ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Análisis del comportamiento de TCP Cubic sobre la plataforma ns-3
(Analysis of the behavior of TCP Cubic over ns-3 platform)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Alejandro Pérez Palacio

Septiembre - 2016

Índice

Índice de Figuras

Resumen ejecutivo

En la actualidad existen una gran variedad de protocolos que buscan mejorar el rendimiento de la red. Este proyecto se centra en los algoritmos de congestión, los cuales proponen un conjunto de técnicas para evitar situaciones de saturación en la red y recuperar los datos en caso de pérdida. Los algoritmos de congestión están especialmente asociados a uno de los principales protocolo de Internet, TCP, que aborda la capa de transporte en la pila de protocolos.

Las redes actuales cada vez disponen de un mayor ancho de banda, mientras que los servidores siguen estando ubicados, en muchas ocasiones, en lugares remotos. Esto hace que sea muy común redes donde el producto ancho de banda, retardo sea elevado. Como se estudiará durante este proyecto, la versión más extendida de TCP, New Reno, no muestra un comportamiento óptimo bajo estas condiciones. Varias alternativas han surgido para mejorar las prestaciones de TCP sobre este tipo de redes, destacando TCP CUBIC, que ha cobrado un papel relevante.

Se pone especial atención sobre los algoritmos de congestión que incorpora TCP CUBIC, implementado por defecto en el Kernel de Linux, que proporciona un incremento considerable del rendimiento sobre este tipo de redes. Para ello, se ha estudiado cuáles son sus principales características y qué beneficios proporciona frente otro tipo de protocolos.

Para analizar las prestaciones de CUBIC se ha hecho uso de la herramienta de simulación ns-3. Primero se ha integrado una implementación disponible de CUBIC, del Instituto Tecnológico de Atlanta. Los resultados muestran un análisis exhaustivo de CUBIC sobre diferentes topologías, tanto cableadas como inalámbricas. Los resultados ponen de manifiesto, en condiciones ideales, un incremento de más del 60 % en comparación con la versión estándar, de TCP New Reno.

Abstract

Nowadays there are many different protocols that try to improve the network's efficiency. This project focuses on the congestion control algorithms which propose a group of techniques to avoid network saturation situations and to recover data in case of loss. Congestion control algorithms are especially associated to one of the main internet protocols, TCP, which deals with the transport layer in the protocol stack.

Current networks have more and more bandwidth, while servers are many times still located in remote places. This makes it very common to find networks in which the large bandwidth delay product is high. As we will study during this project, the most extended version of TCP, New Reno, doesn't show an optimal behaviour under these conditions. Many alternatives have emerged to improve the benefits of TCP on this kind of networks, noting TCP CUBIC, that has achieve a relevant role.

Special emphasis is put on congestion control algorithms that include TCP CUBIC, introduced automatically in the Kernel de Linux, that provides a important increase of the efficiency on this kind of networks. To do so, we have studied which are the main features and the benefits it provides compared to other sort of protocols.

To analyze the benefits of CUBIC we have used the simulation tool Network Simulator 3. First of all, it has been integrated an available implementation of CUBIC of the Institute of Technology of Atlanta. Results show an exhaustive analysis of CUBIC about different topologies, wired and wireless. Results reveal under ideal conditions, an increase of more than 60 % comparing the standard version of TCP New Reno.

Agradecimientos

Quiero agradecerle a varias personas la ayuda que se me ha proporcionado para llevar acabo este proyecto. Por un lado, mi mayor agradecimiento va dirigido a Ramón Agüero y Pablo Garrido, las dos personas que bajo su atención, han hecho posible la realización de este proyecto. Sus conocimientos, pautas y consejos, me han servido de gran ayuda para poder desarrollar al máximo este proyecto. Además quiero dar gracias por la atención extra en esos momentos en los que tanto el tiempo, como la distancia, no estaban de nuestro lado. Sin olvidarme del resto del personal del laboratorio de telemática que han extendido su mano para ofrecerme ayuda cada vez que era necesario.

También quiero dar gracias a personas como Raquel, Sara y Carlota, las cuales han estado soportando mi carácter en los momentos más difíciles a lo largo de toda la carrera, haciendo que todo fuese más llevadero.

Finalmente, también dar las gracias a mis padres y abuelos que han realizado el esfuerzo durante estos 4 años para que las cosas fuesen de la mejor forma posible.

Gracias.

1 Introducción

En este primer capítulo se va a describir cuál es la finalidad de este proyecto. En primer lugar, se realizará una breve descripción del principal protocolo utilizado en Internet, TCP. En segundo lugar, se propondrán los objetivos de esta memoria para finalmente desencadenar la estructura que van a tener los siguientes capítulos.

1.1 Evolución del protocolo TCP

El gran avance tecnológico ha supuesto sin duda un gran desarrollo en las redes de comunicación. En el mundo tenemos desde redes de área local, hasta redes que cruzan de extremo a extremo el planeta permitiendo la conexión entre usuarios.

El aumento de los usuarios está fuertemente ligado al despliegue de redes más extensas y con mejores características, tanto a nivel físico, como a nivel de gestión, almacenamiento y procesado de la información. Como consecuencia de esto, obtener mejores rendimientos ante estas nuevas topologías, conlleva a una gestión óptima de la información. Los protocolos organizan la información de forma coherente para permitir la conexión entre extremos. Uno de los principales y más usado a nivel de transporte, es TCP. Este permite principalmente realizar una conexión fiable extremo a extremo, ofreciendo un acuse de recibo de todos los paquetes que se puedan generar y permitiendo además que estos se reciban en el mismo orden en que se transmitieron originalmente.

Por otro lado, otra de las funciones que propone TCP es el control de congestión. Sobre esta característica, y no más importante que las demás, se van a desencadenar los capítulos posteriores. Existen diferentes métodos para evitar la saturación o congestión de la red, controlando la tasa de transmisión por parte del nodo origen y que adaptarse a las distintas redes actuales para obtener mejores rendimientos.

1.2 Objetivos

Tras haber descrito el planteamiento del problema, se presenta seguidamente los principales objetivos del proyecto. CUBIC es una variante de TCP que permite ajustarse de una forma más eficiente frente a redes de alta velocidad y latencia. Como ayuda para realizar el análisis de este protocolo, se ha utilizado la herramienta de simulación Network Simulator 3 (ns-3). El simulador ns-3, basado en el lenguaje C++ y de código abierto, proporciona las herramientas necesarias para realizar las medidas correspondientes y evitar desplegar una red física real para obtener los resultados.

Como principal objetivo, este proyecto tiene como finalidad ver cuáles son las principales características del protocolo implementado por defecto en el Kernel de Linux, TCP CUBIC. Así mismo también, se realizará una comparativa con otro protocolos más estándar, como es New Reno, o protocolos que ofrecen también buenos resultados ante el mismo tipo de redes, como es Westwood. Por ello los protocolos se evaluarán sobre diferentes escenarios, modificando los parámetros más importantes para ver tanto diferencias como similitudes entre ellos.

El protocolo TCP CUBIC no está implementado por defecto en la arquitectura del simulador ns-3, de manera que otro de los objetivos de este proyecto es permitir que el Grupo de Ingeniería Telemática tenga como referencia un algoritmo de congestión más reciente para la realización de futuros estudios. Para ello, se aprovechará la flexibilidad que ofrece el simulador y se incluirá TCP CUBIC en su arquitectura como un módulo más.

1.3 Estructura del proyecto

En esta sección se va a realizar un desglose de cómo está estructurada esta memoria. Atendiendo una composición por capítulos, se encontrará:

• **Capítulo 2**: este apartado se dividirá en dos partes. En primer lugar, se realizará un análisis de los algoritmos más básicos, sobre los cuales otros se basan como punto de partida. En segundo lugar, la intención es describir las principales propiedades de la implementación TCP en la que centra este proyecto, TCP CUBIC. Para ello, se va a estudiar cómo se comporta la ventana de congestión y qué ventajas proporciona ante los diferentes tipos de redes.

• **Capítulo 3**: este capítulo hará una introducción a la estructura del simulador ns-3, centrando la atención en aquellos módulos relacionados con TCP. Posteriormente, se analizará la implementación del CUBIC en el simulador y se finaliza con un estudio de los valores de ns-3 que afectan en gran medida a esta implementación.

• **Capítulo 4**: una vez realizada la correspondiente parte teórica, este cuarto capítulo resume los principales escenarios y valores a medir para posteriormente realizar un análisis de los resultados obtenidos tras varias simulaciones, justificando así lo descrito en capítulos anteriores.

• **Capítulo 5**: en el quinto y último capítulo de esta memoria, se recogerá un breve análisis del protocolo TCP CUBIC tras haberlo estudiado tanto desde un punto de vista teórico en el capítulo 1 como desde el punto de vista práctico, en el capítulo 4.

2

TCP

Este amplio capítulo resume las principales versiones que existen del protocolo TCP. Posteriormente, se describirá en profundidad las características de TCP CUBIC para a continuación entender su comportamiento ante diferentes escenarios.

2.1 Evolución de las implementaciones TCP

El protocolo TCP es uno de los más utilizados en Internet. Entre sus funciones más importantes está realizar una monitorización de la red y responder con un algoritmo de control de congestión para evitar saturaciones. La congestión se produce cuando la tasa de transmisión de datos es más elevada que lo que soporta la red, desbordando los buffer y provocando pérdida de paquetes.

Uno de los factores más importantes relacionados con el control de congestión es la ventana de congestión, Congestion Window (CWND). Dependiendo del valor que tome la ventana de congestión, el emisor podrá estimar el número de paquetes que puede enviar sin que se produzca congestión. Ante esta característica, se han desarrollado una variedad de implementaciones que tratan este problema de diferente forma, permitiendo que el protocolo TCP se adapte a todo tipo de redes. Algunos se han diferenciado en mínimas expresiones, mientras que han supuesto cambios drásticos en la gestión de la ventana. En las siguientes secciones se va a discutir las propuestas más importantes que ponen solución a este obstáculo.

