++ que simulan los elementos básicos hardware/software de un escenario real. Estos son:

- **Node.** En *ns-3*, la abstracción básica del dispositivo informático se denomina nodo. Representa un dispositivo al que agregar aplicaciones, pilas de protocolos y tarjetas periféricas con sus controladores asociados.
- **Channel.** Representa un canal de comunicación y su comportamiento (inalámbrico, punto a punto, etc.)
- **Net Device.** Dispositivo de red, es una tarjeta de red inalámbrica (e.j WiFi) o cableada (e.j ethernet). Un Net Device se “instala” en un Node para permitirle comunicarse con otros Nodes a través de Channels.
- **Application.** Abstracción básica para un programa de usuario que genera alguna actividad a simular. Una aplicación está asociada a un dispositivo (Node).

ns-3 presenta una arquitectura similar a las computadoras Linux, con interfaces internas (network to device driver) e interfaces de aplicación (sockets) que se adaptan bien a cómo se construyen las computadoras en la actualidad. *ns-3* se centra en las capacidades de emulación que permiten utilizarlo en testbeds y con dispositivos y aplicaciones reales, nuevamente con el objetivo de reducir las posibles discontinuidades al pasar de la simulación a la experimentación. El entorno de simulación desarrollado para este proyecto, explicado en el Capítulo 3, se realiza a partir de la versión *ns-3.34*, con fecha 14 de julio de 2021. Entre las principales novedades de esta versión destaca la posibilidad de emplear TCP BBRv1 para el control de la congestión. Los resultados obtenidos de este y otros algoritmos de control de congestión se presentan también en la memoria (Capítulo 4).

2.4.2. Módulo QUIC en *ns-3*

La implementación nativa de QUIC en *ns-3* empleada se describe en [10]. El código se puede encontrar en un repositorio [11] y actualmente continúa en desarrollo. Esta implementación amplía la estructura de código de TCP en *ns-3*, con las características que presenta QUIC como son: la multiplexación de flujo, el handshake de baja latencia, o la mejora del SACK a través de las tramas ACK. Además, se diseñó la implementación del socket QUIC de modo que sea posible emplear tanto los algoritmos de control de congestión heredados de TCP como el nuevo control de congestión de QUIC (QuicBbr). Entre todas las clases destacan las siguientes:

- **QuicSocketBase.** Es la clase principal de la implementación. Modela las funcionalidades básicas de un socket QUIC, como sucede con TcpSocketBase en TCP. Cada cliente creará una instancia de un único objeto QuicSocketBase, mientras que el servidor creará un nuevo socket para cada conexión entrante. Un objeto QuicSocketBase recibe y transmite paquetes y reconocimientos QUIC, contabiliza las retransmisiones, realiza el control de flujo y congestión a nivel de conexión, se encarga del handshake inicial y del intercambio de los parámetros de transporte, y maneja el ciclo de vida y la máquina de estados de una conexión QUIC.
- **QuicL4Protocol.** El socket QUIC está vinculado a un socket UDP subyacente a través de un objeto QuicL4Protocol, que también maneja la creación inicial del objeto QuicSocketBase, activa el socket UDP para vincularse y conectarse, y maneja la entrega de paquetes entre el socket UDP y QuicSocketBase.
- **QuicStreamBase.** Modela las funcionalidades de un stream. Almacena los datos de la aplicación, realiza un control de flujo a nivel de stream y entrega los datos recibidos a la aplicación.
- **QuicL5Protocol.** Varios objetos QuicStreamBase están conectados a un solo QuicSocketBase a través de un objeto que pertenece a la clase QuicL5Protocol. Un QuicSocketBase contiene un puntero a un objeto QuicL5Protocol, y este último contiene un vector de punteros a múltiples instancias de QuicStreamBase. La clase QuicL5Protocol crea y configura los streams, y se encarga de entregar paquetes o tramas para que se transmitan y reciban a través de los streams y el socket.

Una de las principales características de la implementación de TCP de las últimas versiones de *ns-3* es la modularidad del algoritmo de control de congestión con respecto al código del socket. Se

han implementado y agregado al “internet module” de *ns-3* una serie de algoritmos de control de la congestión, como TCP NewReno, BIC, CUBIC o BBR, que amplían la clase TcpCongestionOps que es la que contiene las funcionalidades básicas de control de congestión. Aprovechándose de esto, la clase QuicSocketBase se puede utilizar en modo heredado, haciendo posible utilizar los algoritmos de control de congestión de TCP. Además, también es posible utilizar algoritmos más personalizados, que aprovechen la información adicional que proporciona el socket QUIC, y desarrollar así métodos de control de congestión solo para QUIC. La compatibilidad con los algoritmos de control de congestión de TCP heredados se logra al tener una instancia de TcpCongestionOps como objeto de control de congestión básico en QuicSocketBase. Cuando el algoritmo de control de congestión se establece en QuicSocketBase, SetCongestionControlAlgorithm verifica si el algoritmo especificado extiende QuicCongestionControl y, en este caso, establece un indicador (m_quicCongestionControlLegacy) en falso. De lo contrario, este último se establece en verdadero, activando el modo heredado.

3 | Entorno de simulación

A continuación, se describen los escenarios y parámetros utilizados en las simulaciones y que han permitido obtener los resultados que se muestran en el Capítulo 4. Como ya se ha mencionado anteriormente, se ha usado el simulador *ns-3*, introducido brevemente en la Sección 2.4.1.

3.1. Implementación

Para implementar un escenario en *ns-3* los pasos a realizar son similares a los que habría que llevar a cabo en un escenario real y pueden resumirse en:

1. Crear los nodos, el canal, la aplicación y la pila de protocolos.
2. Instalar los dispositivos de red (*NetDevices*) en los nodos.
3. Incluir los dispositivos de red en el canal.
4. Instalar en cada nodo la pila de protocolos, y asignarle la dirección IP.
5. Instalar en cada nodo la aplicación, y programar los tiempos de encendido/apagado de la misma.

Así, para la creación de los diferentes escenarios primero se crean tres objetos de la clase *NodeContainer*. Cada uno hace referencia a uno de los roles presentes en MQTT (publisher, subscriber y broker) descritos en la Sección 2.1. El uso de *NodeContainer* permitirá ampliar fácilmente el número de nodos en la simulación, llamando al método *Create* e indicando el número de publishers, subscribers o broker que se necesiten.

Se emplea la clase *InternetStackHelper* para instalar la pila de protocolos de Internet en los nodos utilizados en las simulaciones TCP, y su equivalente, *QuicHelper*, para las simulaciones con QUIC. Después, con *Ipv4AddressHelper* se asignan las direcciones IP a cada interfaz.

Normalmente las aplicaciones se instalan en los nodos mediante un *helper* o asistente. El *helper* toma un *NodeContainer*, que contiene un conjunto de *Ptr<Node>*, y para cada uno de esos nodos crea una instancia de una aplicación (*Ptr<Application>*), la instala y la agrega a un *ApplicationContainer*. Para cada escenario se crean, por tanto, tres *ApplicationContainer* (*publisherApps*, *subscriberApps* y *brokerApp*), que almacenan las aplicaciones instaladas en los tres *NodeContainers*.

En el caso de TCP, se instala, mediante *OnOffHelper*, la aplicación *OnOffAplication* incluida en *ns3*, que permite generar el tráfico TCP. En los escenarios QUIC, se utilizan las aplicaciones *QuicPublisher*

y *QuicSubscriber* creadas tomando como base la clase *QuicClient* del módulo de QUIC. Tanto en TCP como en QUIC fue necesario crear una aplicación que imitase el comportamiento de un broker MQTT. La solución que se implementó fue una aplicación que crea dos sockets, uno para la recepción de los paquetes del publisher y otro para reenviarlos al subscriber correspondiente. Las direcciones IP y los puertos de publisher y subscriber se indican al instalar la aplicación, asociando así un subscriber con uno o varios publishers o viceversa. De esta forma, se evita tener que desarrollar el sistema de temas de MQTT.

Para llevar a cabo las simulaciones se ha desarrollado un script en Python (Listado A.1) que permite simular todos los escenarios para todas las posibles combinaciones de protocolo de transporte/algoritmo de control de congestión, y variar parámetros tales como el ancho de banda, la tasa de generación de tráfico o el RTT de la red automáticamente. Los resultados generados se recogen en ficheros de texto, organizados en diferentes directorios para posteriormente ser procesados en MATLAB y obtener así las gráficas que se incluyen en en el Capítulo 4.

3.2. Tracing

ns-3 dispone de un sistema de trazas que permite, para cada nodo, registrar ciertas métricas tales como el tamaño de la ventana de congestión, el RTT, el RTO o los bytes en vuelo. Para el análisis a nivel de paquete, *ns-3* es capaz de generar archivos pcap (*paquete capture*) y, mediante la clase *FlowMonitor*, también es posible recuperar estadísticas de un flujo, como delay, jitter o el número de paquetes enviados y recibidos.

Para obtener todas los datos analizados durante el proyecto se ha recurrido principalmente al uso de *trace sources*, ver Listado A.2. Esta herramienta de *ns-3* permite recopilar todos los cambios que hayan sufrido determinadas variables durante la simulación. Cada objeto *ns-3* cuenta con una lista de *trace sources* asociadas, documentadas en la API.

Las *trace sources* no son útiles por sí mismas; deben estar conectadas a unas funciones que manejen la información que proporcionan. En cada escenario ha sido necesario incluir estas funciones, denominadas *trace sinks*, que se detallan en el Listado A.4. Esta división explícita permite distribuir varias *trace sources* por el sistema, en lugares que los autores del modelo crean que pueden ser útiles, introduciendo una sobrecarga de ejecución muy pequeña. Una *trace sink* se ejecuta cuando se produce el evento de la *trace source* a la que está conectada, y recibe como parámetro un objeto del tipo indicado por el *TracedCallback* o *TracedValue* de su *trace source*. Si es *TracedValue*, la función debe incluir al menos dos parámetros (uno para el valor antiguo y otro para el nuevo) y si es *TracedCallback*, solo es obligatorio un parámetro. Una vez se conoce el nombre de la *trace source* y se dispone de una *trace sink* hay que realizar una llamada a la función *TraceConnectWithoutContext* sobre el objeto que contiene la *trace source* con los parámetros: nombre de la *trace source* y *MakeCallback*, con la referencia de la función *trace sink*.

Para recopilar todos los datos se han empleado las *trace sources* tanto de los sockets (TCP y QUIC) como de las distintas aplicaciones (*publisher, broker y subscriber*). La clase *TcpSocketBase* del modelo TCP de *ns-3* incluye un gran número de ellas. Se han usado: RTT, *BytesInFlight* y *CongestionWindow*. *CongestionWindow* se ha modificado ligeramente para que devuelva también el RTT en el momento que

se produce el cambio en la ventana. En el caso de QUIC, al estar basado en el modelo TCP, algunas de las *trace sources* ya se encontraban en la implementación. Para el resto, hizo falta modificar el código de *QuicSocketBase* para incluirlas. Para las aplicaciones, se crearon las *trace sources* correspondientes al envío de paquetes (*Tx*) y a la recepción de los mismos (*Rx*). Con ambas es posible obtener el tiempo de envío/recepción de los paquetes y la información del paquete. Además, a partir de los tiempos y el tamaño de los paquetes, se calcula el *throughput* en tiempo de ejecución y se devuelve en un fichero. Al final, el script de cada escenario devuelve un fichero para las siguientes métricas en cada nodo:

- RTT
- Ventana de congestión
- Bytes en vuelo
- Tiempo de envío/recepción de cada paquete
- *Throughput*

3.3. Topologías

3.3.1. Punto a punto

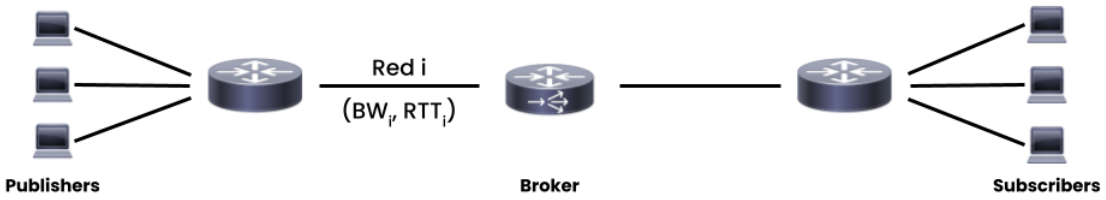


Figura 3.1: Topología punto a punto.

La primera topología adoptada incluye un conjunto de publishers y subscribers, ambos conectados mediante un enlace punto a punto (*PointToPointChannel*) a un broker. Se muestra una representación de la topología en la Figura 3.1. Los enlaces punto a punto se configuran mediante los siguientes atributos: *DataRate*, *Delay* y *ReceiveErrorModel*.

Tabla 3.1: Parámetros de los distintos tipos de redes usados en los testbeds.

	Red 1	**Red 2**	**Red 3**
	WiFi	4G/LTE	Satélite
BW [Mbps]	20	10	1.5
RTT [ms]	25	100	600
FER [%]	0 … 5	0 … 5	0 … 5

El cuello de botella de la red se encuentra entre el publisher y el broker. En la Tabla 3.1 se muestran los parámetros utilizados para reflejar diferentes tecnologías de red: WiFi, Celular y Satelital. El enlace

entre broker y publisher se configura como un enlace de altas prestaciones, con alta capacidad, delay mínimo y sin pérdidas.

3.3.2. WiFi

Figura 3.2: Topología WiFi.

En la segunda topología se sustituye el primer enlace punto a punto por uno WiFi. La clase que hereda de la clase *NetDevice* para WiFi es la clase *WifiNetDevice*, la cual modela un controlador de interfaz de red inalámbrica basado en el estándar IEEE 802.11. La implementación es modular y la descripción general del diseño de cada capa se muestra en la Figura 3.3. Existen tres subcapas de modelos:

- **Modelos de la capa PHY.** Se encargan de las operaciones y funciones de capa PHY, y son los principales responsables de modelar la recepción de paquetes y el seguimiento del consumo de energía. Cada paquete recibido se evalúa probabilísticamente para decidir si se trata de una recepción exitosa o fallida. La probabilidad depende de la modulación, de la relación señal/ruido (e interferencia) del paquete y del estado de la capa física.

- **Modelos MAC Low.** Modelan funciones como el acceso al medio (DCF y EDCA), la protección de tramas (RTS/CTS) y el reconocimiento (ACK/BlockAck).

- **Modelos MAC High.** Implementan en WiFi procesos no críticos en el tiempo, como la generación de beacons a nivel MAC, sondeo y máquinas de estados de las asociaciones del nivel MAC, además de un conjunto de algoritmos de control de tasa binaria, usados por los modelos MAC Low.

A medida que evoluciona el estándar IEEE 802.11, se han ido incorporando funciones, y es cada vez más difícil tener un solo componente que maneje todas las secuencias de intercambio de tramas permitidas. Por ello, en la capa baja de MAC existe una jerarquía de clases *FrameExchangeManager* que permite que el código sea más limpio y escalable, evitando al mismo tiempo código duplicado. *FrameExchangeManager* maneja, también, la agregación de tramas, las retransmisiones, la protección y el reconocimiento. Es en esta clase donde se introduce el modelo de error empleado en el proyecto. De esta forma, el broker descarta algunos paquetes recibidos de la capa PHY según una variable aleatoria configurable en el escenario.

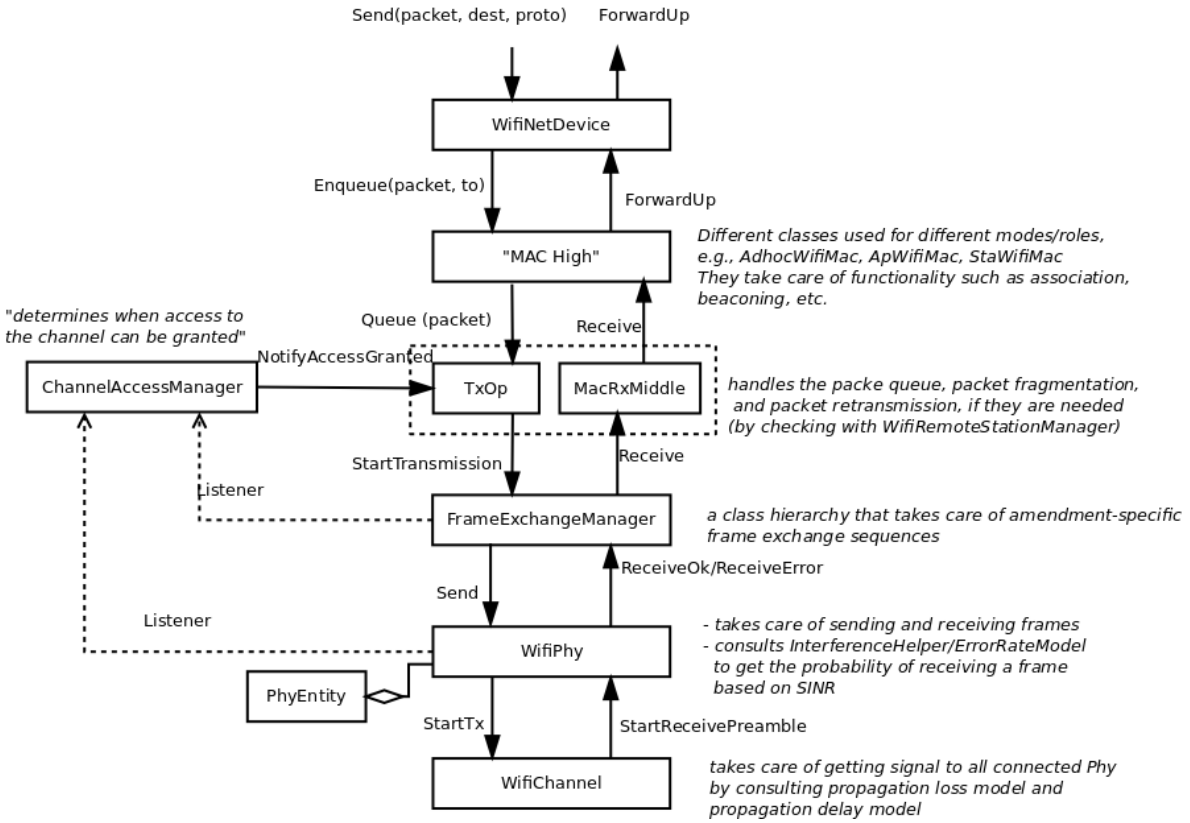


Figura 3.3: Arquitectura del módulo WiFi.

3.4. Dificultades

En esta sección se recogen los principales obstáculos que se han encontrado durante la realización de este proyecto en *ns-3* y las soluciones que se han adoptado.

Un primer obstáculo surge debido a una limitación presente en la implementación de QUIC, por la cual no es posible crear más de un socket UDP en un nodo. Esto resultó un problema a la hora de diseñar el broker, ante el que se optó por dividirlo de manera virtual en dos nodos. El primero se encarga de recibir los paquetes del publisher, y el segundo de enviarlos al subscriber. Para ello, se diseñaron dos aplicaciones, incluyendo la correspondiente al segundo nodo un método (ver Listado A.5) que permitiera enviarle el puntero de cada paquete. De esta forma, cada vez que el primer nodo recibe un paquete por parte del publisher, llama a dicho método para que el segundo nodo tenga acceso al paquete y pueda enviarlo al subscriber.

A diferencia de las simulaciones con TCP, empleando QUIC se han encontrado algunos problemas, sobre todo en escenarios donde se buscaba generar mayor congestión en la red. La implementación de QUIC en *ns-3* todavía continúa en desarrollo y, de hecho, durante el transcurso de este proyecto ha ido recibiendo actualizaciones en su código que han ido solventando algunos de estos problemas. Las simulaciones han sido planteadas con estas limitaciones en mente, adaptantado los escenarios y realizando las simulaciones suficientes para garantizar la fiabilidad de los resultados obtenidos.

4 Resultados

En este capítulo se incluyen los resultados obtenidos al ejecutar las simulaciones en *ns-3* para los diferentes escenarios utlizados para evaluar el comportamiento de TCP y QUIC. Se emplea MATLAB para procesar los datos que devuelve el simulador y representar las gráficas que se muestran a lo largo del capítulo.

4.1. QUIC vs TCP

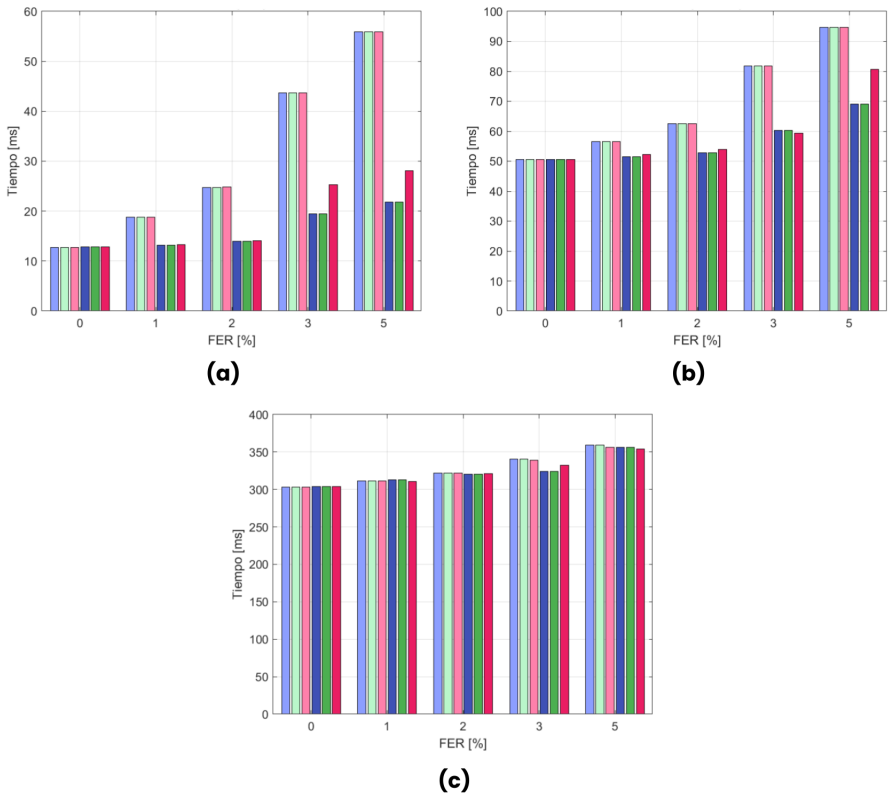


Figura 4.1: Tiempo medio que tarda un paquete, enviado por un publisher, en llegar a un subscriber para los tres tipos de redes estudiados: WiFi (a), Celular (b) y Satelital (c). Aparecen representados TCP y QUIC, en color claro y oscuro respectivamente, empleando diferentes algoritmos de control de congestión: NewReno (azul), CUBIC (verde) y BBR (rosa).

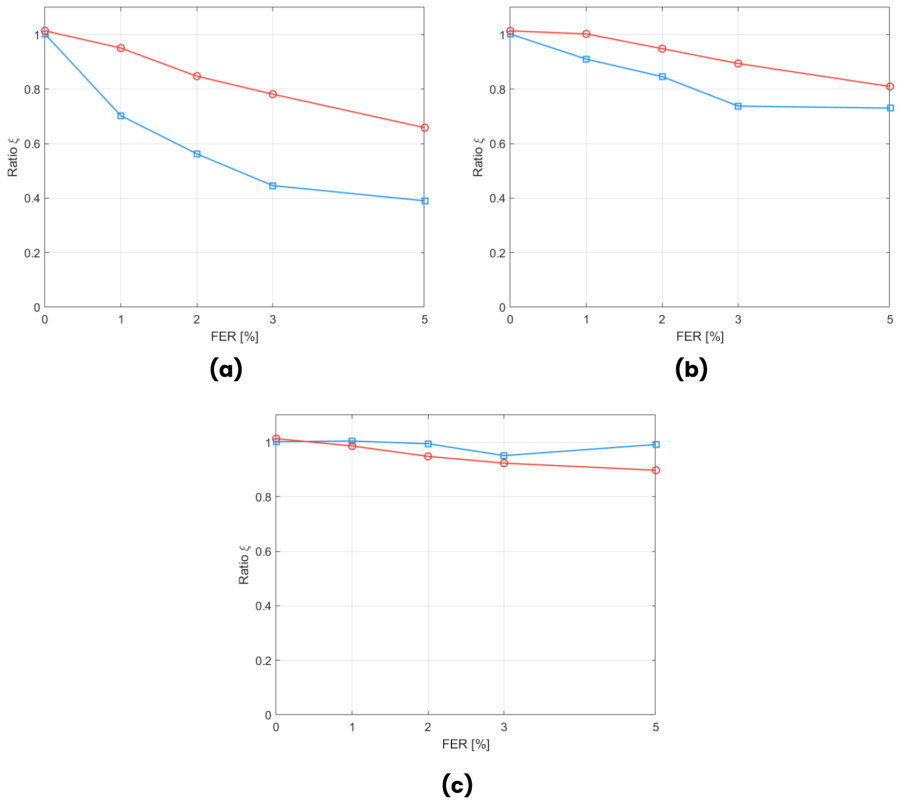


Figura 4.2: Ratio de tiempos entre QUIC y TCP (azul) frente a los resultados obtenidos en [12] (rojo) para los tres tipos de redes: WiFi (a), Celular (b) y Satelital (c).

En el primer escenario se busca complementar los resultados expuestos en [12], donde se observa como MQTT/QUIC logra superar o, al menos, igualar el rendimiento de MQTT/TLS/TCP. Para ello, empleando la topología descrita en la Sección 3.3.1, un publisher envía 1000 paquetes al broker, utilizando una tasa constante de un paquete por segundo. Se realizan simulaciones de las tres configuraciones de red, obteniendo los tiempos de envío y recepción de cada paquete por parte del subscriber. A partir de estos datos, se calcula el *delay* que sufre cada paquete.

Como se puede ver en la Figura 4.1, donde aparecen las medias de los valores observados, QUIC supera a TCP en todos los casos, especialmente en redes con RTT bajos (WiFi). Esta mejora se vuelve más relevante en redes con pérdidas. Por el contrario, cuando el RTT aumenta, y pasa a ser la principal causa del *delay*, la mejora que proporciona QUIC es mucho menos notoria, como puede verse de forma más clara en la conexión satelital. Dado que los paquetes que se transmiten son bastante cortos, se puede concluir que el impacto del ancho de banda es casi insignificante.

La forma de realizar los experimentos en [12] difiere un poco de la empleada en este trabajo. Mientras que aquí se calcula el tiempo que tarda un mensaje Publish en viajar desde un publisher a un subscriber, en dicho artículo se mide el tiempo en recibir 100 mensajes, para posteriormente calcular el ratio que se define en la ecuación 4.1. De esta forma, cuando $\xi < 1$, QUIC mejora a TCP.

$$\xi = \frac{T_{QUIC}}{T_{TCP}} \tag{4.1}$$

Para comparar estas medidas con las obtenidas en este proyecto, con el tiempo medio de los 1000 paquetes se calcula el ratio de la misma forma, obteniendo los resultados de la Figura 4.2. Se observa que los resultados obtenidos en [12] son algo más conservadores, aunque la tendencia según la que QUIC mejora los tiempos medidos en TCP, en escenarios de menor RTT y a medida que aumentan las pérdidas, aparece de forma clara en ambos casos. Además, destaca que la única situación donde $\xi > 1$, es decir, donde TCP supera a QUIC, aparece en redes sin pérdidas. Esta ligera mejora mostrada por TCP, cercana al 1.003 y al 1.01 en [12], es debida a la sobrecarga que QUIC incorpora.

En un segundo escenario (Sección 3.3.2) se dispone de en un enlace WiFi entre publisher y broker, y un enlace punto a punto entre broker y subscriber, en el que se introduce cierto *delay*. Las simulaciones se realizan eliminando las retransmisiones definidas en el protocolo IEEE 802.11 MAC, ya que, de esta forma, se asegura que la fiabilidad de la red recae en el protocolo de transporte. Se llevan a cabo las mismas medidas que en el caso anterior, obteniendo los resultados representados en la Figura 4.3. Se observa como el rendimiento de QUIC es mayor que el de TCP, sobre todo, ante valores altos de FER. Sin embargo, con el aumento del *delay* en el enlace broker-subscriber, la mejora que aporta QUIC ya no es tan significativa.

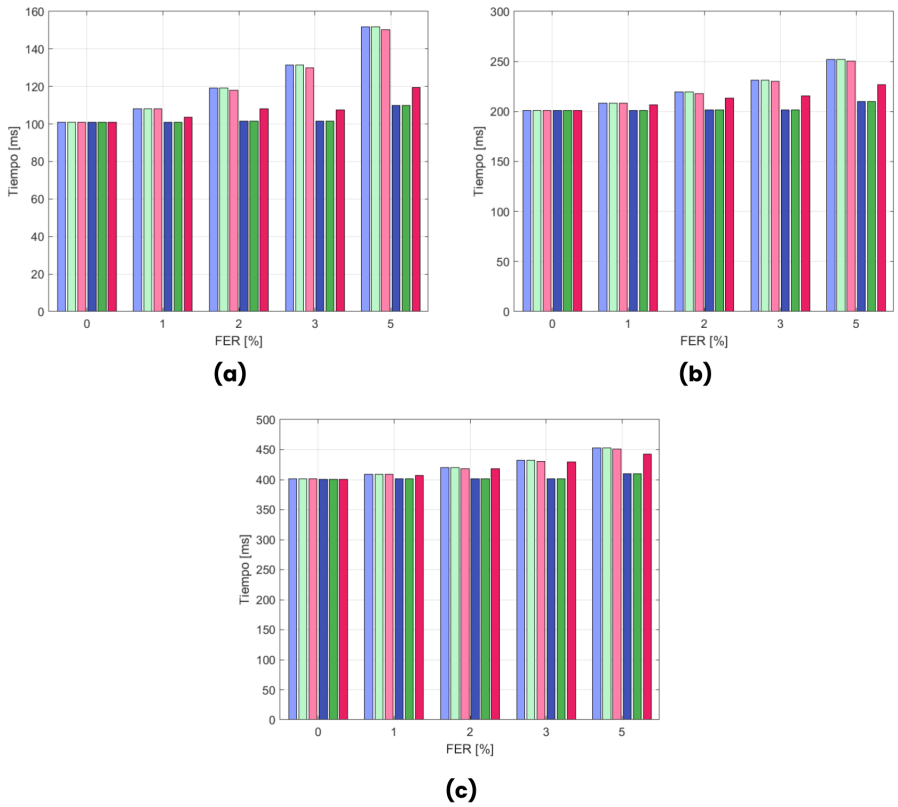


Figura 4.3: Tiempo medio que tarda un paquete, enviado por un publisher, en llegar a un subscriber. El publisher y el broker se encuentran en la misma red WiFi, mientras que la comunicación con el subscriber es mediante un enlace punto a punto, donde se introducen varios valores de *delay*: 100 ms (a), 200 ms (b) y 300 ms (c). Aparecen representados TCP y QUIC, en color claro y oscuro respectivamente, empleando diferentes algoritmos de control de congestión: NewReno (azul), CUBIC (verde) y BBR (rosa).