2.1.1. TCP Tahoe

TCP Tahoe, [1], fue una de las primeras versiones en implementar una solución ante el control de la congestión, [2]. Nació en 1988, de la mano de Van Jacobson, [3]. Cuando un paquete se envía, el receptor ha de contestar mediante un reconocimiento (ACK) para

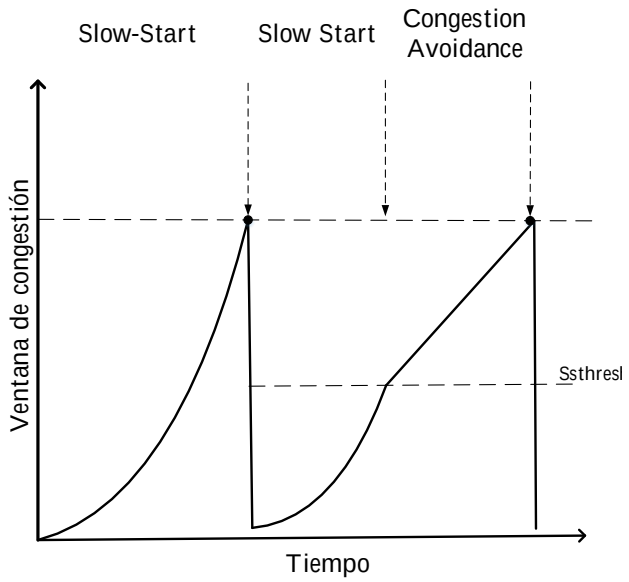


Figura 2.1: Evolución ventana de Congestión de TCP Tahoe

informar al emisor de que el paquete ha llegado correctamente. Un paquete puede llegar a destiempo, perderse, o producirse un *timeout*. Un *timeout* se produce cuando se excede un tiempo estimado para la recepción de un ACK. Cuando se da uno de los tres casos anteriores, el algoritmo de control de congestión busca una solución ante esta pérdida e inicia la retransmisión de los paquetes perdidos.

En referencia el *timeout*, el principal problema es el cálculo del valor adecuado para iniciar una retransmisión, Retransmission timeout (RTO), y así no producir retransmisiones innecesarias (subestimación del RTO) ni tampoco esperar en exceso por parte de la fuente. Para calcular el RTO, hay que tener en cuenta el tiempo que tarda un paquete en llegar al receptor y ser recibido su respectivo ACK, lo que es conocido como Round-Trip Time (RTT). En un hipotético caso donde la ocupación de los nodos intermedios fuese constante, el RTT permanecería en constante valor, sin embargo, en la realidad variará dependiendo de las condiciones actuales de la red.

TCP Tahoe propone para el cálculo del RTO dos algoritmos llamados *round-trip variance estimation* y *exponential retransmit timer backoff*.

round-trip variance estimation, evita una sobreestimación del cálculo del RTO. Cada vez que se envíe un paquete, se estima el valor del RTT, lo que es conocido como $SRTT$, [4]. $SRTT$ es calculado de la forma: $SRTT(i+1) = \alpha \cdot SRTT(i) + (1-\alpha) \cdot S(i)$. Una vez se haya estimado este valor, se realiza el cálculo del RTO de la forma: $RTO(i) = \beta \cdot SRTT$. Donde β es una constante que varía en un intervalo de 1.3 a 2. El valor del RTO siempre será superior al RTT, ya que el tiempo mínimo que puede pasar para detectar una pérdida en una situación ideal es el RTT. El límite superior para evitar la sobreestimación es:

$RTO_{lim} = SRTT + 4 \cdot \beta \cdot SRTT$.

El segundo algoritmo, *exponential retransmit timer backoff*, evita la subestimación. Como solución, se dobla el valor del RTO en cada retransmisión. Cuando la red vuelve al estado normal, el valor delRTO se vuelve a calcular como se ha explicado en el caso *round-trip variance estimation.*

Tras haber realizado el cálculo del RTO adecuado, el algoritmo *Fast Retransmit*,[2], es el encargado de iniciar la retransmisión de los datos en el caso de producirse un *timeout* o se hayan recibido ACK's duplicados. TCP Tahoe incluye el modo *Fast Retransmit* como un algoritmo que se encarga de reducir el umbral de congestión (*slow start threshold* o *ssthresh*) a la mitad, como se ve en la Figura 2.1. Además actualiza el valor de la ventana de congestión a 1MSS, donde el Maximum Segment Size (MSS) es la cantidad máxima de bytes que un receptor puede recibir sin tener que fragmentar el segmento, por eso en la gráfica el valor de la ventana no cae hasta el eje de las abscisas.

Los otros dos algoritmos propuestos por TCP Tahoe son: *Slow-Start* y *Congestion Avoidance*, [2], también representados en la Figura 2.1. *Slow-Start* permite enviar el doble de la información enviada anteriormente por cada ACK recibido, comenzando en 1MSS. Sin embargo, una vez pasado el umbral *ssthresh*, se pasa al segundo modo, *Congestion Avoidance*, proponiendo una evolución más lenta de la ventana, ya que para este valor se considera que la red puede empezar a congestionarse. Este segundo algoritmo permite enviar únicamente un segmento (1MSS) por cada ACK recibido, por eso como se ve en la gráfica, la evolución es constante hasta la siguiente pérdida o *timeout* producido.

2.1.2. TCP RENO

En 1990, se propone el conocido algoritmo, TCP Reno, [1], siguiendo la misma línea de TCP Tahoe. Sin embargo, TCP Reno incluye otro modo, adicional al *Fast Retransmit*, de TCP Tahoe, llamado *Fast Recovery*, [2].

Este algoritmo considera que la recepción de ACK's duplicados no implica una congestión en la red, sino debido a una pérdida puntual. Una vez recibidos 3 ACK's duplicados, la ventana se reduce a la mitad, a diferencia de Tahoe que reducía la ventana a 1MSS. Que la ventana caiga hasta 1MSS, obliga a entrar en el modo *Slow-Start* cada vez que se detecte congestión, ya que $CWND < Ssthresh$. En la Figura 2.2 se puede ver este proceso, ya que la ventana cae hasta el mismo valor que el *ssthresh*, no hasta 1MSS como en la Figura 2.1. Una vez entrado en el modo *Fast Recovery*, TCP Reno espera confirmar el paquete retransmitido. Cuando llegue el ACK correspondiente a este segmento se continúa en el modo *Congestion Avoidance*, en vez de *Slow Start*, por eso en la gráfica, el modo *Fast Recovery* se representa como una zona estable, en la que se espera el ACK correspondiente del segmento perdido. Este método evita el cambio de estados continuo entre el modo *Slow-*

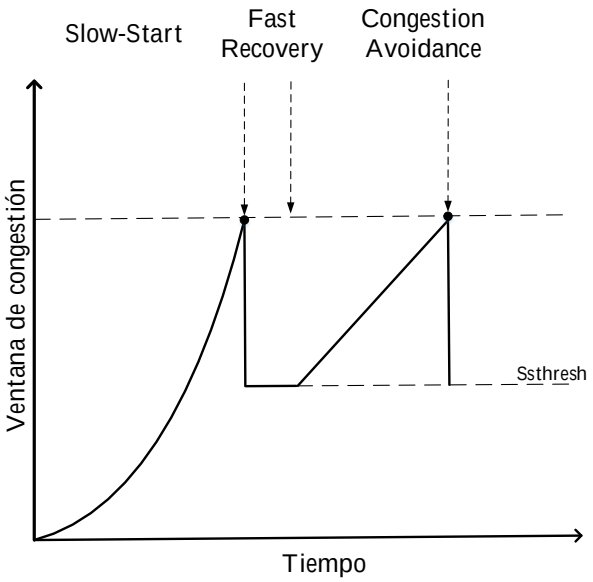


Figura 2.2: Evolución ventana de Congestión de TCP Reno

Start y *Congestion Avoidance* que había en TCP Tahoe, lo cual generaba inestabilidad y en consecuencia una mayor probabilidad de error.

2.1.3. TCP New Reno

TCP Reno proponía como máximo una retransmisión por cada RTT, sin embargo, no tiene en cuenta casos con múltiples pérdidas consecutivas. Por consiguiente, en 1999 se desarrolló el algoritmo de New Reno, [4]. New Reno, [5] es una de las principales implementaciones de TCP, de la cual se basan la mayoría de los siguientes protocolos, incluyendo el que se estudiará en profundidad en esta memoria, TCP CUBIC.

En TCP Reno, la espera del ACK el segmento retransmitido puede generar con gran probabilidad más pérdidas de paquetes, entrando así de manera continua en el modo *Fast Recovery*. Como solución, New Reno una vez haya recibido un triple ACK's, entra al igual que su antecesor en el modo *Fast Retransmit* tras realizar la caída de la ventana y retransmitir el paquete o los paquetes perdidos. Por cada ACK recibido en esta fase puede darse dos situaciones. Por un lado, que el ACK recibido reconozca un único paquete retransmitido, indicando que hay que seguir retransmitiendo los paquetes sucesivos. El otro caso, es que llegue un ACK que confirme todos los paquetes que han sido retransmitidos, lo cual supone el fin del *Fast Recovery*.

2.1.4. Opción TCP SACK y TCP FACK

TCP SACK (Selective Acknowledgment) no es un algoritmo de control de congestión como tal, sino más bien es una opción extendida de TCP New Reno. Partiendo de la mejora que supone el uso del Selective Acknowledgment (SACK), surgió TCP FACK (Forward Acknowledgment) que sí propone otro modelo nuevo para evitar saturaciones en la red.

TCP SACK, [6], fue desarrollado en 1996. Propone una retransmisión selectiva, de ahí su nombre, *Selective ACK*. Esta mejora evita la retransmisión innecesaria de paquetes que han llegado correctamente. En New Reno, el modo *Fast Retransmit* es activado si se detecta la pérdida de uno o varios paquetes o se produce un *timeout*. La retransmisión asociada en New Reno viene dada por el reenvío de todos los paquetes, desde el último no confirmado, hasta el último transmitido antes de detectar la pérdida, es decir, se hace uso de reconocimientos acumulativos. El reenvío de todos los paquetes anteriores al que indicaba pérdida no significa que no hubiesen llegado de manera correcta. Para mejorar en eficiencia, la opción SACK proporciona información en los ACK's de los paquetes que han llegado correctamente, indicando solamente aquellos que se han perdido, permitiendo así que el emisor reenvíe solo los paquetes perdidos, suponiendo una mejora en el rendimiento de la red.

Poco tiempo después de haberse desarrollado TCP SACK, nació TCP FACK como algoritmo de control de congestión, haciendo un punto de unión entre Reno y SACK. FACK hace referencia a *Forward Acknowledgment*, [7]. Por una parte, utiliza ACK selectivos, indicando específicamente los paquetes perdidos. La diferencia es que Reno y New Reno utilizan la información que proporcionan los SACK para ver cuántos paquetes hay que reenviar y cuáles han llegado correctamente. TCP FACK por el contrario, utiliza la información adicional proporcionada por los SACK para saber exactamente la cantidad de paquetes que están pendientes de enviar a la red pero de una forma más sencilla. Para ello hace uso de las tres siguientes variables:

H: número de secuencia más alto del paquete que aún no se ha enviado.
F: el número de secuencia más alto de los paquetes que han sido correctamente recibidos.
R: número de paquetes retransmitidos.

En la Figura 2.3, se puede ver este proceso. El paquete con número de secuencia 2 se ha perdido y posteriormente un paquete con número de secuencia 3 ha llegado correctamente. N2 indica con un ACK que sigue esperando el paquete 2 y con un SACK indicando que le ha llegado correctamente el 3 y espera a recibir el 4 también. TCP FACK realiza el sencillo cálculo de $H - F + R$ para saber cuántos paquetes están pendientes de enviar, en este caso, $H = 4$, $F = 3$ y $R = 0$. De esta forma N1 sabe que se ha perdido un paquete y que tiene que retransmitirlo, sumando una unidad a la variable R. Cuando el cálculo tenga valor nulo, N1 sabrá que han llegado correctamente todos los paquetes. La sencillez del uso de

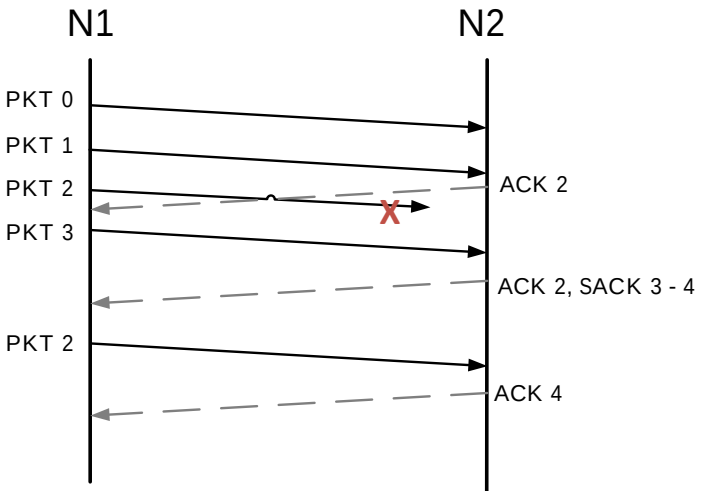


Figura 2.3: Envío de paquetes utilizando opción SACK

estas tres variables provoca que el protocolo TCP FACK tenga mayor robustez frente a algoritmos como Reno y New Reno.

2.1.5. TCP Vegas

TCP es una implementación desarrollada en 1995, basada en Reno. Este nuevo algoritmo difiere bastante de los anteriores, proponiendo nuevas técnicas a las vistas anteriormente.

Fueron Brakmo and Peterson los que propusieron cambiar el modo *Congestion Avoidance* en este nuevo protocolo, TCP Vegas, [1], [8]. La finalidad es medir la capacidad del buffer de los routers que forman la red para calcular el grado de congestión y anticiparse a futuras congestiones. Para ello se van tomando medidas del RTT. La variable RTT_{min} calculada es la referencia de un RTT libre de congestión. Si la actual medida del RTT es mayor al RTT_{min}, significa que hay paquetes almacenados en la cola y que pueden dar lugar a congestión. TCP Vegas es considerado un proactivo porque previene la congestión, mientras que en Reno solamente se sabe si han llegado un triple ACK o un *timeout*.

Por cada paquete enviado, se estima el RTT, como se explicó en la Sección 2.1.1 para TCP Tahoe. Por cada paquete enviado también se hace una marca de tiempo. Si el tiempo transcurrido desde el comienzo de esta marca es superior al RTT estimado, se inicia directamente la retransmisión del paquete, sin esperar a llegar al tercer ACK duplicado para activar el *Fast Retransmit* o a que se produzca *timeout*.

TCP Vegas, como en la versión de New Reno, se soluciona el problema de la retransmisión múltiple. Los últimos segmentos enviados antes de la retransmisión del paquete que indicó una posible congestión, también tienen su marca de tiempo. En caso de que estos

también superen el valor del RTT estimado, se retransmitirán como se hizo para el primero.

En el modo *Congestion Avoidance* también se realizan modificaciones. No solo propone una retransmisión proactiva, también mide la cantidad de paquetes que hay en la cola de los buffer por cada RTT. Este cálculo depende de la velocidad de transmisión los paquetes, es decir, del RTT actual. La variable Δ indica el número de paquetes en cola de la forma $\Delta = cwnd - cwnd_{cong}$, donde $cwnd_{cong}$ es el valor de la ventana cuando TCP Vegas detecta congestión. Atendiendo a este valor, se proponen dos umbrales, $\beta = 4$ y $\alpha = 2$. Una vez definidos estos, se destacan principalmente dos zonas:

Zona A: cuando $\Delta > \beta$, en este caso, la ventana decrece 1MSS.
Zona B: caso en el que $\Delta < \beta$, produciendo un incremento de ventana de 1MSS. Sin embargo, aquí se tiene en cuenta el segundo umbral para tener más exactitud y evitar inestabilidades. Si $\Delta < \alpha$ se incrementa, en caso contrario, se considera un estado donde no hay actualizaciones de la CWND.

Por último, TCP Vegas también propone una modificación ante el modo *Slow-Start*. En Reno, al iniciar la conexión, no se tiene conocimiento del ancho de banda disponible en la red. Quizá un crecimiento exponencial como es el del modo *Slow-Start*, podría dar lugar a una congestión en caso de que la capacidad de los buffer de la red fuese pequeña. Como solución, si la variable Δ excede el umbral propuesto, se sale de *Slow-Start* y se entra en *Congestion Avoidance*, evitando llegar al valor *ssthresh* de Reno. De este modo, si se detecta congestión en los buffer del cuello de botella, el crecimiento de la ventana de *Slow-Start* se detiene y se pasa a *Congestion Avoidance* para una evolución de la ventana más pausada.

2.1.6. TCP Westwood

El desarrollo de las redes de comunicación no solo se ha producido en medios cableados, también en inalámbricos. Atendiendo a las conexiones WiFi, un algoritmo de congestión que ofrece rendimientos considerables en la eficiencia de la red es TCP Westwood. Esta versión se considera también un punto de partida para otras futuras implementaciones más avanzadas.

Las redes inalámbricas el aire como medio de transmisión, esto produce que este tipo de redes sean propensas a la pérdida de paquetes, debido a las interferencias y obstáculos. Los algoritmos de congestión no conocen cuál ha sido la fuente de la pérdida, si se ha perdido por congestión o por interferencias de la señal, por ello, TCP Westwood hace uso de otros métodos para obtener mejores rendimientos frente a estas topologías.

TCP Westwood, [1], [9], partiendo del algoritmo de New Reno, suprime todo lo respectivo al modo *Fast Recovery*, proponiendo un método heurístico que permite la evolución de la ventana de congestión más adecuada para no reducir la tasa de envío en caso de pérdi-

da. Este método realiza el cálculo de la CWND de la forma: $CWND = Bandwidth \cdot RTT$, donde el ancho de banda representa la tasa de envío los paquetes ACK por parte del receptor multiplicado por el RTT, es decir, si el receptor mantiene constante la tasa del envío de ACK's, se considera que no hay un incremento de la congestión en los buffer. Con este método se estima cómo es la red desde el punto de vista del transmisor, dependiendo de lo recibido por parte del receptor.

Este algoritmo no solamente presenta resultados óptimos ante redes inalámbricas, también en redes donde el ancho de banda utilizado y el RTT toman valores altos, como es el caso de CUBIC.

2.1.7. BIC TCP

Por último, se va a hacer referencia a TCP BIC, [1], [10], que es el el antecesor de la versión que se va a evaluar en este proyecto, TCP CUBIC. TCP BIC es un algoritmo adaptado para redes de alta velocidad y grandes valores de RTT. Suponiendo que New Reno tuviese que adaptarse a este tipo de redes, el crecimiento de su ventana en el modo *Congestion Avoidance* provocaría una lenta evolución, impidiendo alcanzar valores eficaces en la tasa de transmisión.

BIC TCP, [1], nació en 2004 y propone una extensión a la versión New Reno, añadiendo un modo más, llamado *Fast Convergence*, obteniendo un valor más óptimo de la CWND en función de la disponibilidad en la red. El algoritmo de congestión BIC se divide en cuatro fases:

- ***Binary search increase***: la pérdida de un paquete indica la caída de la ventana de congestión. TCP BIC hace uso de las variables $cwnd_{min}$ y $cwnd_{max}$, que son el valor mínimo y máximo de la ventana, respectivamente. Cuando decrece la ventana, se actualiza $cwnd_{min}$ a ese valor. El valor óptimo se establece a la mitad, entre $cwnd_{min}$ y $cwnd_{max}$. Si el valor $cwnd_{max}$ toma valores altos, el proceso para alcanzar el valor óptimo de la ventana puede ser pesado, por este motivo, se proponen otras dos variables, S_{min} y S_{max}. Si el punto medio es más grande que S_{max}, la ventana solo crecerá hasta el valor que indique $SMmax$. El valor de la ventana en su crecimiento hasta dicho valor se hace referencia como $cwnd_{target}$. El proceso continuará hasta que la diferencia entre $cwnd_{target}$ y el valor actual de la ventana sea menor a S_{min}.
- ***Slow-Start***: esta fase no es como la conocida en New Reno. Si el valor de CWND es mayor a $cwnd_{max}$ hay que buscar el siguiente valor de este. En este modo, $cwnd_{min}$ a su vez se actualiza al valor actual de la ventana y para el cálculo de $cwnd_{max}$ se sigue el proceso de *Slow-Start* de New Reno, se duplica la ventana por cada ACK recibido, hasta que el múltiplo alcanza el valor que indica S_{max}. Una vez llegado a este

punto, se sale de *Slow-Start* y TCP BIC entra a *Binary search increase*, explicado anteriormente.

- ***Fast Convergence***: este modo ajusta un valor adecuado de la ventana de congestión para todos los flujos que compartan el mismo canal, impidiendo que un solo flujo no ocupe todo el ancho de banda, ofreciendo así las mismas condiciones para todos.
- ***Reno***: TCP BIC accede a este modo para calcular el incremento de ventana como lo haría New Reno. Si el resultado obtenido puede obtener resultados más óptimos si se compara con los obtenidos por BIC, se hará uso de este nuevo cálculo. Este caso se da principalmente en conexiones de baja latencia y ancho de banda.

La ventaja de TCP BIC es que propone un crecimiento lento alrededor de $cwnd_{max}$, lugar donde se produjo la pérdida. Si no se produce ninguna pérdida, la ventana volverá a incrementar rápidamente en búsqueda del nuevo $cwnd_{max}$. En el algoritmo de New Reno, esto no lo tiene en cuenta, y su crecimiento sigue la misma forma, independientemente de cuando se produjo la pérdida. Sin embargo, este protocolo puede provocar inestabilidad en la red debido al constante cambio de fases, incrementando así la probabilidad de pérdidas en la red.

2.2 TCP CUBIC como algoritmo de congestión

Una congestión en la red puede venir por diversos motivos, dando lugar a pérdida de paquetes, desorden en el trayecto de estos o una saturación en la red. Como solución, aparecen distintas variantes de TCP como se ha mencionado anteriormente. La versión estándar New Reno, propone un algoritmo básico del cual parten muchas de las implementaciones, incluido TCP CUBIC, [11], nacido en 2008.