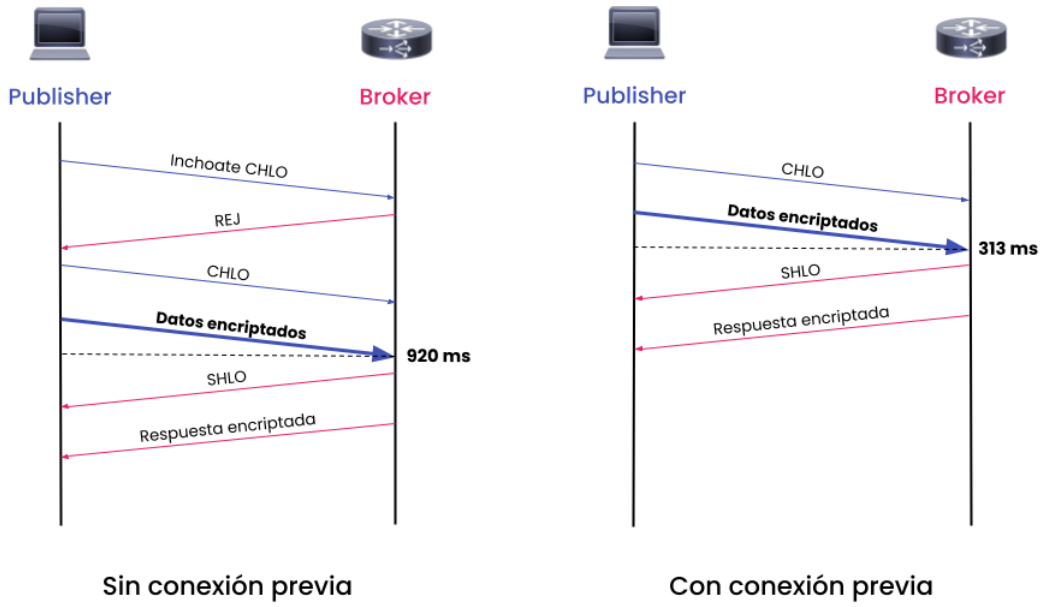


Figura 4.4: Ejemplo de establecimiento de conexión QUIC entre publisher y broker en una red satelital (RTT = 600 ms).

Otra de las ventajas que presenta QUIC, comentada en la Sección 2.2, es una menor latencia en el establecimiento de la conexión frente al tradicional esquema TCP/TLS. Esta funcionalidad resulta interesante en un entorno donde las reconexiones entre los diferentes dispositivos son bastante comunes, ayundando a reducir de forma notable la variabilidad en los tiempos de recepción, como se demuestra en [13]. En la Figura 4.4 se prueba el handshake de QUIC en una red satelital. Si es la primera vez que el publisher conecta con el broker, el primer mensaje para el establecimiento de la conexión MQTT se enviará tras el handshake de 1-RTT (inchoate CHLO + REJ), mientras que si el publisher almacena en caché la configuración del broker de una conexión anterior, es posible enviar el primer mensaje MQTT de forma casi inmediata.

4.2. Congestión

Por último, el tercer escenario se centra todavía más en el papel de los algoritmos de congestión. En el Anexo B se incluyen todos los resultados de este conjunto de experimentos. Los resultados se organizan de acuerdo al número de flujos presentes en la red.

4.2.1. Un flujo

Como se observa en las Figuras 4.5 y 4.6, los tres algoritmos de control de congestión muestran un crecimiento muy agresivo de su ventana de congestión al inicio de la transmisión, ya que buscan alcanzar el máximo *throughput* posible rápidamente. Se observa un crecimiento mayor en BBR que en el *Slow Start* de NewReno y CUBIC. En estos dos últimos existe una leve diferencia en esta primera fase. La versión de CUBIC que incorpora *ns-3* sigue la implementación de Linux 3.14, la cual difiere ligeramente de la versión descrita en su RFC, aunque, sí que incorpora el *Hybrid Slow Start* o *HyStart* [14], activado por defecto mediante un atributo (*Attribute {HyStart}*). Este mecanismo emplea una heurística basada en

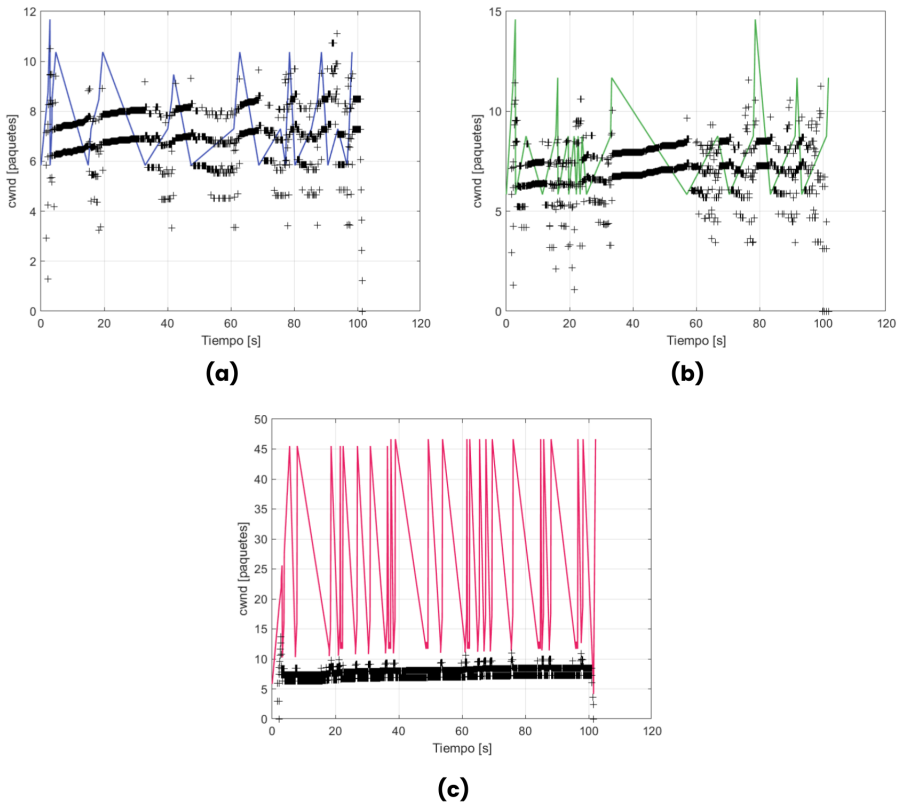


Figura 4.5: CWND y bytes en vuelo en QUIC para un publisher enviando paquetes a tasa constante cada 100 ms, en una red satelital y FER del 2 %. Aparecen NewReno (a), CUBIC (b) y BBR (c).

el RTT para salir antes del *Slow Start*, tratando de adelantarse a las primeras pérdidas. Terminada esta fase, en la Figura 4.5 se muestra como la ventana de congestión de BBR alcanza un valor casi cuatro veces mayor, para después comenzar a disminuir en busca del RTT mínimo. A partir de este punto, se observa como BBR mantiene una ventana mucho más alta, aumentando la tasa de envío periódicamente en busca de un mayor aprovechamiento de la capacidad de la red, sin penalizar el RTT, y reduciéndola en caso de que esto no sea posible. Destacar que CUBIC alcanza picos más altos que NewReno, al entrar en la región convexa de su función cúbica (estado *Max Probing*).

Con la Figura 4.7, pese a aumentar el tráfico con respecto a la Figura 4.1, se vuelve a comprobar que el ancho de banda no tiene casi influencia en los resultados. El aumento del RTT, sin embargo, sí afecta de forma notable al rendimiento, reduciendo el *goodput* hasta un 75 % en el peor caso (red satelital con un FER del 5 %) en los algoritmos basados en pérdidas. Se observa como BBR muestra un mejor comportamiento, manteniendo un *goodput* elevado en todas las situaciones pese a las pérdidas.

4.2.2. Dos o más flujos

Varios estudios [15] [16] [17] [18] han puesto de manifiesto ciertos problemas en el comportamiento de BBR a la hora de compartir el canal, tanto con otros flujos BBR (intra-protocolo), como con flujos que emplean otros algoritmos de congestión (inter-protocolo), en especial si estos están basados en pérdidas. Los principales inconvenientes encontrados son los siguientes:

- **Agresividad.** En la fase de *StartUp*, el aumento continuo de la tasa de envío crea una cola de larga

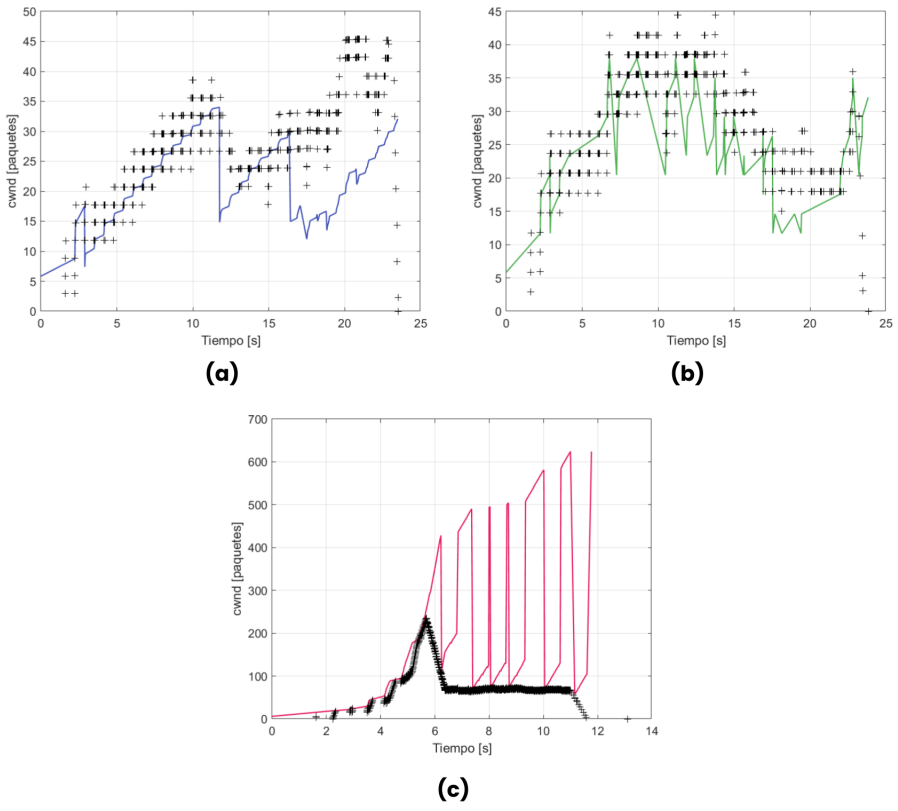


Figura 4.6: CWND y bytes en vuelo en QUIC para un publisher enviando paquetes a tasa constante cada 10 ms, en una red satelital y FER del 2 %. Aparecen NewReno (a), CUBIC (b) y BBR (c).

duración, lo que puede llegar a anular completamente los demás flujos. Además, en las pruebas en busca de mayor ancho de banda (fase *ProbeBW*), en escenarios donde múltiples flujos BBR comparten un mismo cuello de botella, cada flujo sobrestima la tasa de entrega, lo que resulta en la creación de una cola permanente de más de 1,5 BDP. Esto provoca una pérdida excesiva de paquetes cuando los tamaños de buffer son pequeños ($<$1 BDP).

- **RTT-*Fairness*.** Si varios flujos BBR con diferentes tiempos de propagación comparten un cuello de botella común, el flujo con mayor RTT utilizaría más ancho de banda en comparación con el resto, puesto que calcula un mayor BDP y, por tanto, inyecta más datos en la red. En la Figura 4.8 se ha simulado un ejemplo de este problema.

- ***Fairness* con el control de congestión basado en pérdidas.** El *throughput* que obtiene BBR y los algoritmos de control de congestión basados en pérdidas depende del tamaño del buffer del cuello de botella. En escenarios donde BBR comparte el canal con CUBIC, con tamaños de buffer pequeños, BBR puede acaparar casi el total del ancho de banda disponible ($\sim$90 %), mientras que con un tamaño de buffer grande ($\geq$1,5 BDP), CUBIC obtiene alrededor del 80 % del ancho de banda total.

En las Tablas 4.1 y 4.2 se muestra el *throughput* y el Índice de Jain, parámetro propuesto por Chiu y Jain [19] para medir lo equitativo del reparto de los recursos de red. El Índice de Jain se calcula usando la Ecuación 4.2, donde k es el número de flujos y x representa el *throughput* promedio de cada uno. Cuanto

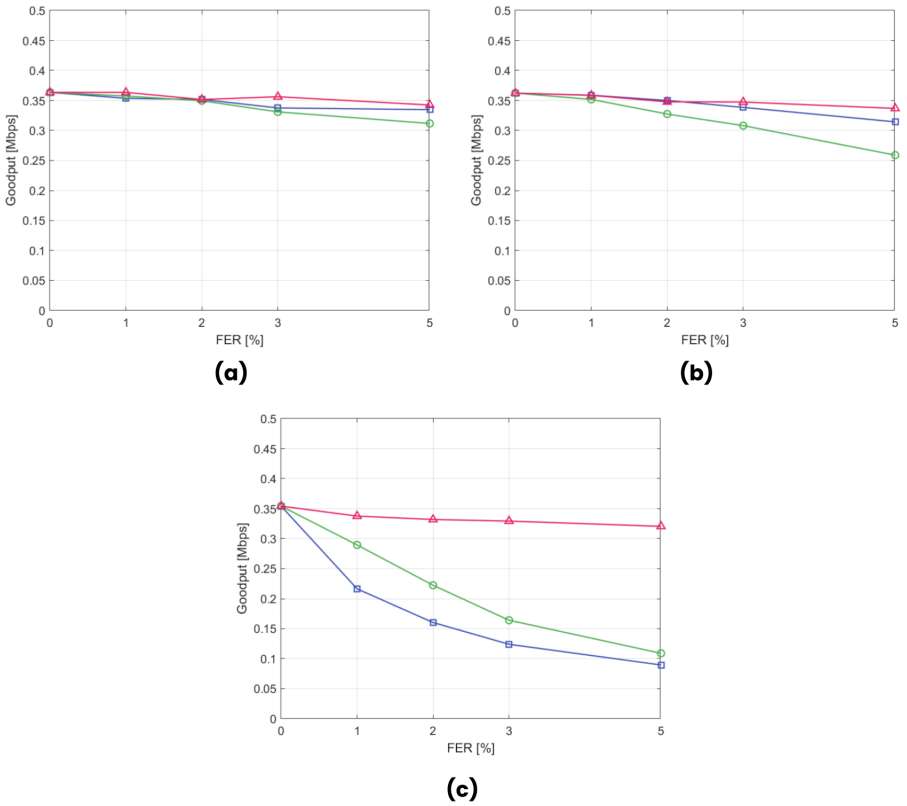


Figura 4.7: *Goodput* medio tras 100 simulaciones independientes en QUIC para varios valores de FER y tipos de red: WiFi (a), Celular (b) y Satelital (c)

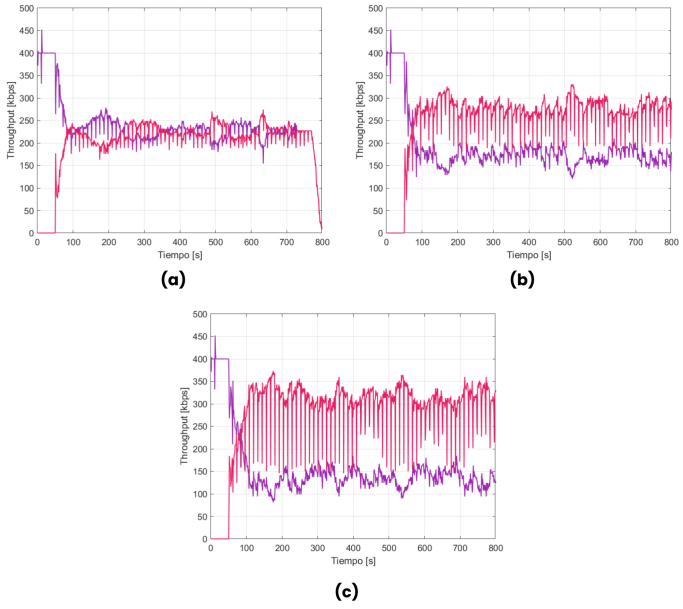


Figura 4.8: Ejemplo del problema RTT-fairness en BBR: dos flujos con un mismo RTT de 100 ms (a), con el segundo (rosa) de 200 ms (b) y aumentándolo a 300 ms (c).

Tabla 4.1: Resultados BBR.

		Throughput BBR	*Throughput* total	Índ. Jain
TCP	NewReno	4.13 %	84.12 %	0.55
	CUBIC	3.86 %	83.85 %	0.55
	BBR	46.63 %	90.19 %	1
QUIC	NewReno	2.55 %	76.82 %	0.53
	CUBIC	2.62 %	76.79 %	0.54
	BBR	46.78 %	94.91 %	1

Tabla 4.2: Resultados CUBIC.

	Throughput CUBIC	*Throughput* total	Índ. Jain
NewReno	79.86 %	89.41 %	0.62
BBR	79.99 %	83.14 %	0.54
CUBIC	45.05 %	88.1 %	1

más cerca esté el resultado de 1, mayor será el equilibrio entre los flujos que compiten por el ancho de banda.

$$F = \frac{(\sum_{i=1}^{k} x_i)^2}{k \cdot \sum_{i=1}^{k} x_i^2} \tag{4.2}$$

Para realizar las simulaciones, los publishers tienen tiempos de inicio distintos, con el objetivo de observar la convergencia de los flujos y la capacidad de respuesta del primero. Se fija el valor del RTT del enlace a 100 ms y el ancho de banda disponible a 500 kbps, para que los dos publishers, con tasas de envío de un paquete cada 10 ms (400 kbps), deban repartirse el ancho de banda. El tiempo de simulación se establece a 500 segundos, para permitir que los flujos abarquen una cantidad suficiente de RTTs y el tamaño de los buffers es suficientemente grande como para que no resulte una limitación. Puede verse en los resultados obtenidos que, como era de esperar, BBR y los algoritmos de congestión basados en pérdidas no logran alcanzar un reparto justo del ancho de banda. Nótese que en el caso de dos flujos, el peor índice de Jain posible es de 0.5. Por el contrario, dos flujos BBR alcanzan un reparto equitativo de los recursos de la red, como puede observarse en la Figura 4.9. Hay, sin embargo, una limitación en los resultados mostrados en ambas tablas. En las simulaciones con QUIC, BBR emplea la información extra que aportan los sockets QUIC, mientras que en los casos de NewReno y de CUBIC se utiliza QUIC en el modo heredado de TCP, tal como se describe en la Sección 2.4.2. Por ello, no es del todo justa la comparación de las implementaciones “incompletas” de estos dos algoritmos, y el rendimiento mostrado en la Tabla 4.1, debería ser, en principio, ligeramente superior al obtenido con TCP, tal como sucede con la implementación de BBR.

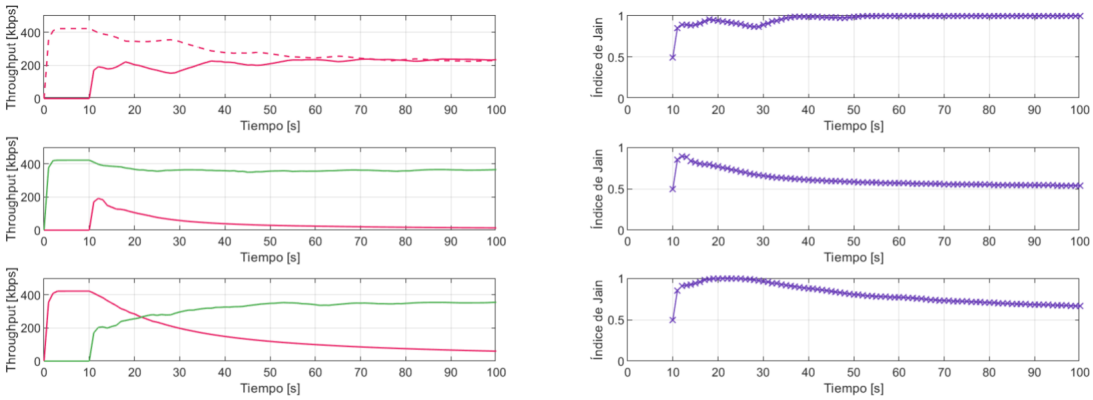


Figura 4.9: Flujos BBR (rosa) y CUBIC (verde) compartiendo el mismo enlace publishers-broker.

Tabla 4.3: Índice de Jain para múltiples flujos BBR y CUBIC, con diferentes tamaños de buffer.

	BufSize >> BDP	**BufSize = BDP**
4xCUBIC	0.98	0.99
4xBBR	0.87	0.95
4xCUBIC + 1xBBR	0.9	0.8
4xBBR + 1xCUBIC	0.57	0.63
1xCUBIC + 1xBBR	0.52	0.95
1xBBR + 1xCUBIC	0.60	0.99

De todas las medidas realizadas hasta el momento se concluye que BBR, pese a ofrecer un buen rendimiento en escenarios de un publisher y de dos publishers intra-protocolo, sufre a la hora de compartir el canal con algoritmos basados en pérdidas. Este comportamiento se acentúa si se introducen más publishers. En las Figuras 4.10 y 4.11 se compara el desempeño de BBR y CUBIC en diferentes situaciones, calculando el *throughput* de forma periódica (cada segundo) para cada uno de los publishers implicados. En el caso de cinco publishers, Figuras 4.10.b y 4.11.b, se representa la suma de los cuatro flujos que emplean el mismo algoritmo de control de congestión. Las medidas se han obtenido a partir de los paquetes recibidos en el broker, en un enlace publishers-broker sin pérdidas, con un RTT de 600 ms y donde el ancho de banda disponible se ha ajustado para que sea algo inferior a la tasa de envío total de todos los publishers. Esta tasa está limitada a nivel de aplicación a 400 Kbps. Además, para llevar a cabo las medidas de la Figura 4.11 se introduce un buffer intermedio de aproximadamente un BDP entre publisher y broker. Se observa como los resultados obtenidos son bastante mejores, obteniendo un índice de Jain superior en esta configuración, como se muestra en la Tabla 4.3.

A partir de estos resultados se puede concluir que cuando BBR y CUBIC coexisten, el equilibrio entre ellos depende claramente del tamaño del buffer. Para dos flujos en un enlace con un buffer de un tamaño cercano al BDP de la red, el reparto del ancho de banda es bastante equitativo, sin embargo, este equilibrio resulta frágil, ya que, con un tamaño de buffer suficiente, se produce un cambio drástico en el rendimiento.

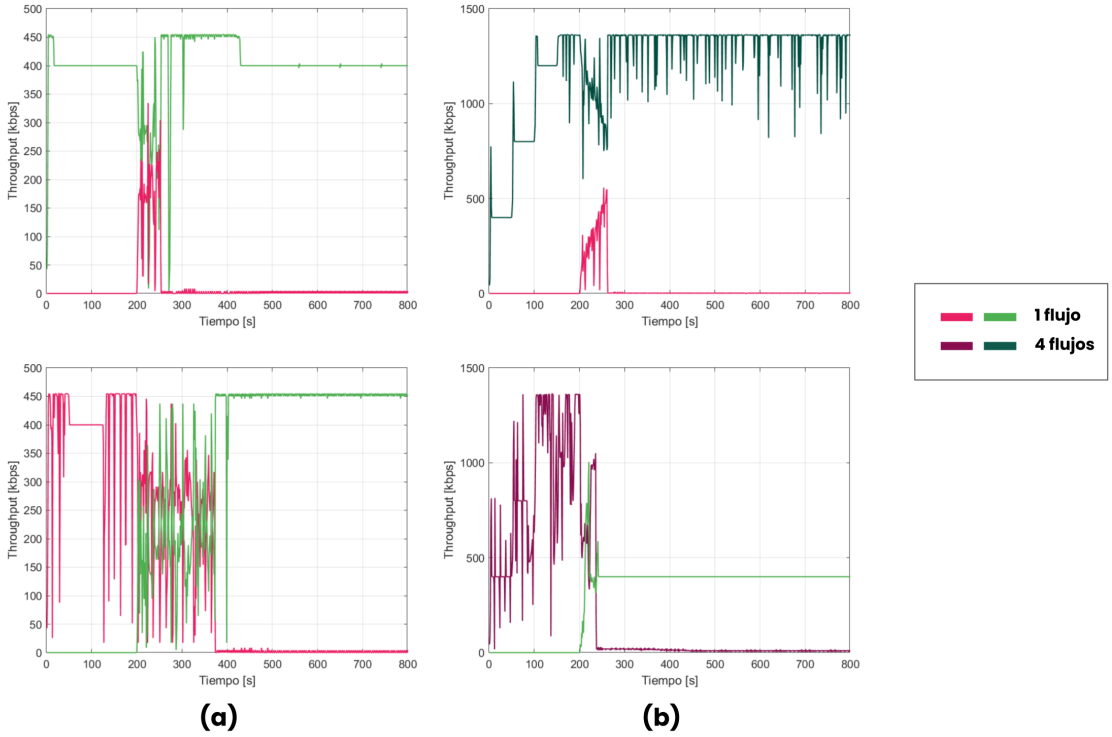


Figura 4.10: *Throughput* instantáneo de varios flujos BBR (rosa) y CUBIC (verde) compartiendo el mismo enlace publishers-broker, y donde están transmitiendo: (a) dos publishers y (b) cinco publishers.

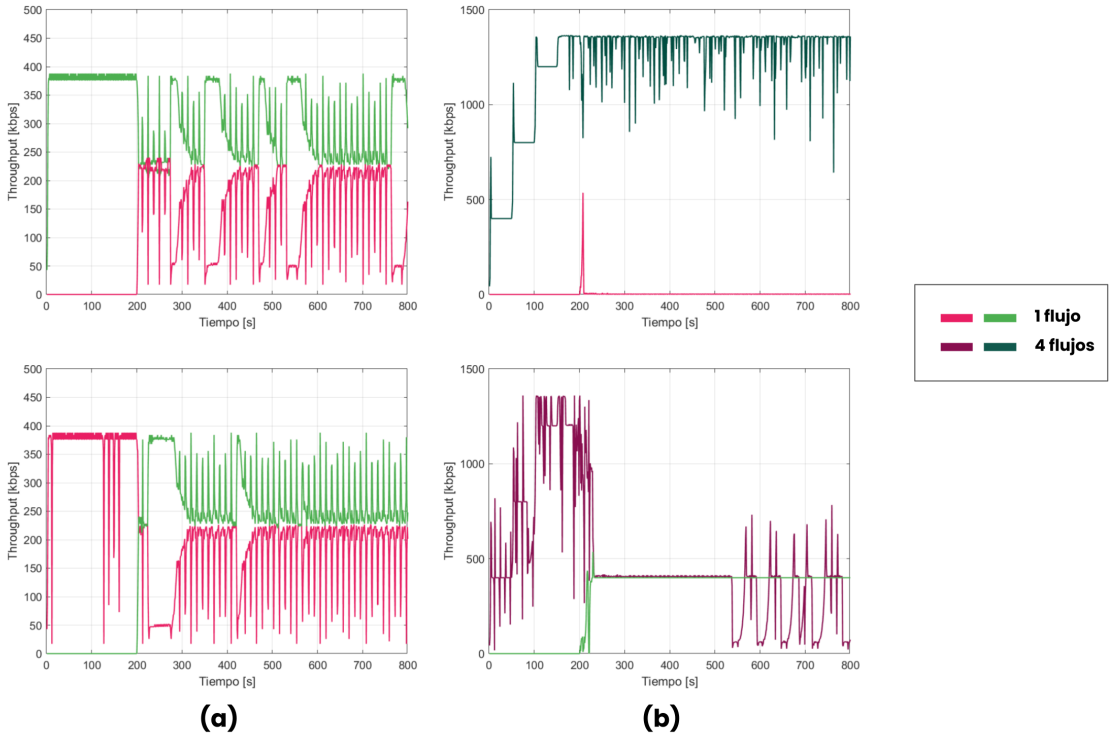


Figura 4.11: *Throughput* instantáneo de varios flujos BBR (rosa) y CUBIC (verde) compartiendo el mismo enlace publishers-broker con un buffer intermedio de aproximadamente un BDP, y donde están transmitiendo: (a) dos publishers y (b) cinco publishers.

El reparto justo entre ambos algoritmos de control de congestión se produce por la siguiente situación. Primero CUBIC aumenta de forma constante el tamaño de la ventana de congestión, hasta llenar completamente el buffer. BBR establece periódicamente el tamaño de la ventana de congestión en cuatro paquetes para medir el *RTprop*. Sin embargo, al llenarse el buffer, BBR mide un RTT superior al real. En consecuencia, BBR genera un exceso de cola, lo que sobrepasa al flujo CUBIC. BBR envía muchos más datos de los que el enlace puede admitir, traduciéndose en una pérdida excesiva de paquetes al desbordar el buffer. Cuando BBR ingresa de nuevo a la fase *ProbeRTT* drena el buffer por completo, y mide un *RTprop* más cercano al real. Al mismo tiempo, CUBIC comienza a aumentar la tasa de envío con un crecimiento agresivo de la ventana de congestión. El proceso anterior se repite de forma constante. Este equilibrio se rompe al aumentar el tamaño del buffer hasta el punto que el incremento de la tasa de envío de CUBIC deja de ser suficiente para generar pérdidas en el buffer o, por lo menos, no las suficientes para entrar en el proceso anterior. De esta forma, BBR mide un *RTprop* alto, pero no logra superar a CUBIC, que termina por acaparar un alto porcentaje del ancho de banda.

A diferencia de las simulaciones con dos flujos, los resultados obtenidos para cinco flujos, muestran un comportamiento menos predecible, sobre todo cuando un flujo BBR irrumpe en un enlace que ya cuenta con varios flujos CUBIC transmitiendo. En este caso, los flujos CUBIC eliminan completamente al nuevo en los dos escenarios vistos.

Existe un tercer escenario donde BBR logra obtener más ancho de banda que CUBIC. En la implementación de BBR, se establece el tamaño máximo de la ventana de congestión en 2 BDP, sin embargo, actualizar la ventana cada vez que se recibe un ACK puede suponer un cambio drástico, causando efectos adversos en el rendimiento. Por ello, BBR define un límite superior: *target_cwnd*. Se calcula este límite como $cwn\_dgain \cdot BtlBw \cdot RTprop$, donde *cwnd_gain* es una constante cuyo valor es 2. Esto resulta un problema cuando BBR compite con CUBIC en el mismo enlace con un buffer de menos de un BDP, ya que hace que BBR llene tanto el cuello de botella como el buffer. Esto da como resultado colas de hasta un BDP, más de lo que el enlace y el buffer pueden soportar, provocando que CUBIC no pueda aumentar su ventana.

Durante las pruebas también se observó, al igual que en [16], que el comportamiento de múltiples flujos BBR es inestable, reaccionando de forma diferente a pequeñas variaciones en las características de la red, algo que no ha ocurrido en el caso de CUBIC. Actualmente, Google está desarrollando BBRv2, presentado en [20], versión que introduce un conjunto de mejoras en el modelado de la red (objetivos de pérdida explícitos, Explicit Congestion Notification (ECN) y límites en los datos en vuelo) y en el *fairness* (cambios en la búsqueda de mayor ancho de banda y margen para permitir el crecimiento de nuevos flujos). Por todo ello, se espera que BBRv2 solucione o, por lo menos, mitigue los problemas que se han detectado en la primera versión.