La principal ventaja que propone TCP CUBIC es que la forma de la ventana de congestión sigue una función cúbica, lo que permite que sea más escalable y estable que su antecesor, el protocolo BIC. La función de la CWND está sigue la Expresión 2.1.

$$W_{cubic}(t) = C(t - K)^3 + W_{max} \tag{2.1}$$

Donde C es una constante de CUBIC, por defecto $C = 0.4$. El valor t (*elapsed time*) indica el tiempo transcurrido desde que se produjo la caída de la ventana hasta el momento actual. W_{max} es el valor de la CWND cuando se ha producido la pérdida y finalmente K es el tiempo estimado para alcanzarse W_{max} y está definido por la Expresión 2.2.

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

La variable β es una constante de caída proporcionada por CUBIC, cuyo valor puede variar, pero por defecto $\beta = 0.8$. Se puede analizar que W_{cubic} en la Expresión 2.1 no depende del valor de RTT, sino del tiempo trascurrido desde la última congestión producida.

La Figura 2.4 representa cómo es la forma que sigue la ventana en TCP CUBIC. Se puede observar que W_{max} delimita dos partes claramente diferenciables, *Steady State Behavior* y *Max Probing*.

- ***Steady State Behavior***: sigue una forma cóncava, este estado abarca desde que se produjo la reducción de la ventana, hasta que se alcanza el valor W_{max}, valor de la ventana en el que se produjo la última pérdida. Una vez reducida la ventana, esta empieza a crecer de forma rápida. Sin embargo, cuando se alcanzan valores cercanos a W_{max}, el crecimiento es más lento, debido a que esta zona se consideró conflictiva y produjo la reducción de la CWND en la última pérdida. Cuando $t = K$, CUBIC habrá alcanzado el valor de W_{max} y se cambiará al siguiente estado de la función.
- **Max Probing**: tras haber alcanzado el valor W_{max}, la ventana de congestión sigue una forma convexa. De la misma manera, en valores de ventana cercanos a W_{max}, la función crece pausadamente, una vez pasado este punto de inflexión, la ventana sigue un proceso evolutivo más rápido, hasta que se detecte otra pérdida en la red y haya que reducir la ventana.

Una de las ventajas de TCP CUBIC es que la ventana siga la forma de una función cúbica, evitando el cambio de estados que generen inestabilidad en la red, como en TCP BIC.

Sin embargo, no es la única ventaja, el crecimiento constante entorno a W_{max} proporciona también escalabilidad en la red. Un crecimiento pausado entorno a este valor en función de t obliga a que el envío de la información se realice de forma cautelosa en zonas donde se indicó la última vez que había congestión. Además, el crecimiento rápido de la ventana para valores lejanos a W_{max} también se ve positivamente afectado en redes con altos valores de ancho de banda y latencia, es decir, redes donde el producto de ambos

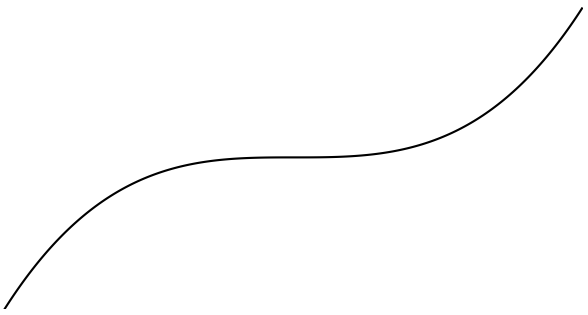


Figura 2.4: Crecimiento de la ventana de CUBIC

factores Bandwidth-Delay Product (BDP) es alto. Este tipo de redes son más propensas a error debido al aumento de flujo de los datos y del retardo que suponen estos al atravesar la red, por lo que TCP CUBIC con estas características, ofrece mejores rendimientos.

Otro de los problemas básicos en las redes de comunicación, es que varios flujos compartan un mismo canal para llegar a su destino. El problema recae sobre la capacidad utilizada por los flujos. Si dos o más flujos comparten el mismo canal, lo ideal sería que de una forma justa, todos compartieran el canal de la misma forma para obtener rendimientos similares. Como solución a esto, CUBIC realiza de forma justa el problema de la convergencia de flujos, [12]. Cuando el transmisor detecta una pérdida de datos, CUBIC no diferencia si ha unido un nuevo flujo a la red, por ello, si CWND $< W_{max}$, caso en el que se produce una pérdida antes de la anterior, el valor de W_{max} sigue la Expresión 2.3.En cambio, si CWND $> W_{max}$, el valor que toma sigue la Expresión 2.4.

$$W_{max} = cwnd \cdot \frac{1+\beta}{2} \tag{2.3}$$

$$W_{max} = cwnd \tag{2.4}$$

En referencia a 2.3, se observa que si la congestión se produce antes que la última registrada, el valor de W_{max} tomará un valor más bajo, ya que se multiplica por un factor menor que la unidad. Al tomar W_{max} un valor menor, la velocidad de transmisión del flujo principal comienza a tomar valores inferiores y alcanzará W_{max} previamente, permitiendo así que el resto de flujos aumenten el valor de su ventana. Una vez detectada la congestión, la ventana decrece de la forma $cwnd = cwnd \cdot \beta$. Como resultado, si la caída se produce en el flujo principal, la CWND al tener un valor alto por ocupar la mayor parte del ancho de banda del canal, tendrá una caída más grande que si se produce en los flujos secundarios, cuyo valor de CWND al principio de la conexión es menor. Al caer el flujo principal, CUBIC impedirá que se recupere hasta niveles tan altos, ya que el valor de W_{max} es inferior, ofreciendo así a los demás flujos una repartición justa del ancho de banda disponible.

Una vez recibido un ACK, tras producirse la congestión, se calcula el siguiente valor de W_{cubic}, como $W_{cubic}(t + RTT)$, siguiendo la Expresión 2.1. Dependiendo del resultado obtenido se puede trabajar en tres diferentes modos, los cuales se van a desarrollar a continuación.

2.2.1. TCP- Friendly

Hasta el momento, se ha hecho referencia a CUBIC a redes con un alto valor de BPD. Sin embargo, debe ser compatible con redes de otra índole. Aquí es donde aparece una de las limitaciones de CUBIC. Como se ha analizado en la Sección 2.2, CUBIC propone una evolución de la ventana dependiente del tiempo y es bastante independiente del RTT.

Como consecuencia, en redes con bajo RTT, la CWND crecerá más lento si se compara con algoritmos de congestión que sí dependen de este valor, como es el caso de New Reno. Aunque este problema se considere limitación, CUBIC también propone soluciones y no se queda atrás. Para obtener el mejor rendimiento ante este tipo de situaciones, el algoritmo de congestión propone una adaptación de la forma de su ventana lo más parecido a la de New Reno. Es aquí cuando TCP entra en el modo *TCP-Friendly*, [11].

Este modo intenta adaptar el envío de información lo más parecido posible al Additive-increase multiplicative-decrease (AIMD) de TCP New Reno. La media del AIMD sigue la Expresión 2.5.

$$\frac{1}{RTT} \cdot \sqrt{\frac{\alpha}{2} \cdot \frac{1+\beta}{1-\beta} \cdot \frac{1}{p}} \tag{2.5}$$

Donde α es lo que aumenta la ventana. β es el factor multiplicativo de decrecimiento del *ssthresh* y el valor p indica la probabilidad de error de los paquetes. Siguiendo el algoritmo del *Congestion Avoidance* de New Reno, se conoce que $\beta = 0.5$, ya que el umbral *ssthresh* cae a la mitad al detectarse una pérdida y $\alpha = 1$, ya que a partir de dicho valor el incremento de la CWND es de un segmento por cada ACK recibido. Aplicando estos valores a la Expresión 2.5 se obtiene la Expresión 2.6.

$$\frac{1}{RTT} \cdot \sqrt{\frac{3}{2} \cdot \frac{1}{p}} \tag{2.6}$$

El valor obtenido en esta última expresión es el objetivo a seguir para CUBIC. Al igualar la Expresión 2.5 y la Expresión 2.6 y dejar β como variable ($\beta_{newreno}$ y β_{cubic} no tiene por qué ser iguales) se obtiene el valor de α necesario para que CUBIC pueda cumplir la condición. El valor de α está representado en la Expresión 2.7.

$$\alpha = 3 \cdot \frac{1-\beta}{1+\beta} \tag{2.7}$$

Una vez obtenido el valor del incremento de la ventana, α, en función del decrecimiento, β, es posible alcanzar la misma media de envío que en New Reno, obteniendo finalmente la Expresión 2.8.

$$W_{tcp}(t) = W_{max} \cdot \beta + 3 \cdot \frac{1-\beta}{1+\beta} \cdot \frac{t}{RTT} \tag{2.8}$$

Este valor es calculado siempre que haya que actualizar la ventana, no es un modo al que se entra a parte tras terminar el proceso de otro. Una vez obtenido el valor, se compara con el valor de W_{cubic} de la Expresión 2.1, para ver si el valor obtenido siguiendo el algoritmo New Reno, supone una ventaja o no. Dependiendo del valor W_{tcp}, CUBIC se sitúa ante dos casos:

- $W_{tcp} > W_{cubic}$: el valor aproximado a New Reno es mayor, por lo que W_{cubic} va a obtener peores rendimientos, debido al lento crecimiento de su ventana en comparación al que podría proporcionar siguiendo el algoritmo de New Reno. La intención es actualizar este valor de la forma $W_{cubic} = W_{tcp}$ para mejor el rendimiento de la red.
- $W_{tcp} < W_{cubic}$: el valor W_{tcp} es menor que el proporcionado por el propio CUBIC. En este caso CUBIC no hace uso del modo *TCP Friendly*, siguiendo la evolución de la ventana de la Expresión 2.1. Para esta situación, el valor W_{tcp} no se utiliza pero sí se calcula porque el nodo fuente no tiene conocimiento de las características de la red, por lo que se podría decir que CUBIC es un protocolo proactivo ante esta situación.

2.2.2. TCP CUBIC en zona cóncava

La zona cóncava hace referencia a la zona *Steady State Behaviour*, en el que el valor actual de la ventana no ha llegado a W_{max}. Cuando llegue un ACK, la ventana se actualiza y tendrá que incrementar. Sin embargo, CUBIC no ofrece un incremento de ventana por cada ACK recibido. Para que se actualice, hay que tener en cuenta dos factores. Por un lado el valor *cnt*, que sigue la Expresión 2.9. El otro factor es un umbral que viene dado por el número de ACK's recibidos desde la última actualización de la ventana. Para entender mejor cómo se comporta CUBIC, este umbral recibirá el nombre por ahora de ACK_{cnt}. W_{cubic} para actualizarse va a tomar un valor más alto y al ser inversamente proporcional en la Expresión 2.9, como resultado obtendremos un resultado de *cnt* inferior, permitiendo así que solamente se actualice la ventana en el caso de que $ACK_{cnt} > cnt$. Hay que tener en cuenta que este mismo proceso también se aplica al modo *TCP Friendly*, donde el valor de *cnt* sigue la también la Expresión 2.9 pero se utiliza el valor de W_{tcp}.

$$cnt = \frac{cwnd}{(W_{cubic}(t + RTT) - cwnd} \tag{2.9}$$

2.2.3. TCP CUBIC en zona convexa

La zona convexa hace alusión a *Max Probing*, explicado en la Sección 2.2.1. En este caso, no hay ninguna diferencia ante la zona cóncava, se sigue el mismo proceso. El valor de *cnt* se calcula igual, la forma convexa viene dada por la forma que sigue la ventana al seguir la Expresión 2.1.