5 | Conclusiones

En este proyecto se ha estudiado el desempeño del protocolo QUIC en escenarios IIoT frente al transporte tradicional TCP/TLS, empleando para ello una implementación nativa de QUIC en el simulador *ns-3*. Gracias a las posibilidades de emulación que ofrece esta plataforma, se ha realizado una extensa campaña de medidas, configurando diferentes tecnologías de red, caracterizadas por diferentes valores de ancho de banda y retardo. Se han desarrollado dos escenarios a simular, en los que un publisher envía paquetes de forma constante, y se ha observado como el protocolo QUIC supera claramente a TCP, especialmente para conexiones que tienen un RTT bajo y valores altos de FER. Como era de esperar, el impacto del ancho de banda es casi insignificante, ya que se considera el intercambio esporádico de paquetes cortos. También, se han comentado las mejoras que introduce QUIC, como son el establecimiento de conexión 0-RTT, la integración de TLS 1.3 y la multiplexación de streams. Por último, se comparan diferentes algoritmos de control de congestión y cómo se comportan a la hora de compartir un mismo enlace.

Ante la imposibilidad de usar una implementación de MQTT en *ns-3*, se ha optado por desarrollar aplicaciones que emulen en cierta medida el funcionamiento de este protocolo. En un futuro trabajo, con una implementación nativa de MQTT, sería posible llevar a cabo un mayor número de medidas, explorando nuevas funciones como, por ejemplo, la agrupación de varios mensajes MQTT en un solo paquete QUIC. En cuanto a los algoritmos de control de congestión, tanto la última versión de *ns-3* disponible (*ns-3.34*) como el módulo de QUIC, disponen únicamente de BBR en su primera versión, BBRv1. Existen nuevas versiones de BBR, como pueden ser BBR', BBRPlus o BBR+. Actualmente, Google ha lanzado la versión alpha de BBRv2, y continúa desarrollando la versión final. Como línea futura está la evaluación de las mejoras que aportan estas nuevas versiones de BBR, sobre todo, en situaciones donde se comparta el medio con algoritmos basados en pérdidas.

Bibliografía

[1] K. B. Andrew Banks Ed Briggs y R. Gupta. *MQTT Version 5.0*. OASIS Standard. OASIS, 2019.

[2] J. Iyengar y M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, 2021.

[3] M. Belshe, R. Peon y M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. 2015. DOI: 10.17487/RFC7540.

[4] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor, 2018.

[5] M. Thomson y S. Turner. *Using TLS to Secure QUIC*. RFC 9001. RFC Editor, 2021.

[6] J. Iyengar e I. Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. RFC Editor, 2021.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh y V. Jacobson. "BBR: Congestion-Based Congestion Control: Measuring Bottleneck Bandwidth and Round-Trip Propagation Time". En: *Queue* 14.5 (2016), págs. 20-53. DOI: 10.1145/3012426.3022184.

[8] T. Henderson, S. Floyd, A. Gurtov e Y. Nishida. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. http://www.rfc-editor.org/rfc/rfc6582.txt. RFC Editor, 2012.

[9] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert y R. Scheffenegger. *CUBIC for Fast Long-Distance Networks*. RFC 8312. RFC Editor, 2018.

[10] A. De Biasio, F. Chiariotti, M. Polese, A. Zanella y M. Zorzi. "A QUIC Implementation for Ns-3". En: *Proceedings of the 2019 Workshop on Ns-3*. WNS3 2019. Florence, Italy: Association for Computing Machinery, 2019, págs. 1-8. DOI: 10.1145/3321349.3321351.

[11] U. o. P. SIGNET Lab - DEI. *QUIC implementation for ns-3*. https://github.com/signetlabdei/quic.

[12] F. Fernández, M. Zverev, P. Garrido, J. R. Juárez, J. Bilbao y R. Agüero. "And QUIC meets IoT: performance assessment of MQTT over QUIC". En: *2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2020, págs. 1-6. DOI: 10.1109/WiMob50308.2020.9253384.

[13] F. Fernández, M. Zverev, P. Garrido, J. R. Juárez, J. Bilbao y R. Agüero. "Even Lower Latency in IIoT: Evaluation of QUIC in Industrial IoT Scenarios". En: *Sensors* 21.17 (2021). DOI: 10.3390/s21175737.

[14] S. Ha e I. Rhee. “Taming the elephants: New TCP slow start”. En: *Computer Networks* 55 (2011), págs. 2092-2110. DOI: 10.1016/j.comnet.2011.01.014.

[15] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer y G. Carle. “Towards a Deeper Understanding of TCP BBR Congestion Control”. En: *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. 2018, págs. 1-9. DOI: 10.23919/IFIPNetworking.2018.8696830.

[16] M. Hock, R. Bless y M. Zitterbart. “Experimental evaluation of BBR congestion control”. En: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. 2017, págs. 1-10. DOI: 10.1109/ICNP.2017.8117540.

[17] Y.-J. Song, G.-H. Kim, I. Mahmud, W.-K. Seo e Y.-Z. Cho. “Understanding of BBRv2: Evaluation and Comparison With BBRv1 Congestion Control Algorithm”. En: *IEEE Access* 9 (2021), págs. 37131-37145. DOI: 10.1109/ACCESS.2021.3061696.

[18] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian y A. Gandhi. “When to use and when not to use BBR: An empirical analysis and evaluation study”. En: 2019, págs. 130-136. DOI: 10.1145/3355369.3355579.

[19] D. M. Chiu y R. Jain. “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks”. En: *Computer Networks and ISDN Systems* 17 (1989), págs. 1-14. DOI: 10.1016/0169-7552%2889%2990019-6.

[20] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, P. J. Victor Vasiliev, Y. Seung, M. Mathis y V. Jacobson. *BBR v2 A Model-based Congestion Control*. https://datatracker.ietf.org/meeting/106/materials/slides-106-iccrg-update-on-bbrv2. Prague, Czech Republic, 2019.

A | Anexo código

```
import sys
import os
from multiprocessing import Pool

def start_simulation(bw, delay, fer, prot, maxNumPublishers, seedMax):
    for numPublishers in range(maxNumPublishers):
        for seed in range(seedMax):
            isdir = os.path.isdir(
                "Quic/bw%s_delay%s_fer%.2f_%s_pubs%d_run%d"
                    % (bw, delay, fer, prot, numPublishers+1, seed))
            if not isdir:
                os.mkdir("Quic/bw%s_delay%s_fer%.2f_%s_pubs%d_run%d"
                    % (bw, delay, fer, prot, numPublishers+1, seed))
            os.system(('./waf  --run "quicPointToPoint --access_bandwidth=%s --access_delay=%s --fer=%.2f --transport_prot=%s --prefix_file_name="Quic/bw%s_delay%s_fer%.2f_%s_pubs%d_run%d/quicPointToPoint" --num_flows=%d --max_packets=1000 --run=%d --interval=1000"')
                % (bw, delay, fer, prot, bw, delay, fer, prot, numPublishers+1, seed, numPublishers+1, seed))

if len(sys.argv) != 2:
    print("usage: ./Test [maxNumPublishers]")
    sys.exit(0)

maxNumPublishers = int(sys.argv[1])

# run
pool = Pool(processes=1)
pool.starmap(start_simulation, [
             # Red 1
             ("20Mbps", "12.5ms", 0, "TcpNewReno", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.01, "TcpNewReno", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.02, "TcpNewReno", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.03, "TcpNewReno", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.05, "TcpNewReno", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0, "TcpCubic", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.01, "TcpCubic", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.02, "TcpCubic", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.03, "TcpCubic", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0.05, "TcpCubic", maxNumPublishers, 100),
             ("20Mbps", "12.5ms", 0, "QuicBbr", maxNumPublishers, 100),
```

```
            ("20Mbps", "12.5ms", 0.01, "QuicBbr", maxNumPublishers, 100),
            ("20Mbps", "12.5ms", 0.02, "QuicBbr", maxNumPublishers, 100),
            ("20Mbps", "12.5ms", 0.03, "QuicBbr", maxNumPublishers, 100),
            ("20Mbps", "12.5ms", 0.05, "QuicBbr", maxNumPublishers, 100),
            # Red 2
            ("10Mbps", "50ms", 0, "TcpNewReno", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.01, "TcpNewReno", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.02, "TcpNewReno", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.03, "TcpNewReno", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.05, "TcpNewReno", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0, "TcpCubic", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.01, "TcpCubic", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.02, "TcpCubic", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.03, "TcpCubic", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.05, "TcpCubic", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0, "QuicBbr", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.01, "QuicBbr", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.02, "QuicBbr", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.03, "QuicBbr", maxNumPublishers, 100),
            ("10Mbps", "50ms", 0.05, "QuicBbr", maxNumPublishers, 100),
            # Red 3
            ("1.5Mbps", "300ms", 0, "TcpNewReno", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.01, "TcpNewReno", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.02, "TcpNewReno", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.03, "TcpNewReno", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.05, "TcpNewReno", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0, "TcpCubic", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.01, "TcpCubic", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.02, "TcpCubic", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.03, "TcpCubic", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.05, "TcpCubic", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0, "QuicBbr", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.01, "QuicBbr", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.02, "QuicBbr", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.03, "QuicBbr", maxNumPublishers, 100),
            ("1.5Mbps", "300ms", 0.05, "QuicBbr", maxNumPublishers, 100)
            ])
```

Listado A.1: Script Python.m

```
if (tracing)
{
  for (uint16_t i = 0; i < num_flows; i++)
  {
    auto n = publishers.Get (i);
    Time t = MilliSeconds(1001+10*i);
    Simulator::Schedule (t, &Traces, n->GetId(), // Tracing métricas de los
    publishers
          "PUB", ".txt");
  }

  Simulator::Schedule(NanoSeconds(1), &TraceTime, prefix_file_name); // Tracing de
    los tiempos de cada paquete
```

```
12
13    AsciiTraceHelper ascii;
14    Ptr<OutputStreamWrapper> ThputMedioStream = ascii.CreateFileStream((
        prefix_file_name + "-thputMedio.txt").c_str());
15    Simulator::Schedule(Seconds(stop_time), &resumen, ThputMedioStream); // Tracing al
        final de la simulación del throughput y paquetes perdidos
16  }
```

Listado A.2: Tracing métricas de la red.

```
1  static void
2  Traces(uint32_t id, std::string pathVersion, std::string finalPart)
3  {
4    AsciiTraceHelper asciiTraceHelper;
5
6    std::ostringstream pathCWND;
7    pathCWND << "/NodeList/" << id << "/$ns3::QuicL4Protocol/SocketList/0/
       QuicSocketBase/CongestionWindow";
8    NS_LOG_INFO("Matches cwnd " << Config::LookupMatches(pathCWND.str().c_str()).GetN
       ());
9    std::ostringstream fileCWND;
10   fileCWND << prefix_file_name << "-cwnd-" << pathVersion << id << "" << finalPart;
11   Ptr<OutputStreamWrapper> streamCWND = asciiTraceHelper.CreateFileStream (fileCWND.
       str ().c_str ());
12   Config::ConnectWithoutContext (pathCWND.str ().c_str (), MakeBoundCallback(&
       CwndTracer, streamCWND));
13
14   std::ostringstream pathRTT;
15   pathRTT << "/NodeList/" << id << "/$ns3::QuicL4Protocol/SocketList/0/
       QuicSocketBase/RTT";
16   std::ostringstream fileRTT;
17   fileRTT << prefix_file_name << "-rtt-" << pathVersion << id << "" << finalPart;
18   Ptr<OutputStreamWrapper> streamRTT = asciiTraceHelper.CreateFileStream (fileRTT.
       str ().c_str ());
19   Config::ConnectWithoutContext (pathRTT.str ().c_str (), MakeBoundCallback(&
       RttTracer, streamRTT));
20
21   std::ostringstream pathInFlight;
22   pathInFlight << "/NodeList/" << id << "/$ns3::QuicL4Protocol/SocketList/*/
       QuicSocketBase/BytesInFlight";
23   NS_LOG_INFO("Matches BytesInFlight " << Config::LookupMatches(pathInFlight.str().
       c_str()).GetN());
24   std::ostringstream fileInFlight;
25   fileInFlight << prefix_file_name << "-inFlight-" << pathVersion << id << "" <<
       finalPart;
26   Ptr<OutputStreamWrapper> streamInFlight = asciiTraceHelper.CreateFileStream (
       fileInFlight.str ().c_str ());
27   Config::ConnectWithoutContext (pathInFlight.str ().c_str (), MakeBoundCallback (&
       InFlightTracer, streamInFlight));
28 }
```

Listado A.3: Función *Traces*.

```
static void
RttTracer(Ptr<OutputStreamWrapper> stream, Time oldval, Time newval)
{
  if (firstRtt)
  {
    *stream->GetStream() << "0.0 " << oldval.GetSeconds() << std::endl;
    firstRtt = false;
  }
  *stream->GetStream() << Simulator::Now().GetSeconds() << " " << newval.GetSeconds
    () << std::endl;
}
```

Listado A.4: Ejemplo de función *trace sink*.

```
void Broker2::FwdPacket(Ptr<Packet> pkt)
{
  NS_LOG_FUNCTION(this);
  if (m_subsSocket != nullptr) {
  NS_LOG_INFO("--  QUIC Broker forwarding at " << Simulator::Now().GetSeconds());
  m_subsSocket->SendTo(pkt, m_lastUsedStream, m_subsAddr);
  ++m_lastUsedStream;
  if (m_lastUsedStream > m_numStreams)
  {
    m_lastUsedStream = 1;
}
```

Listado A.5: Función *FwdPacket*.

B Anexo figuras

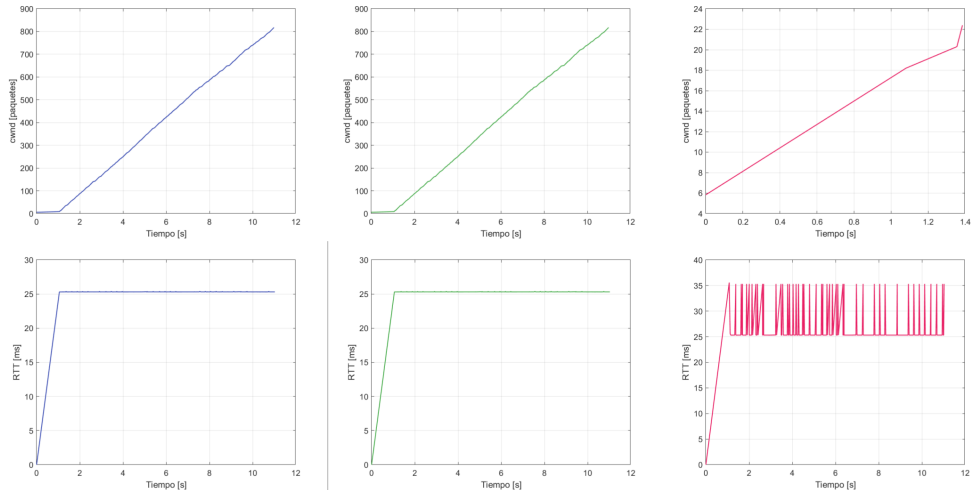


Figura B.1: QUIC en red WiFi con FER 0 % y tasa de envío de 10 ms.

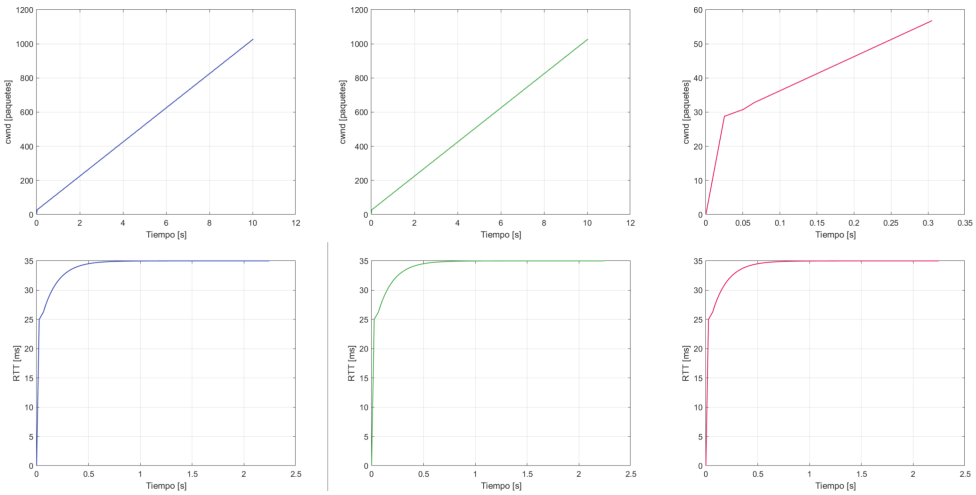


Figura B.2: TCP en red WiFi con FER 0 % y tasa de envío de 10 ms.

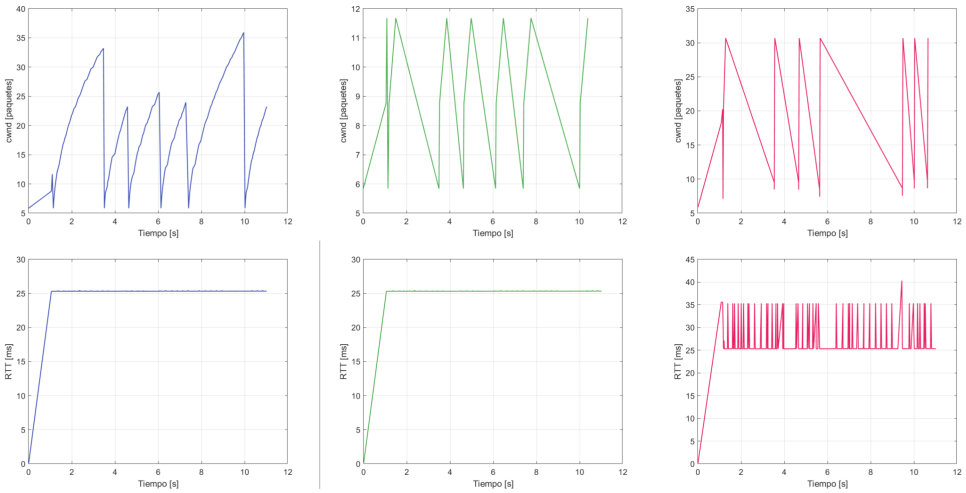


Figura B.3: QUIC en red WiFi con FER 1 % y tasa de envío de 10 ms.

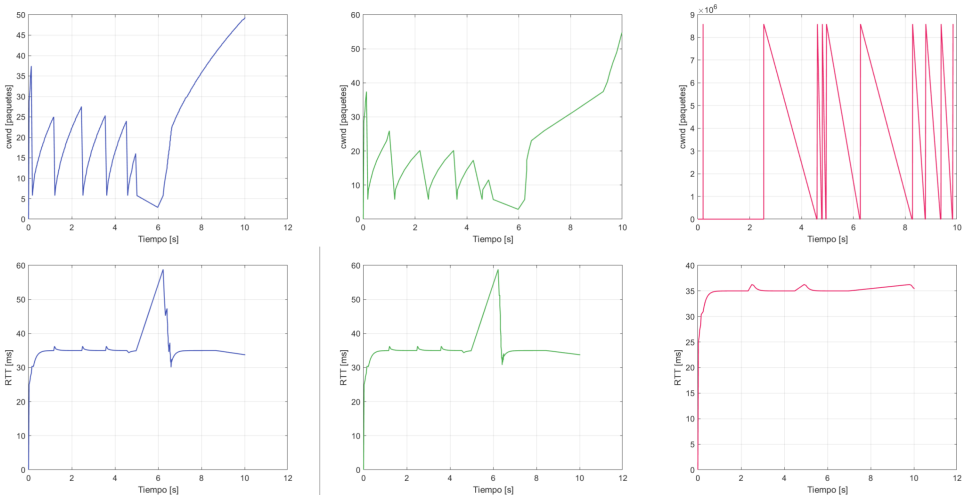


Figura B.4: TCP en red WiFi con FER 1 % y tasa de envío de 10 ms.

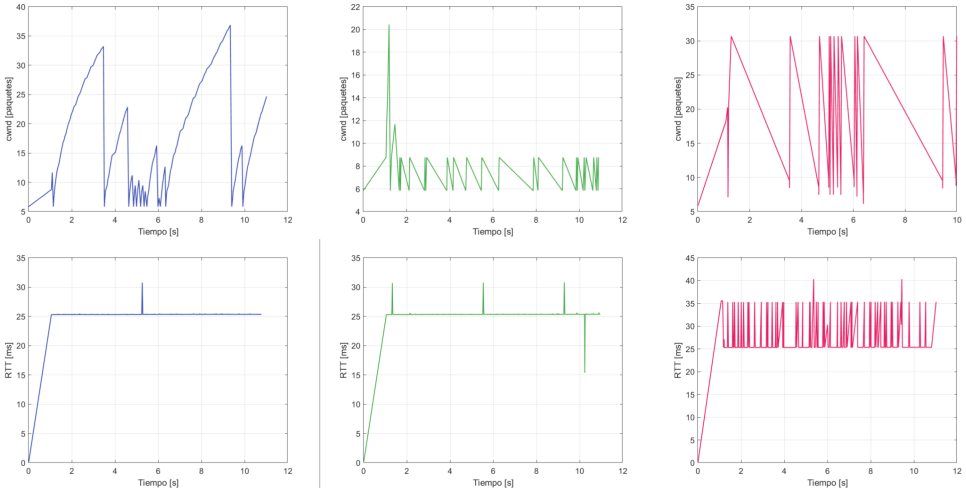


Figura B.5: QUIC en red WiFi con FER 2 % y tasa de envío de 10 ms.

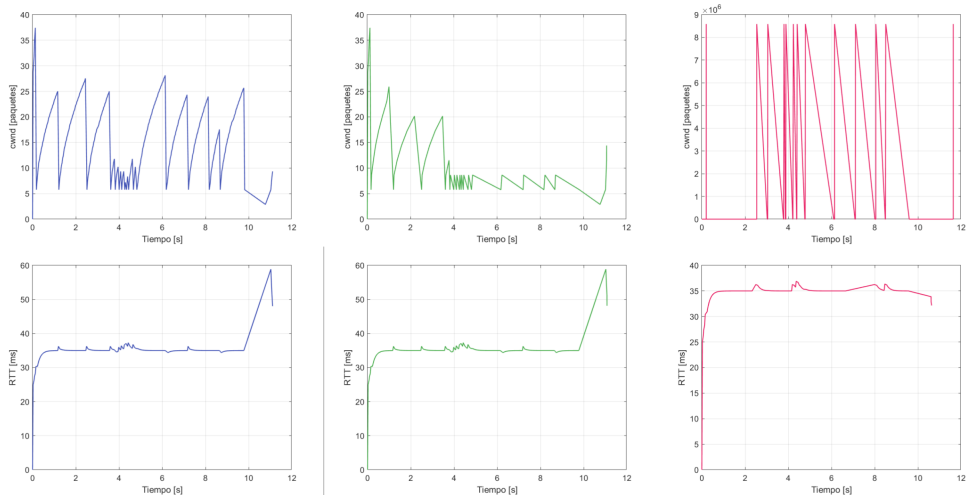


Figura B.6: TCP en red WiFi con FER 2 % y tasa de envío de 10 ms.

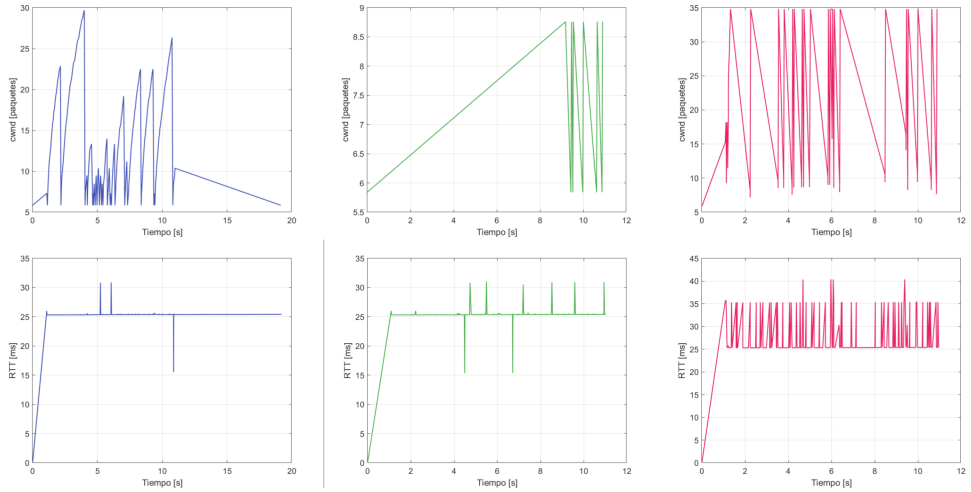


Figura B.7: QUIC en red WiFi con FER 3 % y tasa de envío de 10 ms.

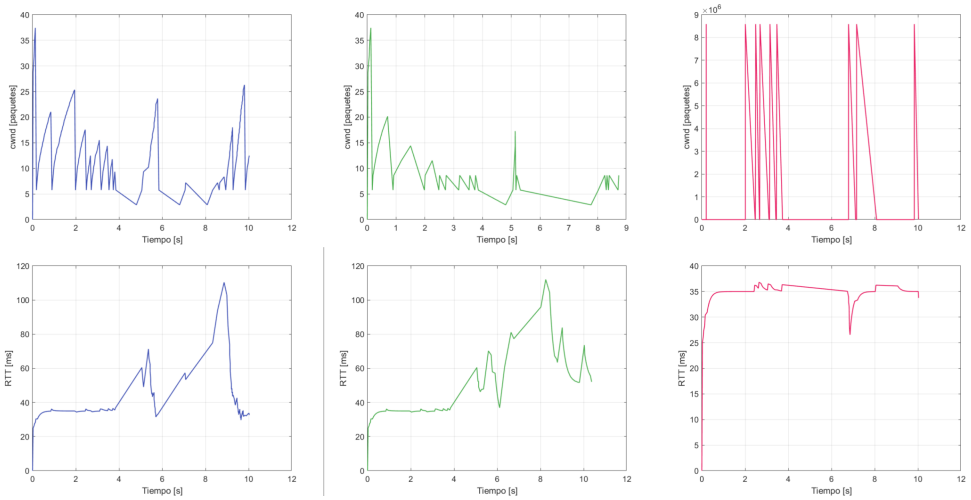


Figura B.8: TCP en red WiFi con FER 3 % y tasa de envío de 10 ms.

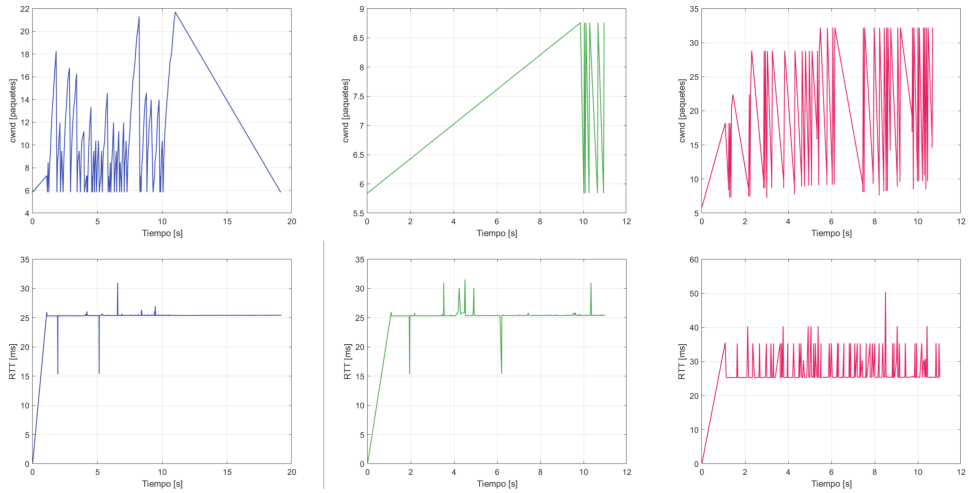


Figura B.9: QUIC en red WiFi con FER 5 % y tasa de envío de 10 ms.

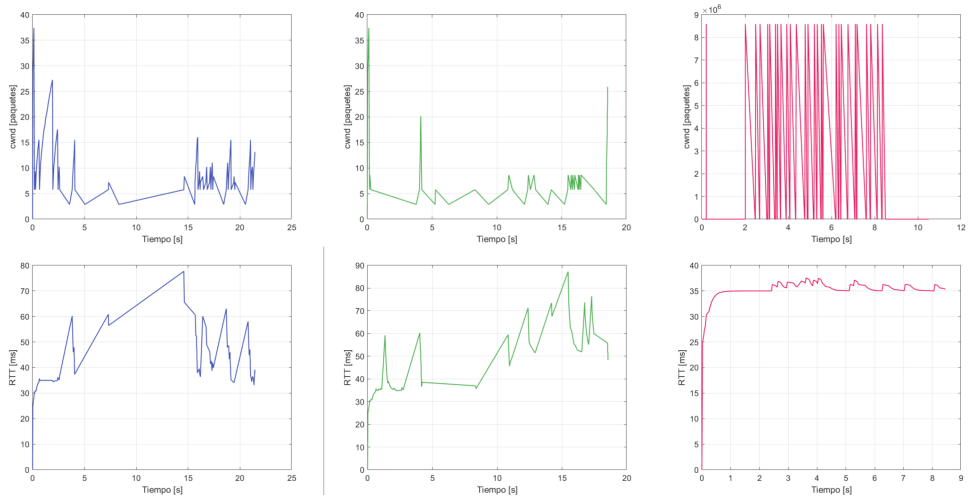


Figura B.10: TCP en red WiFi con FER 5 % y tasa de envío de 10 ms.

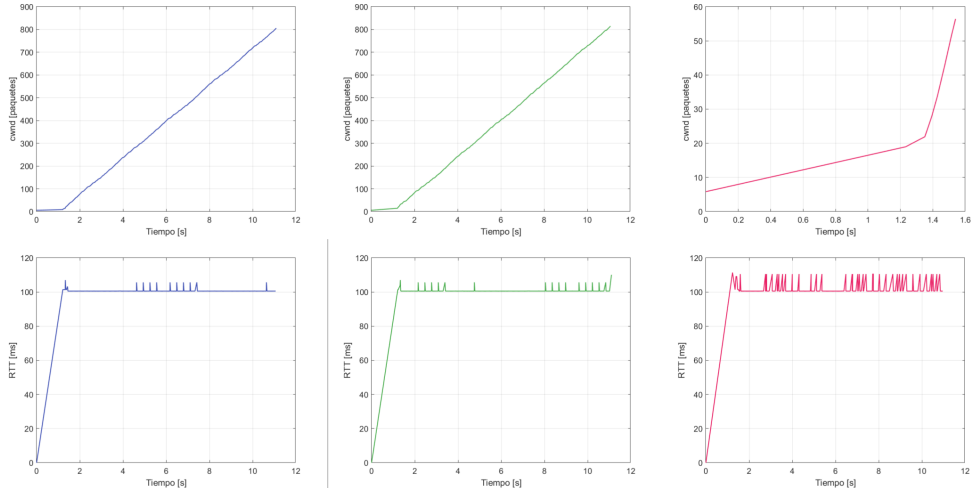


Figura B.11: QUIC en red celular con FER 0 % y tasa de envío de 10 ms.