3

NS-3 Simulator

En este capítulo se hace una introducción a la estructura del simulador utilizado para realizar este proyecto, ns-3. En primer lugar se realiza una breve descripción de su arquitectura, prestando mayor atención en aquellos módulos que trata el protocolo TCP y las aplicaciones más utilizadas. La segunda parte hace mención a la implementación de TCP CUBIC para ns-3, se hará un resumen de las funciones y variables más importantes para complementar los conocimientos expuestos en anteriores capítulos.

3.1 Fundamentos básicos de ns-3

Atendiendo a su historia, el Network Simulator (ns) tiene sus orígenes alrededor del año 1989 como una extensión de otro simulador, llamado *REAL*. Sin embargo, no nació la primera versión oficial e independiente hasta 1995 con la ayuda de DARPA (agencia encargada de desarrollar tecnologías para el uso militar), la Universidad de Berkeley y el centro de investigación XEROX PARC de California, entre otros. El núcleo de ns-1 estaba fundamentado en un lenguaje básico de programación de objetos, como es el C++, además de incluir también en sus bases el lenguaje Tool Command Language (Tcl). Años más tarde, nació la segunda versión del simulador, conocida como ns-2. En esta contribuyeron otros participantes como Sun Microsystems y UCB - Daelus, aportando módulos para simular topologías inalámbricas. ns-2 siguió basando su arquitectura en el lenguaje C++ pero evolucionó ante el lenguaje de comandos Tcl, pasando a utilizar su extensión orientada a objetos, llamada Object Tool Command Language (OTcl). Finalmente, en 2005 de la mano de Tom Henderson se propone el núcleo de la tercera y actual versión del simulador, totalmente incompatible con la anterior. Desarrolladores de la Universidad de Washington, el Instituto Tecnológico de Georgia y el centro de investigación francés INRIA dieron paso finalmente en Junio de 2008 al primer lanzamiento, llamado ns-3.1. Actualmente, esta versión sigue en desarrollo, encontrándose ahora mismo en la ns-3.25, lanzada en Marzo del 2016. El núcleo sigue centrado en el uso del lenguaje C++, sin embargo, también tiene disponibles sus características para el lenguaje Phyton.

En definición, la plataforma de simulación ns-3 es un conjunto de librerías que proporciona a los usuarios un amplio entorno de simulación para el uso educacional y de investigación.La finalidad, es evitar la utilización de sistemas reales que incrementen el coste y la dificultad de la investigación. El simulador ns-3 es usado principalmente en entornos Linux y al ser de software libre,bajo licencia GNU GPLv2, permite la modificación y extensión de su código para ofrecer mayor libertad a la hora de obtener los resultados requeridos.

Para este proyecto, se ha utilizado la versión 3.24, la cual no incluía aún la implementación para CUBIC en su estructura, aunque gracias al código proporcionado por el Instituto Tecnológico de Atlanta, [13], se han podido añadir los módulos correspondientes para realizar los futuros resultados que se verán en el capítulo 4.

3.2 Estructura ns-3

Esta sección va a hacer una clasificación de la estructura de ns-3, además de explicar cómo funcionan las aplicaciones y los módulos TCP que se utilizan fundamentalmente.

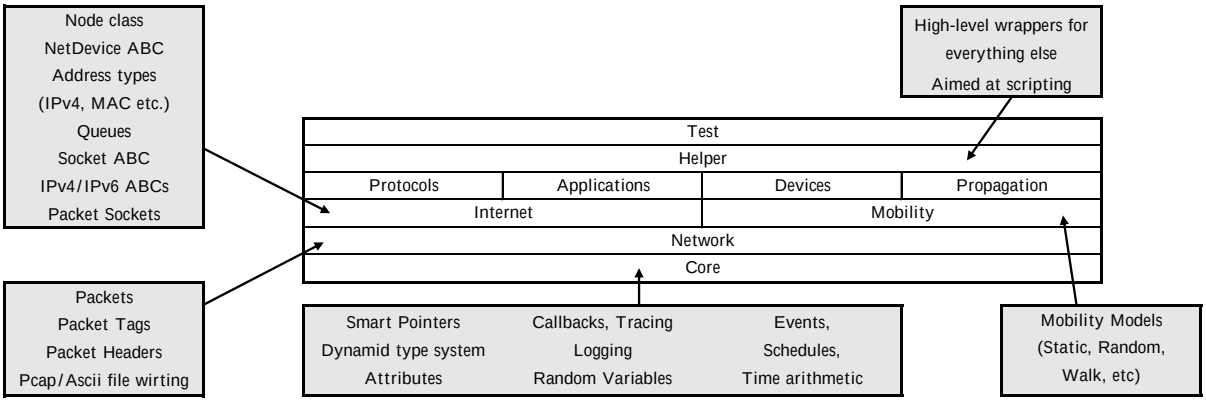


Figura 3.1: Estructura del simulador ns-3

En cuanto a la estructura general, ns-3 sigue un sistema de directorios, abarcando una amplia gama de protocolos, desde la capa física, hasta el nivel de aplicación. Como se observa en la Figura 3.1, la estructura del simulador está dividida en capas. La capa inferior incluye todo tipo de parámetros relacionados con el núcleo, como son tiempos, eventos y atributos generales. A medida que se asciende a capas superiores, es cuando entran en proceso los módulos más importantes que se van a mencionar a continuación.

Este proyecto presta atención principalmente en los módulos de *Internet*, *Network* y *Applications*, además de otros proporcionados por el Grupo de Telemática de la Universidad

de Cantabria para facilitar el despliegue y el análisis de los resultados. El módulo *Network*, abarca principalmente el diseño de los paquetes, sockets, cabeceras y direcciones, por lo que se considera un módulo básico. Por ello, se va a prestar más atención a los otros dos restantes.

Applications module

De nuevo, la cantidad de aplicaciones es bastante extensa, sin embargo, este proyecto está focalizado en tres:

- **Bulk Send Application**: es una aplicación utilizada para generar información. Su principal función es enviar tantos paquetes como permita el canal durante un tiempo determinado. Para ello se le pueden modificar atributos para flexibilizar la búsqueda de resultados. En este caso, solamente ha sido preciso modificar *SendSize*, para indicar el tamaño de los paquetes. Está definida en *src/applications/model/bulk-send-application.cc/.h* y deriva de la clase *Application*.
- **PacketSink**: es utilizada por los nodos receptores. Simplemente recibe la información por parte de los nodos fuente y es independiente del tipo de aplicación que ha genera el tráfico desde el otro extremo. Simplemente se instala en el nodo correspondiente , asociando un socket creado previamente y se delimita el tiempo que tiene que estar escuchando el tráfico mediante las variables *Start Stop*. Está definida en: *src/applications/model/packet-sink.cc/.h* y las anteriores, también deriva de *Application*.

Internet module

Este módulo abarca todo lo referente al protocolo IP, TCP y UDP, aunque una vez más, se va a describir únicamente los módulos más utilizados en este proyecto y aquello que se han modificado.

- **Clase TcpSocket**: almacena todas las propiedades del socket asociado a los nodos, indicando parámetros como el tamaño de los buffer, el umbral *ssthresh*, el tamaño del segmento o el valor inicial de la ventana. Se encuentra en: *src/internet/model/tcp-socket.cc/.h* y deriva de *Socket*.
- **Clase TcpSocketFactory**: es utilizada por la clase *TcpL4Protocol* para generar el socket correspondiente. No depende de ningún parámetro, genera el socket con su correspondiente Id, los parámetros derivan de *TcpSocket*. Se encuentra en: *src/internet /model/tcp-socket-factory.cc/.h*, derivando de *SocketFactory*

- **Clase TcpHeader**: define todos los parámetros de la cabecera TCP, desde las longitudes hasta los números de secuencia de los paquetes, incluyendo también los checksum correspondientes. Para más información, esta clase está definida en:*src/internet/model/tcp-header.cc/.h* y deriva de la clase *Header*

- **Clase TcpRxBuffer**: hace referencia al buffer de recepción, manejando parámetros como es el tamaño del buffer diponible o el número de secuencia. Además realiza la extracción y adicción de los paquetes desde los nodos fuente hacia la red. Deriva de la clase *Object* y se puede encontrar el código en: *src/internet/model/tcp-rx-buffer.cc/.h.*

- **Clase TcpTxBuffer**: al igual que *TcpRxBuffer* pero esta hace referencia al buffer del transmisor, abarcando la misma información que se ha explicado en la anterior clase pero haciendo referencia al otro extremo de la conexión. Deriva de la clase *Object* y su código está almacenado en: *src/internet/model/tcp-tx-buffer.cc/.h.*

- **Clase RttEstimator**: realiza el cálculo del RTT estimado para darlo a conocer a los algoritmos de congestión necesarios, y permite saber el tiempo que tarda un segmento en llegar a su destino, procesarse y que su correspondiente reconocimiento llegue de nuevo al nodo origen. Está basada en funciones que utilizan marcas de tiempo para calcular la medida la varianza de los datos obtenidos. Esta clase deriva de *Object* y se encuentra en: *src/internet/model/rtt-estimator.cc/.h.*

- **Clase TcpL4Protocol**: conecta con la capa IP inferior, para ser más exacto, envía y recibe información de la clase *IpL4Protocol*. Además ofrece conexión también con el socket, ofreciendo los paquetes a las capas superiores y viceversa. Deriva de la clase *IpL4Protocol* y está definida en: *src/internet/model/tcp-l4-protocol.cc/.h.*

- **Clase TcpSocketBase**: es la clase que sirve de punto de conexión entre los algoritmos de congestión de TCP y el protocolo TCP como tal, definido en las anteriores clases. Almacena la información acerca de atributos como son los buffer, el RTT utilizado, las opciones de TCP, ventanas de congestión y marcas de tiempo que son utilizados para trabajar en los distintos modos de operación del propio algoritmo de congestión. Deriva de *TcpSocket* y está definida en: *src/internet/model/tcp-socket-base.cc/.h.*

Las anteriores clases, se consideran las clases base del simulador ns-3, utilizadas por el protocolo TCP. A parte de estas, también se ha de mencionar los módulos que implican los algoritmos de congestión, TCP New Reno y TCP CUBIC. Este último se tuvo que añadir al entorno ya que no estaba implementado por defecto. Ambos, derivan de la clase *TcpSocketBase* y su código se encuentra en: *src/internet/model/tcp-newreno.cc/.h* y *src/internet/model/tcp-cubic.cc/.h* respectivamente. En cuanto a la definición de qué funciones realizan y cómo lo hacen, la información de New Reno se puede encontrar en la Sección 2.1.3, complementada con el algoritmo Reno de la Sección 2.1.2, mientras que la de CUBIC, en la Sección 2.2.

Por otro lado, se ha añadido un módulo proporcionado por el Grupo de Telemática de la Universidad de Cantabria, llamado *Scenario-Creator*. Sobre este se han realizado ampliaciones en el código para adaptarse a la finalidad de esta investigación. La utilización de estos módulos tenía la necesidad de modificar los archivos base del propio simulador NS-3 para su correcto funcionamiento, aunque no se genera ningún cambio en el modo de funcionamiento, simplemente se hace uso de la flexibilidad de ns-3 para extender el código. En cuanto a su clasificación, se divide principalmente en las dos siguientes clases:

- **Clase Configure Scenario**: su función principal es simplificar el despligue de escenarios, partiendo de un archivo externo donde se guardan los diferentes parámetros y la configuración de la red, obteniendo así un módulo sistemático y sencillo para realizar experimentos. Deriva de la clase *Object* y está situado en: *src/scenario-creator/model/configure-scenario.cc/.h.*
- **Clase Proprietary Tracing**: clase que se ha utilizado en todas las simulaciones. Está creada para obtener los tipos de trazas necesarias a diferentes niveles, desde el nivel físico, hasta el nivel de aplicación, permitiendo guardar los datos en un fichero de salida. Deriva de *Object* y se ubica en: *src/scenario-creator/model/proprietary-tracing.cc/.h.*

3.3 TCP-CUBIC en ns-3

En esta sección se va a explicar cómo se implementa en el simulador ns-3 el algoritmo original de CUBIC. Desde la primera versión del simulador ns-3, implementada en 2006 para el Kernel de Linux, se han producido muchas actualizaciones, en este caso se ha utilizado la versión 3.24, desarrollada por Brett Levasseur, Mark Claypool y Robert Kinicki. Antes de comenzar, esta implementación no sigue al pie de la letra lo explicado anteriormente, utiliza propias variables para adaptarse al Kernel de Linux y el cálculo de las funciones de CUBIC están escalados por más factores, como se podrá ver en la siguiente sección.

3.3.1. Variables de CUBIC en ns-3

Hay variables que no se utilizan en el algoritmo original, por ello se va realizar una división en dos tipos de variables. Aquellas que tienen relación con el algoritmo original son:

- β – constante β, por defecto en la implementación, $\beta = 819$, mientras que en CUBIC en general este valor es 0.8. Sin embargo, cuando se trabaje con *beta* se dividirá por la constante $BICTCP\_BETA\_SCALE$, obteniendo como resultado $beta \supset 0.8$.

- ***t*** – hace alusión a la variable t ya definida en el algoritmo original.
- ***originPoint*** – valor de la CWND una vez producida la congestión, es decir, el punto de partida de *Steady State Behaviour*.
- ***cwnd*** – variable que indica cuánto vale la ventana de congestión actual.
- ***ssThresh*** – hace referencia al umbral de congestión $ssThresh$ asociado a New Reno. El valor inicial es 64KB.
- ***cnt*** – variable que guarda el cálculo de cnt para decidir posteriormente si es posible actualizar la ventana.
- ***cWndCnt*** – contador de ACK's recibidos desde la última pérdida. Es el umbral que en la sección 2.2.2 se llamó ACK_{cnt}.
- ***lastMax*** – coincide con W_{max}.
- ***cwndSeg*** – este valor se utiliza para actualizar la ventana, y se calcula de la forma $cwndSeg = \frac{cwnd}{segmentSize}$, es decir, el valor de la ventana en segmentos.
- ***tcpCwnd*** – variable que indica el W_{tcp} del modo *TCP-Friendly*.
- ***windowTarget*** – valor de W_{cubic}.
- ***K*** – valor K del CUBIC original.
- ***tcpFriendly*** – variable booleana que indica si está en *TCP-Friendly*.
- ***inFastRec*** – indica si está en *Fast Recovery*.
- ***recover*** – número de secuencia del paquete perdido que se ha de retransmitir.
- ***dMin*** – el valor mínimo del RTT, se tiene en cuenta para calcular el valor de t.

Por otro lado las variables no definidas en el algoritmo original de CUBIC y que complementan a las anteriores, están representadas en ns-3 como:

- ***epoch_start*** – indica el momento en el que se ha producido la pérdida de paquetes. A partir de esta variable se calculan los valores de t y K para la Expresión 2.1.
- ***JIFFY_RATIO*** – al haber variables que dependen del tiempo ha de conocerse también los *jiffies*. En Linux, un *jiffy* es el tiempo entre dos eventos de reloj del ordenador. Son marcas de tiempo que toman valores pequeños, entre 1 y 10 ms, dependiendo del reloj de cada ordenador. Es decir, sirve de conversión entre la marca de tiempo, y el ordenador. En la implementación $JIFFY\_RATIO = 1000$, que en realidad hace referencia a 1ms.

- ***bicScale*** – este valor no forma parte del algoritmo original de CUBIC pero sí de la implementación de Linux. *bicScale* es usado para calcular C y *cubeRttScale*. Por defecto $bicScale = 41$.
- ***cubeRttScale*** – es una constante de escalado del valor original C. Sin embargo, solo escala para calcular el valor C en W_{cubic}(Expresión 2.1). Para el valor de C en el cálculo de K no se utiliza. Por defecto $cubeRttScale = bicScale \cdot 10$.
- ***BICTCP_BETA_SCALE*** – constante de escalado usada en el valor β de CUBIC. $\beta = 1024$.
- ***BICTCP_HZ*** – otra variable utilizada para convertir unidades de tiempo. $BICTCP\_HZ = 10$.
- ***cubeFactor*** – utilizada para calcular K, por defecto: $cubeFactor = \frac{1ull<<(10+3\cdot BICTCP\_HZ)}{410}$. 1ull indica que debe ser tratado como una variable de tipo unsigned long long. Si se tiene en cuenta los valores por defecto de las variables que implican el cálculo, el resultado es $10,0732$.

3.3.2. Funciones CUBIC en ns-3

Una vez que se sabe de qué forma interpreta ns-3 las variables, se va a pasar a hacer un análisis de las funciones, explicando los pasos que realizan y asociándolo al algoritmo original descrito en la sección 2.2.

NewAck()

A esta función se accede en caso de llegada de un nuevo ACK, el pseudo-código asociado está representado en la Función 1. Esta función está dividida en tres partes, cada una depende a un modo de operación distinto. La primera parte tiene la misma estructura que en New Reno, se comprueba si el modo *Fast-Recovery* está activado y con la variable *recover* se decide si es un $ACK_{partial}$ o ACK_{full}, como se explica en 2.1.3. En caso de no estar en *Fast-Recovery*, la segunda parte de la función comprueba si se encuentra en *Slow-Start*, para ello, el valor de la ventana tiene que superar el umbral de congestión *ssthresh*.

Finalmente en la tercera y última parte, es donde entra el propio algoritmo de CUBIC. Al principio se hace una estimación del RTT y se comprueba si es el mínimo obtenido, en caso contrario se actualiza para poder calcular $W_{cubic}(t + RTT)$. La búsqueda del RTT mínimo permite obtener el máximo valor de W_{cubic}. Hay que destacar que el RTT mínimo es interpretado como el RTT de un camino libre de congestión, declarando CUBIC como un protocolo reactivo ante esta situación. Posteriormente se comprueba si se puede actualizar

```
NewAck():
if inFastRec & seq < recover then
 | Retransmit ();
else if inFastRec & seq ≥ recover then
 | cwnd = min (ssThresh, BytesInFlight () + segmentSize);
 | inFastRec = false;
end
if cWnd ≤ ssThresh then
 | cWnd += segmentSize;
else
 | rtt = rtt → Estimate ();
 | if dMin == 0 then
 |  | dMin = rtt ≪ 3;
 | end
 | cnt = CubicUpdate ();
 | if cnt ≠ 0 then
 |  | if cwndCnt > cnt then
 |  |  | cwnd += segmentSize;
 |  |  | cwndCnt = 0;
 |  | else
 |  |  | cwndCnt += 1;
 |  | end
 | end
end
```

Función 1: Pseudo-código de NewACK()

la ventana habiendo recibido el parámetro *cnt* de la función *CubicUpdate()*. Si este valor es menor que *cwndCnt* (variable que indica la cantidad de segmentos recibidos tras haberse producido la última congestión), se actualizará, como se ha explicado en 2.2.2.

CubicRoot ()

CUBIC utiliza la raíz cúbica para obtener el valor K. En la implementación de ns-3 esto cambia, se hace uso de la función CubicRoot(). En un caso normal se utilizaría la función pow de C++, pero el uso de CubicRoot ofrece mayor eficiencia obteniendo el resultado hasta 10 veces más rápido. Sin embargo, no es un método exacto, se basa en el cálculo de Newton-Rhaphson y una tabla de *lookup*, el cual propone un error con una media aproximada del 0.195 %. Newton-Rhaphson sigue un método iterativo. En la primera iteración, el número se divide por 3, mientras que las siguientes iteraciones siguen la Expresión 3.1. El valor a hace referencia al número al que se le aplica la raíz cúbica.