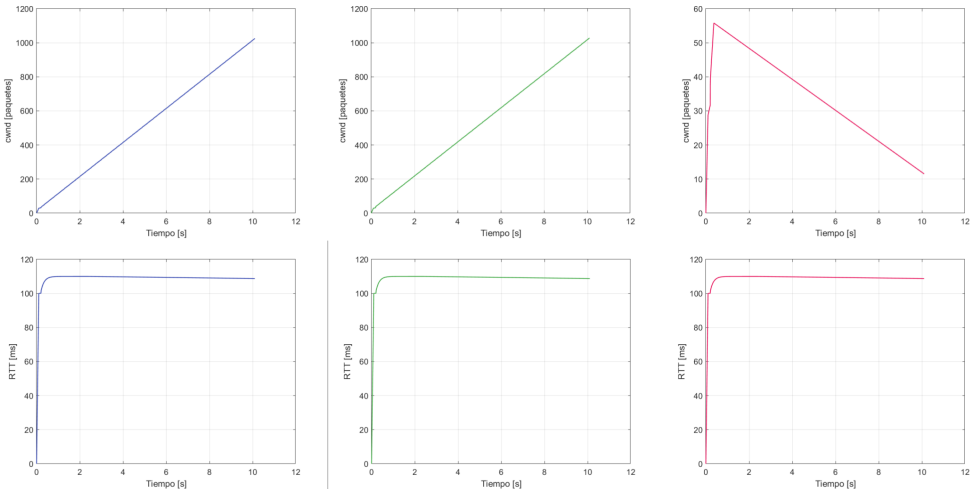


Figura B.12: TCP en red celular con FER 0 % y tasa de envío de 10 ms.

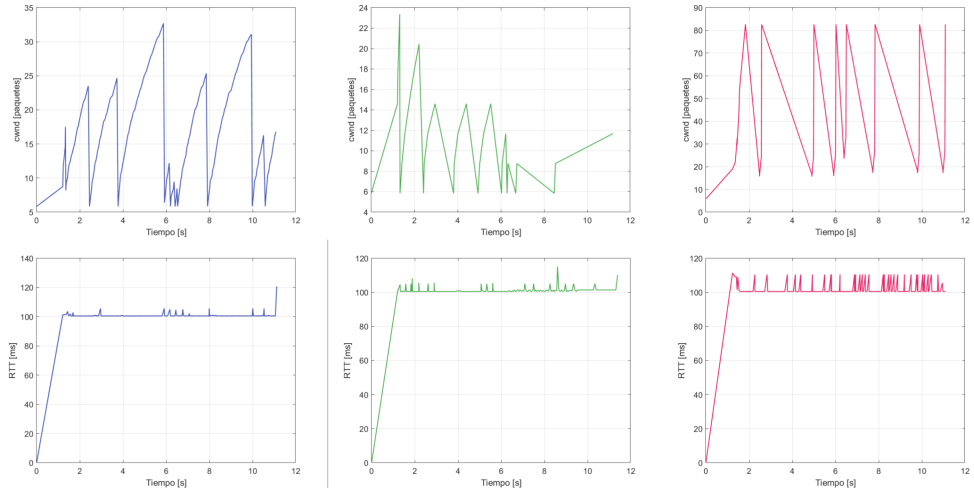


Figura B.13: QUIC en red celular con FER 1 % y tasa de envío de 10 ms.

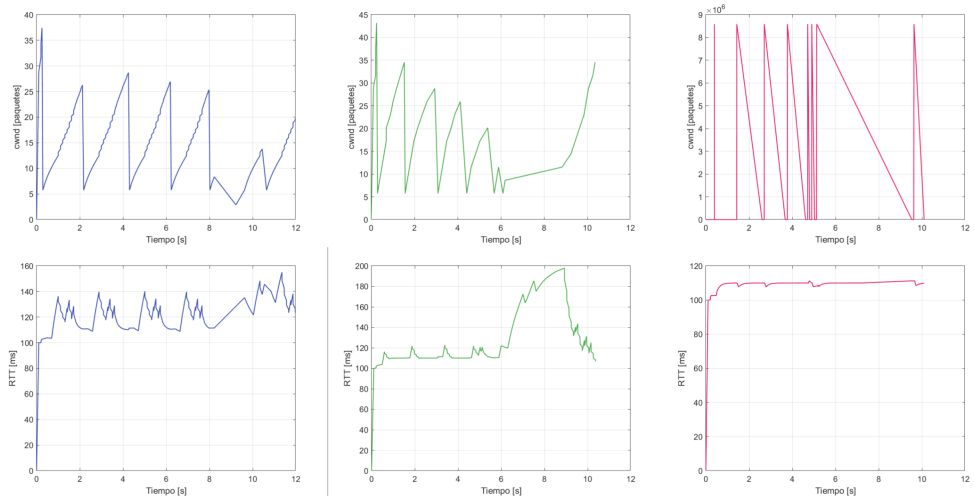


Figura B.14: TCP en red celular con FER 1 % y tasa de envío de 10 ms.

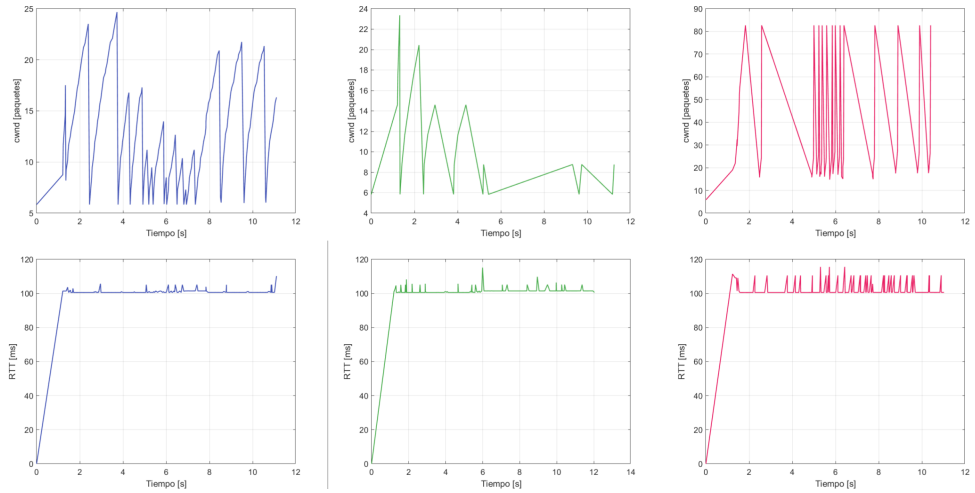


Figura B.15: QUIC en red celular con FER 2 % y tasa de envío de 10 ms.

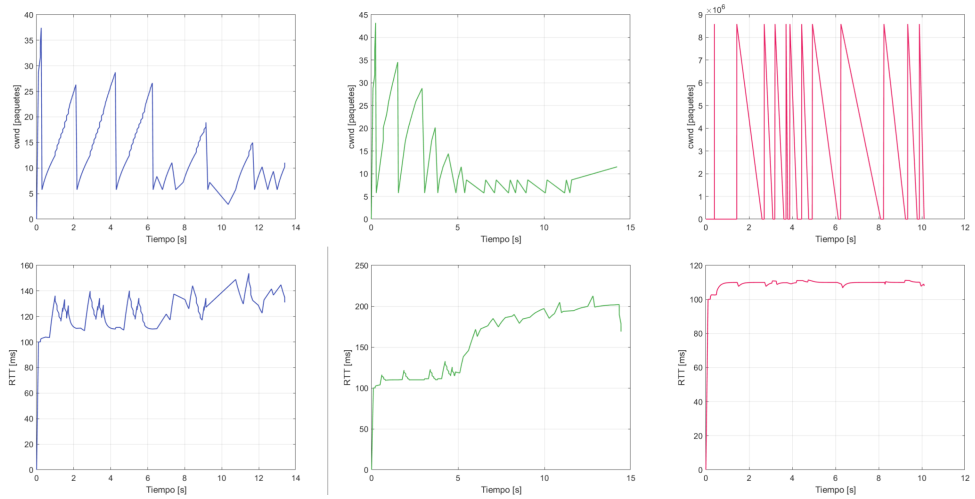


Figura B.16: TCP en red celular con FER 2 % y tasa de envío de 10 ms.

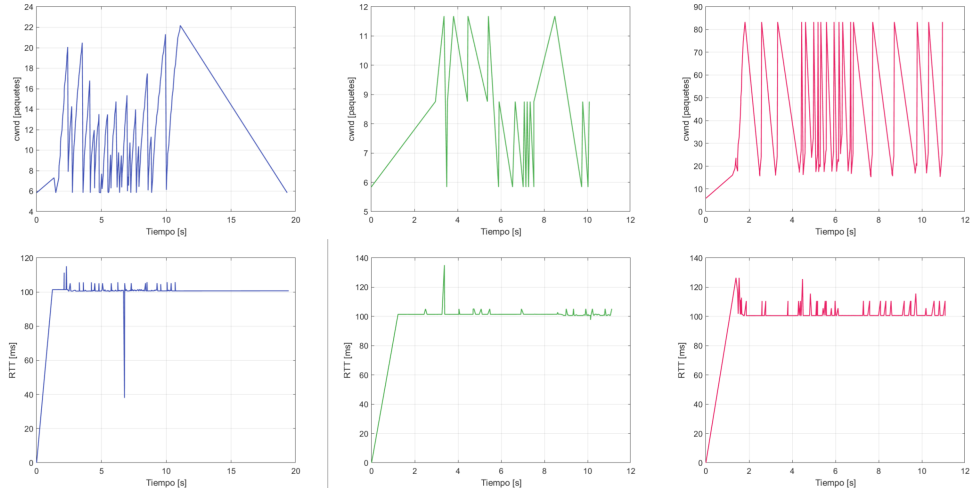


Figura B.17: QUIC en red celular con FER 3 % y tasa de envío de 10 ms.

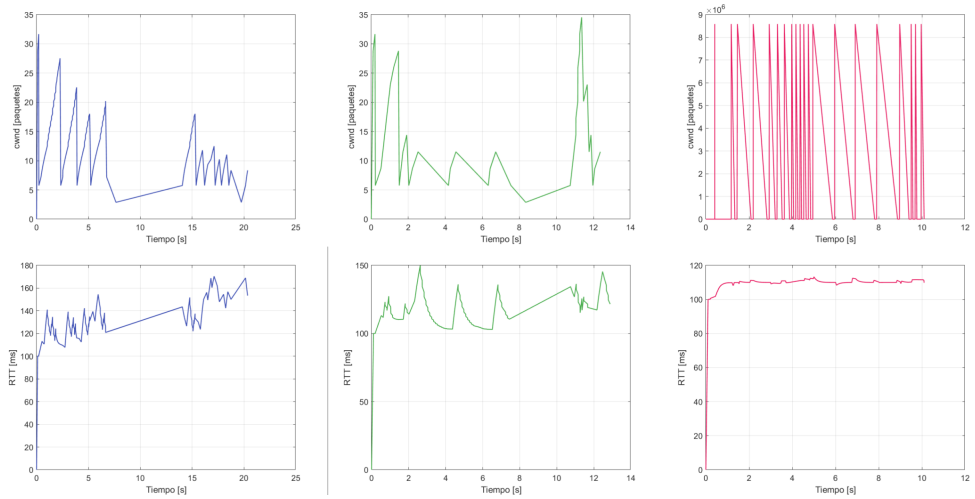


Figura B.18: TCP en red celular con FER 3 % y tasa de envío de 10 ms.

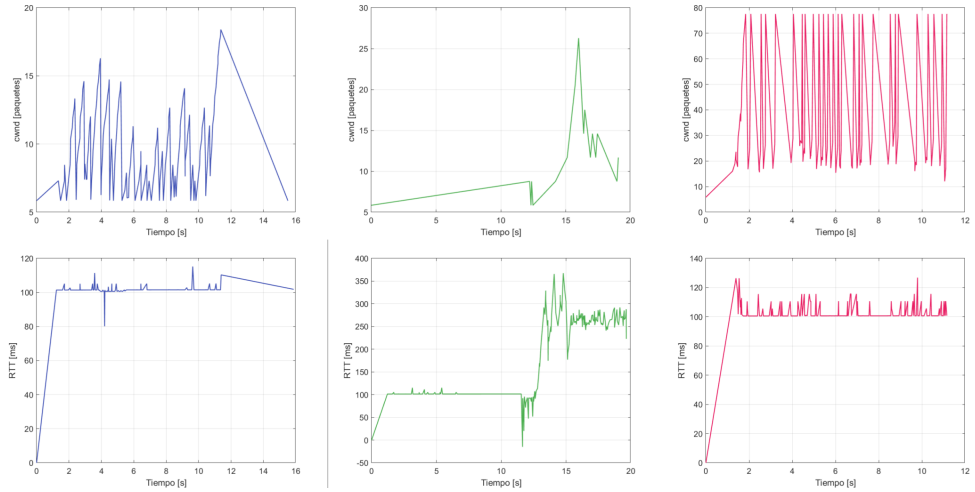


Figura B.19: QUIC en red celular con FER 5 % y tasa de envío de 10 ms.

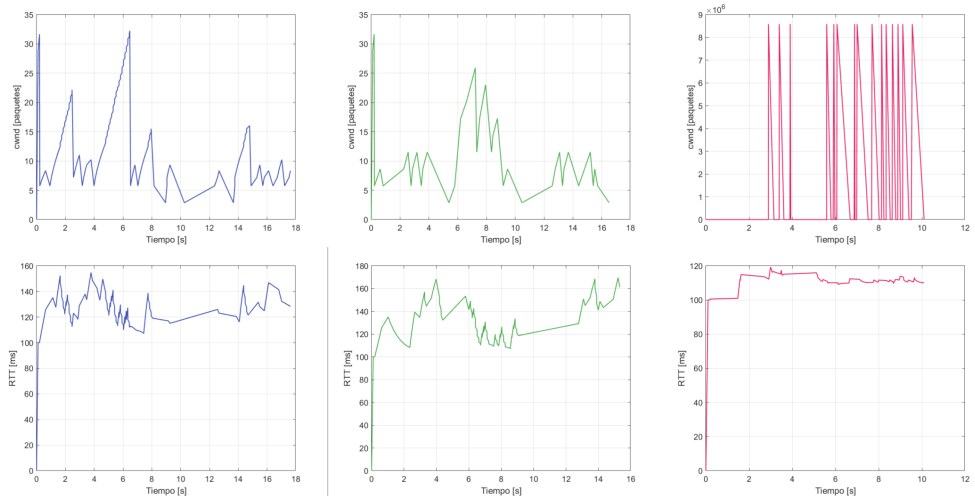


Figura B.20: TCP en red celular con FER 5 % y tasa de envío de 10 ms.