Cuando la diferencia entre el último valor calculado y su anterior es muy pequeño, se detiene, dando el resultado final.

$$x_{k+1} = \frac{1}{3}(\frac{a}{x_k^2} + 2 \cdot x_k) \tag{3.1}$$

CubicUpdate ()

En esta función se obtiene el valor de *cnt* que permite actualizar la ventana cuando llegue un ACK, el pseudo-código asociado está representado en la Función 2. Se hace uso principalmente de las variables *cwndSeg*, *windowTarget* y *cnt*. Trabajar con *cwndSeg*, implica que se utiliza el valor de la ventana en segmentos, en vez de en bytes.

```
CubicUpdate():
cwndSeg = cwnd / segmentSize;
ackCnt += 1;
lastTime = time_stamp ();
if epochStart == 0 then
    epochStart = time_stamp ();
    ackCnt = 1;
    if lastMax ≤ cwndSeg then
        k = 0 ;
        originPoint = cwndSeg ;
    else
        k = CubicRoot(((lastMax − cwndSeg) · cubeFactor)) ;
        originPoint = lastMax ;
    end
end
t = ((time_stamp() − epochStart + (dMin ≫ 3)) ≪ BICTCP_HZ) / HZ ;
if windowTarget > cwndSeg then
    cnt = cwndSeg / (windowTarget − cwndSeg);
else
    cnt = 100 · cwndSeg;
end
if tcpFriendly then
    cnt = CubicTcpFriendliness(cnt);
end
return cnt;
```

Función 2: Pseudo-código de CubicUpdate()

Primero se añade una unidad a *ackCnt*, que es la variable que indica el número de ACK recibidos desde la última caída de la ventana y posteriormente se decide si es posible dar el incremento de la ventana desde la función *NewAck()*. En segundo lugar, si *epochStart* es 0, indica que se acaba de producir la congestión. Si la ventana está por debajo de W_{max} habrá que realizar el cálculo de K mediante *CubicRoot()*, siguiendo la Expresión 3.2 y dejar indicado que el *originPoint* sigue siendo W_{max}. En caso de ser mayor, se sabe que W_{max} ya se ha alcanzado y por lo tanto $K = 0$. El *originPoint* por otro lado se actualizará cada vez que entre al valor actual de la ventana hasta que se produzca la siguiente congestión, obteniendo finalmente W_{max} y así poder calcular W_{cubic}.

$$k = CubicRoot(((lastMax - cwndSeg) \cdot cubeFactor)) \tag{3.2}$$

A continuación, se calcula t y $windowTarget$, haciendo uso de variables descritas en 3.3.1 que no aparecen en el algoritmo original, pero son necesarias para trabajar en el Kernel de Linux. $windowTarget$ está asociado a W_{cubic}. El siguiente paso es calcular *cnt*, para ello si W_{cubic} es mayor que la ventana, se actualiza *cnt* siguiendo la Expresión 2.9. En caso contrario, se considera que el crecimiento es muy pequeño y se aplica directamente la Expresión 3.3.

$$cnt = 100 \cdot cwnd \tag{3.3}$$

Finalmente, se calcula el valor de *cnt* en el modo *TCP-Friendly*, mediante la función *CubicTcpFriendliness* explicada a continuación.

CubicTcpFriendliness()

Realiza el cálculo del *cnt* con el valor de W_{tcp} para el modo *TCP-Friendly* de CUBIC. Siguiendo la Función 3, en primer lugar, se calcula W_{tcp} como se ha explicado en la Sección 2.2.1. CUBIC ha de seguir la Expresión 2.8 del AIMD de New Reno para aumentar su eficiencia. La diferencia con New Reno empieza cuando se entra al modo *Congestion Avoidance*, en el que la ventana aumenta 1MSS por cada ACK recibido. ns-3 utiliza la variable *betaScale* para calcular el α en la Expresión 2.7. A diferencia del algoritmo original, *betaScale* está pasado a unidades del Kernel de Linux de la forma descrita en la ecuación 3.4. Finalmente, *ackCnt* decrece en términos de *betaScale* (variable que indica cómo crece la ventana) para aumentar así W_{tcp} un 1MSS.

$$betaScale = \frac{1}{3} \cdot \frac{8 \cdot (BICTCP\_BETA\_SCALE + beta)}{BICTCP\_BETA\_SCALE - beta} \tag{3.4}$$

El valor max_cnt indica el *cnt* calculado con W_{tcp}. Finalmente, si valor de *cnt* obtenido en la función *CubicUpdate()* es mayor que el calculado en *CubicTcpFriendliness*, se hace uso del segundo al resultar más eficiente. Antes de devolver el valor obtenido de *cnt* se le aplica la Expresión 3.5. Dividirlo por la variable *delayedAck*, es una técnica utilizada para

```
TCPFriendliness():
cwndSeg = cwnd / segmentSize;
max_cnt = 0;
betaScale = 1/3 · 8·(BICTCP_BETA_SCALE+beta) / BICTCP_BETA_SCALE−beta;
while ackCnt > (cwndSeg · betaScale) ≫ 3 do
    ackCnt -= (cwndSeg · betaScale) ≫ 3;
    tcpCwnd += 1;
end
if tcpCwnd > cwndSeg then
    max_cnt = cwndSeg / tcpCwnd−cwndSeg;
    if cnt > max_cnt then
        cnt = max_cnt;
    end
end
cnt = cnt<<4 / delayedAck;
return cnt;
```

Función 3: Pseudo-código de TCPFriendliness()

reducir el número de respuestas enviadas por el receptor, en la que se combinan varios ACK's en uno único para obtener mayor eficiencia.

$$cnt = \frac{cnt << 4}{delayedAck} \tag{3.5}$$

DupAck()

Esta función es llamada cuando se detectan ACK's duplicados, para activar el modo *Fast Recovery* y retransmitir aquellos paquetes perdidos. La variable *retxThresh* indica el contador de ACK necesarios para entrar en dicho modo, como se sabe, este valor es 3 en New Reno. Una vez se alcance el umbral se realizarán los siguientes pasos: Se calcula W_{max} por si la siguiente vez ya se da la congestión y hay que reducir la ventana. W_{max} dependerá de si el valor de la ventana es menor (Expresión 2.3, o si es mayor Expresión 2.4). Finalmente, se activa el modo *Fast-recovery* y tanto la ventana, como el umbral *ssthresh* decrecen, multiplicándose por β. En este caso β está referenciado a la variable *temp* que sigue la Expresión 3.6 e impide que la ventana no caiga por debajo de 2MSS.

$$temp = max(2, \frac{cwndSeg \cdot beta}{BICTCP\_BETA\_SCALE}) \tag{3.6}$$

```
DupAck():
cwndSeg = cwnd / segmentSize;
if retxThresh == 3 & !inFastRec then
    epochStart = 0;
    if cwndSeg < lastMax then
        lastMax = cwndSeg·(BICTCP_BETA_SCALE+beta) / 2·BICTCP_BETA_SCALE ;
    else
        lastMax = cwndSeg;
    end
    temp= max(2, cwndSeg·beta / BICTCP_BETA_SCALE);
    cwnd = temp · segmentSize ;
    ssThresh = temp · segmentSize ;
    inFastRec = true;
    DoRetransmit ();
else if inFastRec then
    cwnd += segmentSize;
    SendPendingData ();
else if !inFastRec & Size(txBuffer) > 0 then
    nextTxSequence = SendDataPacket ();
end
```

Función 4: Pseudo-código de DupACK()

Una vez activado el *Fast-recovery* en el anterior caso, si llega otro ACK duplicado, la ventana de aumenta en 1MSS por cada RTT y se envía tantos datos pendientes de enviar como permita la ventana del transmisor (función SendPendingData de TcpSocketBase ns-3). Finalmente, si no estamos en *Fast-recovery*, llega un ACK duplicado y además el buffer del transmisor es mayor que 0, es que hay información para enviar pendiente y se envían el máximo número de bytes posibles haciendo llamada a la función *SendDataPacket* de *TcpSocketBase*.

3.4 Adaptación de ns-3 al proyecto

Esta sección va a describir diferentes obstáculos que han encontrado durante la realización de este proyecto y se han ido solventando.

3.4.1. ns-3 y la ventana de congestión

En primer lugar, aún habiendo versiones posteriores, para este proyecto se utilizó la versión 3.20, ya que al comienzo del proyecto era la versión más reciente que se adaptaba a la implementación de CUBIC. Con respecto a esta versión, la simulación de diferentes escenarios daba lugar a datos erróneos, en los que la ventana de congestión y el rendimiento de la red no alcanzaban valores superiores a 10 Mbps. Para topologías donde el ancho de banda era pequeño no generaban ningún error puesto que no se excedía dicho valor. El problema surgía al realizar simulaciones con altos anchos de banda, ya que es en estas redes donde CUBIC obtiene mejores rendimientos, los cuales no se estaban obteniendo. Pero el problema no tenía origen en CUBIC, el mismo caso afectaba de igual forma a New Reno. En la Figura 3.2 se puede ver que el límite del throughput, el cual no supera los 10 Mbps, ante un experimento sin errores, con un ancho de banda de 250 Mbps y 54ms de RTT.

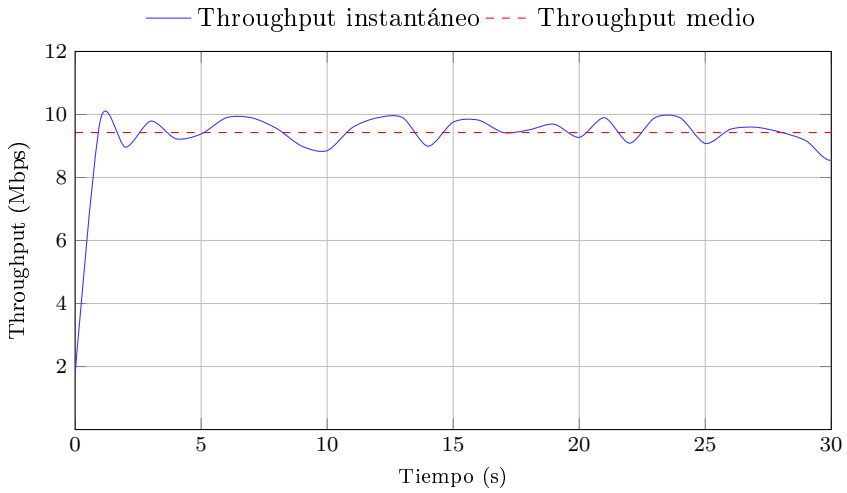


Figura 3.2: Evolución del troughput sin opción de Window Scaling

El problema estaba en el campo ventana de la cabecera TCP, representada en la Figura 3.3, algo independiente del algoritmo de congestión utilizado. Este campo tiene una longitud de 2 Bytes e indica el valor de la ventana de congestión, lo que implica que el valor máximo que puede representar este campo es 2^{16}, es decir, 65536 bytes. Siguiendo el experimento mostrado de la Figura 3.2, el valor máximo del rendimiento de la red (throughput) viene dado por la Expresión 3.7. Por eso mismo, se puede observar en la Figura un throughput medio de 9.7090 Mbps como máximo.

$$Throughput = \frac{CWND_{max}\ (\text{bits})}{\text{RTT}\ (\text{seg})} = \frac{65536 \cdot 8}{0.054} = 9.7090\ \text{Mbps} \tag{3.7}$$

1	2	3	4
Puerto origen		Puerto Destino	
Numero de secuencia			
Numero de acuse de recibo			
Longitud de Cabecera	Reservado/Flags	Tamaño de ventana	
Checksum		Puntero urgente	
Opciones			

Figura 3.3: Cabecera TCP

En consecuencia, cuando la ventana tomaba valores superiores a este no entraba en dicho campo, impidiendo alcanzar los valores correctos.

La solución a este problema es activar la opción TCP de *Window Scale*, definida en IETF RFC 1323, [14] (actualmente está implementada por defecto). Esta opción permite alcanzar valores superiores a 65536 Bytes. Para ello, se negocia directamente un factor de escalado en los paquetes de sincronización del inicio de la transmisión. Al tener dos ventanas, una de transmisión y otra de recepción, se negociará uno en cada sentido. Este factor de escalado se elige dependiendo de cual sea el buffer máximo de TCP asociado a la red. Para la versión 3.20 esta opción aún no estaba implementada en el simulador, lo que conllevó a utilizar una versión superior de ns-3. Finalmente como se ha mencionado en 3.1, se ha utilizado a lo largo del proyecto la versión 3.24 que sí tiene implementada la opción de TCP Window Scale.