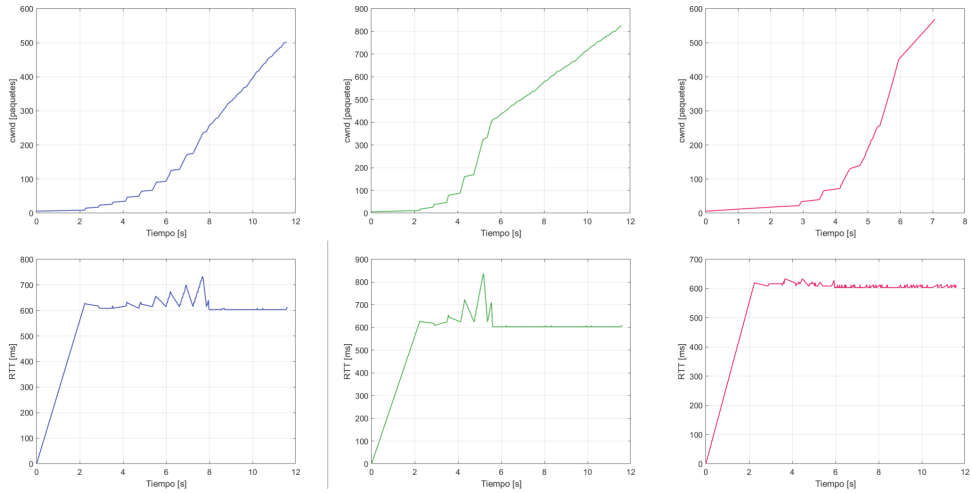


Figura B.21: QUIC en red satelital con FER 0 % y tasa de envío de 10 ms.

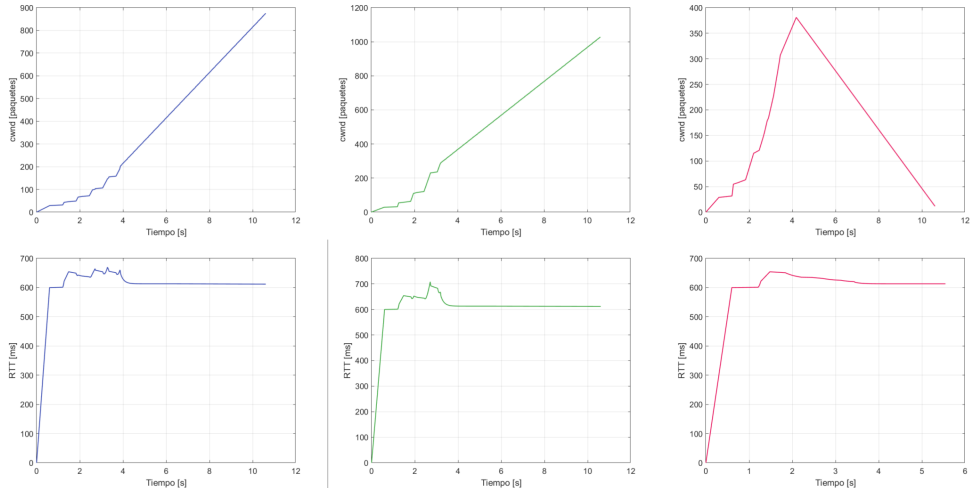


Figura B.22: TCP en red satelital con FER 0 % y tasa de envío de 10 ms.

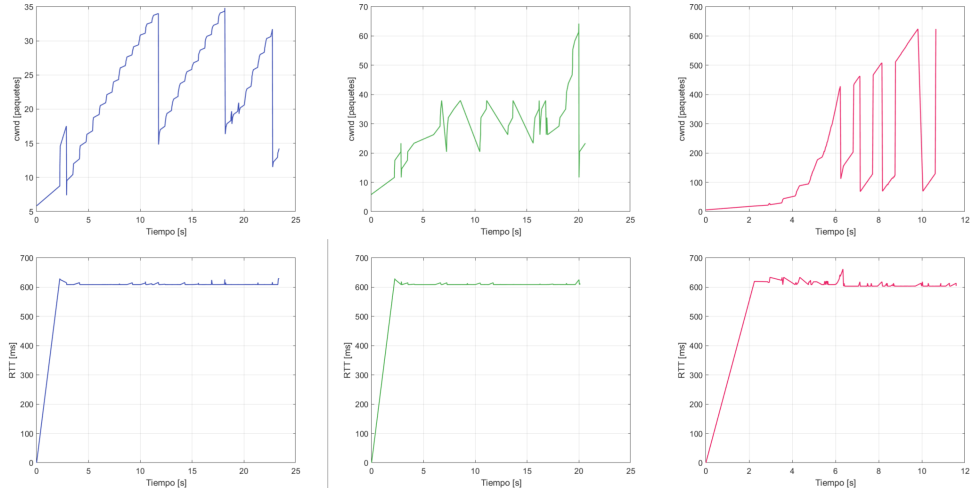


Figura B.23: QUIC en red satelital con FER 1 % y tasa de envío de 10 ms.

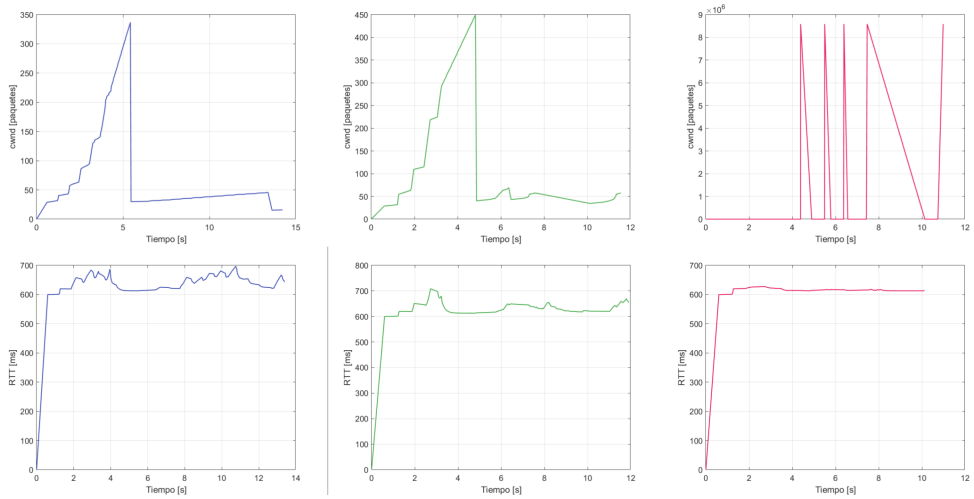


Figura B.24: TCP en red satelital con FER 1 % y tasa de envío de 10 ms.

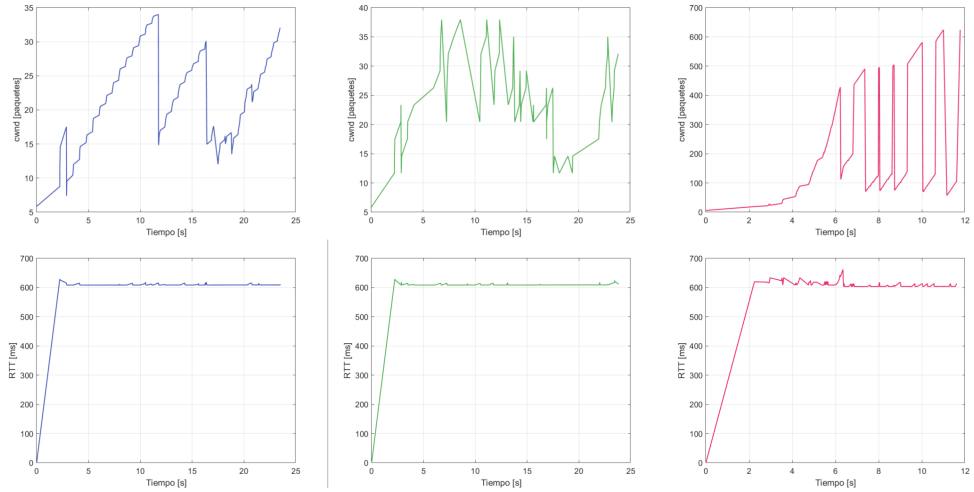


Figura B.25: QUIC en red satelital con FER 2 % y tasa de envío de 10 ms.

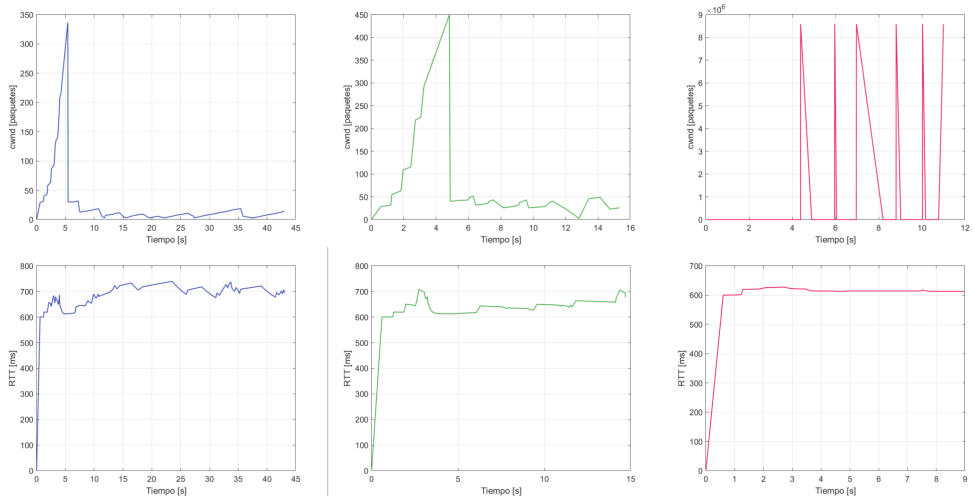


Figura B.26: TCP en red satelital con FER 2 % y tasa de envío de 10 ms.

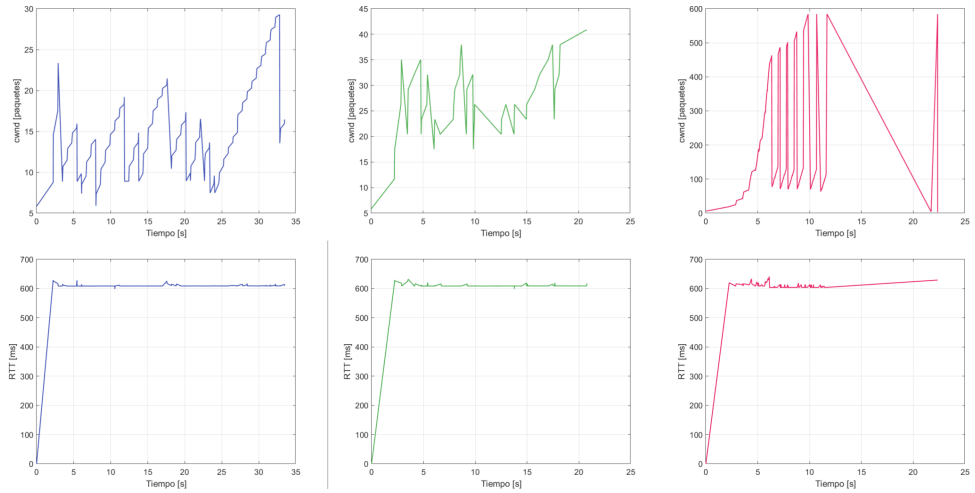


Figura B.27: QUIC en red satelital con FER 3 % y tasa de envío de 10 ms.

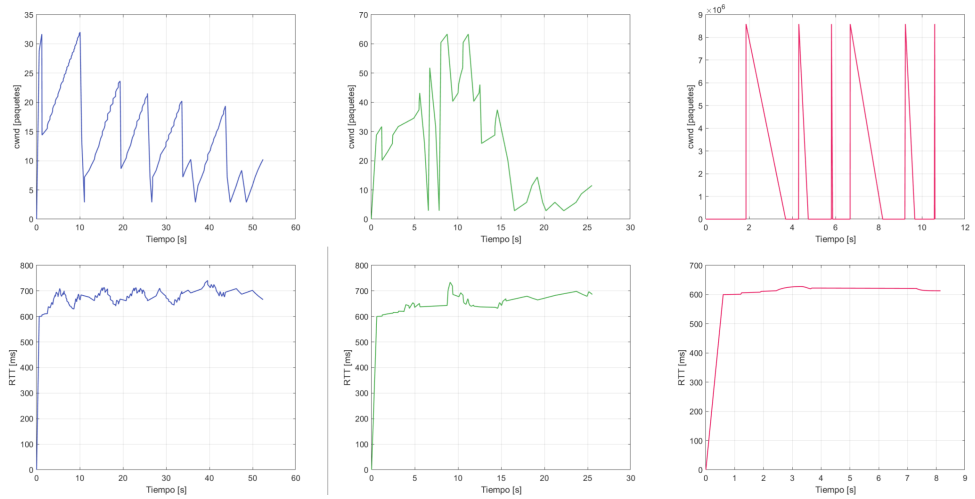


Figura B.28: TCP en red satelital con FER 3 % y tasa de envío de 10 ms.

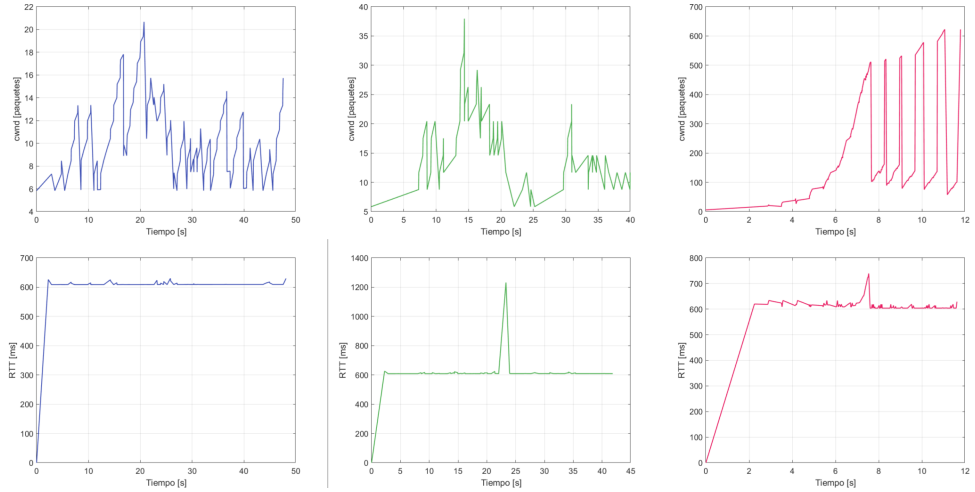


Figura B.29: QUIC en red satelital con FER 5 % y tasa de envío de 10 ms.

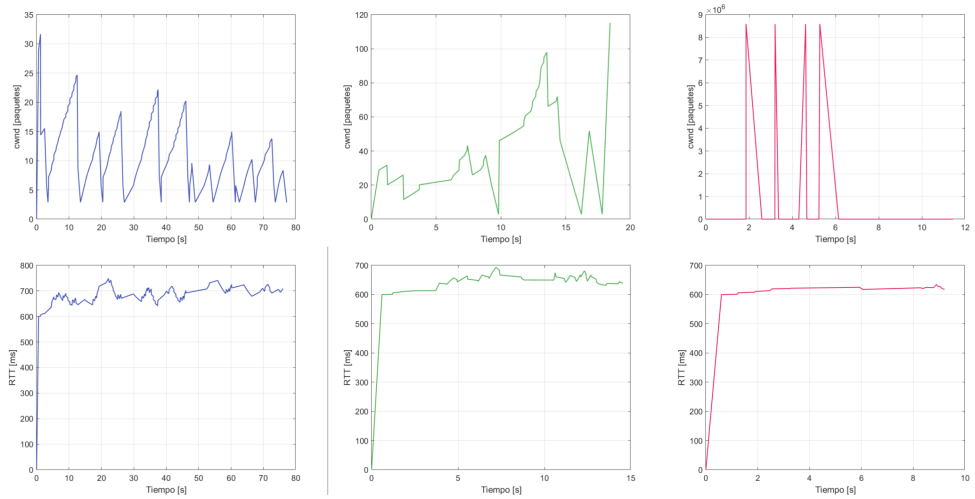


Figura B.30: TCP en red satelital con FER 5 % y tasa de envío de 10 ms.