3.4.2. ns-3 y adaptaciones del laboratorio

Ante el problema del Window Scale, el cambio de versión también conllevó a la modificación del código para el módulo *Scenario Creator*, explicado en 3.2.

La utilización de estos módulos modificaba archivos base del propio simulador ns-3 para su funcionamiento. Al cambiar de versión, el código de los archivos base se tuvo que adaptar de nuevo uno a uno para que el funcionamiento de estos módulos fuese el correcto. Además en la versión 3.24 algunos ficheros utilizados fueron sustituidos por otros, lo cual generó problemas y tuvo o que añadirse los antiguos al construir la versión o sustituirlos por los nuevos. Uno de los cambios importantes, fue en la clase *Error-model*, la cual se emplea para generar un modelo de errores en los escenarios propuestos. De esta clase derivaban otras, como por ejemplo *hash-id*. Al cambiar de versión, estas clases desaparecieron, sustituyéndose por otras. El proceso de cambio de versión fue algo tedioso, puesto que implicaban clases que no se utilizaron en el proyecto, pero que a la hora de compilar todos los módulos generaban errores que había que resolver.

4

Resultados y comparaciones

A lo largo de este capítulo, se van a realizar diferentes experimentos sencillos para evaluar el rendimiento de CUBIC y qué características presenta frente a otras versiones de TCP. Primero se presentan los escenarios y parámetros fundamentales utilizados para posteriormente poder comprender el tipo de resultados obtenidos. Finalmente se recogen todos los datos contraponiéndolos con los conceptos tratados en capítulos anteriores y se realizará una comparativa con protocolos como New Reno y Westwood para ver las principales diferencias.

4.1 Escenarios y parámetros utilizados

A continuación se va describir qué parámetros y escenarios se han utilizado para las simulaciones. En la sección 4.2 se expondrán los resultados TCP evaluados.

4.1.1. Parámetros fundamentales

Aprovechando la flexibilidad que proporciona el simulador ns-3, se ha han evaluado diferentes parámetros que conviene analizar para entender mejor el comportamiento de TCP.

- *RTT*: es uno de los parámetros fundamentales en este proyecto. Hace referencia al tiempo que tarda un paquete en salir del nodo fuente, llegar al nodo destino y recibir el correspondiente reconocimiento de forma correcta.
- *Modelo de errores*: para los resultados se ha utilizado, aparte de las pérdidas que puede darse por la congestión de los buffer de la red, un modelo de errores, el cual se le asignará un valor asociado al porcentaje de pérdida de paquetes que se obtienen. El

modelo de errores de ns-3 descarta los paquetes siguiendo una distribución uniforme. Este valor afecta en gran medida a TCP como se comprobará en futuras simulaciones.

- *Buffer TCP*: almacena los paquetes en el nodo origen de niveles superiores y en el destino a medida que se van recibiendo y se pueden recoger. Posteriormente, en caso de ser posible y no haya congestión en la red, los paquetes saldrán de la cola del buffer y se enviarán a la red. Hay que tener en cuenta que si el valor es muy pequeño, los paquetes se pueden perder porque no hay capacidad suficiente para almacenarlos. En situaciones ideales, lo correcto es utilizar un valor igual al BDP utilizado, sin embargo, en este proyecto se ha utiliza el doble, para ver hasta dónde puede llegar el algoritmo de CUBIC y que no se vea limitado por ello.
- *Buffer a nivel físico*: este buffer hace referencia a la capacidad que tienen los nodos de la red a nivel físico. Al igual que los buffer TCP, para evitar limitaciones, se utiliza de nuevo, el doble del valor BDP.

4.1.2. Proceso de medida

Tras haber descrito los parámetros fundamentales, se pasa a describir los conceptos necesarios para interpretar los resultados obtenidos en las simulaciones.

- *Ventana de congestión (CWND)*: este valor hace referencia a la cantidad de datos que un nodo puede emitir sin esperar a recibir un ACK. Puede representarse en varias unidades,por comodidad a la hora de interpretar los resultados, se representará en segmentos, por ello hay que tener en cuenta el tamaño del paquete utilizado. Para todas las conexiones, el tamaño máximo permitido es 1500 bytes. Partiendo de este valor, se ha adaptando el tamaño del paquete teniendo en cuenta las correspondientes cabeceras TCP e IP para que no haya fragmentación. El cálculo final de la ventana vendrá dado por el tamaño de la ventana en bytes, dividido por el tamaño del paquete.
- *Throughput*: es el valor que indica la eficiencia de la red. Este parámetro o se modifica en el propio simulador, como en los casos anteriores, es un cálculo externo que sigue la expresión siguiente:

$$\text{Throughput (Mbps)} = \frac{\text{Número de bits útiles recibidos (bits)}}{\text{Tiempo total de simulación (seg)}}$$

- *Utilización del canal*: este valor muestra cuánto se aprovecha la capacidad de la red disponible. Su cálculo está dado por la expresión:

$$\text{Utilización del canal (\%)} = 100 \cdot \frac{\text{Throughput (Mbps)}}{\text{Ancho de banda total (Mbps)}}$$

4.1.3. Cómo se han obtenido los resultados

Todas las simulaciones se han obtenido mediante el método de Montecarlo, repitiendo las simulaciones y representado la media de los resultados y el intervalo de confianza. El intervalo de confianza define el rango en el que se pueden encontrar las medidas obtenidas en las simulaciones. El cálculo se realiza para el 95 % de las medidas, siguiendo la expresión:

$$\text{Intervalo de confianza (Mbps)} = 2,2281 \cdot \frac{\sqrt{\text{Varianza de medias}}}{50}$$

4.1.4. Topología punto a punto

Este tipo de red, representada en la Figura 4.1 está propuesta básicamente para ver cómo se comporta TCP CUBIC.

Figura 4.1: Topología Punto a Punto

La red está formada solamente por N1 y N2. El nodo fuente (N1) utiliza la aplicación Bulk Sender de ns-3 generar información, mientras que el nodo destino (N2) usa la aplicación Packet Sink, también de ns-3. Ambas aplicaciones están definidas en la Sección 3.2. Este modelo se ha propuesto para ver las diferentes características propias de CUBIC ante una red de alta velocidad y alta latencia. Para evaluar el comportamiento de CUBIC se ha utilizado un canal de 500 Mbps y 400ms de RTT, lo que se puede considerar una red con un BDP alto. Inicialmente se propone un modelo sin errores y posteriormente se han ido añadiendo errores al canal para ver cómo se comporta, yendo desde pérdidas del 0.001 % de los paquetes, hasta el 0.01 %. Como se ha comentado anteriormente, el protocolo TCP es vulnerable ante un modelo de errores, por lo que con estas tasas, la eficiencia se verá muy afectada.

Para esta red se va a analizar cómo es la evolución temporal de la ventana de congestión. Además se realizarán hasta 50 simulaciones para ver cual es el throughput obtenido.

4.1.5. Topología Dumbbell

El siguiente experimento se ha realizado para ver cómo actúa CUBIC al compartir el canal con otros flujos en la red. Además se comparará los diversos resultados con New Reno. Para ello, con la ayuda del simulador ns-3, se ha desplegado un escenario siguiendo una topología *Dumbbell* mostrada en la Figura 4.2.

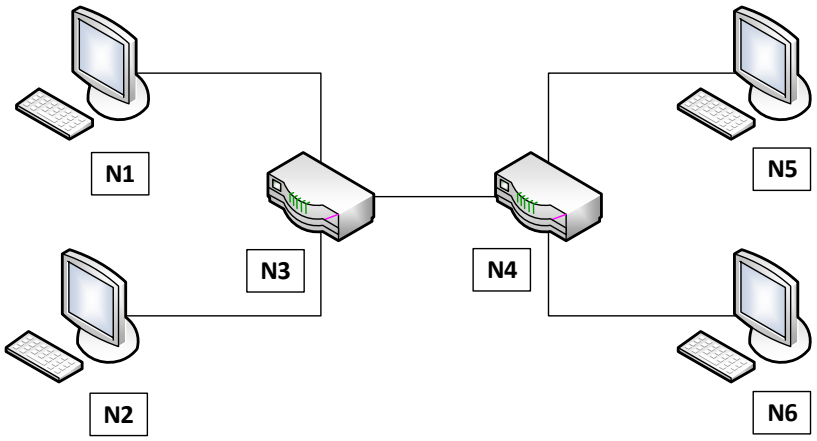


Figura 4.2: Topología Dumbbell

La red está formada por 6 nodos. En un extremo están los nodos fuente (N1 y N2) y en el otro, los nodos destino (N5 y N6). Entre ambos extremos se formará el cuello de botella de la red. Se puede observar cómo el cuello de botella se produce entre los nodos N3 y N4 al tener que compartir el ancho de banda entre los dos flujos. Los nodos intermedios se consideran *Drop Tail Routers*, lo que significa que siguen el algoritmo de *Drop Tail*. Este algoritmo permite gestionar de forma sencilla los paquetes que almacenan en su buffer siguiendo una estructura First In, First Out (FIFO), donde el primero que entra es el primero que sale, independientemente del lugar de origen del paquete. Para todos los experimentos realizados, cada nodo fuente envía información a un único nodo destino, de tal forma que N1 envía su información a N5, mientras que N2 lo hace a N6. Al igual que en la topología *Point to Point*, los nodos fuente utilizan la aplicación Bulk Sender y los nodos destino, Packet Sink.

En relación a los datos utilizados, la conexión punto a punto entre cada par de nodos extremos con los nodos intermedios tiene un RTT de 0,2ms. En el cuello de botella se propone un valor de RTT de 0.2ms para ver cómo se comportan ambos protocolos ante un RTT bajo y posteriormente se usará 200ms de RTT para obtener los mismo resultados ante uno alto. Las capacidades de todas las conexiones van a tener el mismo valor, variarán entre 50, 100 y 500 Mbps. La diferencia es que en la zona central tendrán que compartir el ancho de banda para los dos flujos y en los extremos tienen una conexión propia para cada uno. Por último lugar, hay que destacar también los datos utilizados en los buffer, tanto en los de TCP como en los físicos. Ambos dependen del RTT y de la capacidad de

la red. Se hará un producto entre ambos (BDP) y se multiplicará por 2 para asegurar que los valores obtenidos van a ser los máximos que van a poder dar CUBIC y New Reno.

Los primeros resultados compararán el comportamiento de la ventana de congestión de CUBIC y New Reno a diversas situaciones. Finalmente también se analizará la utilización de la red, para ver cuánta capacidad del canal es desaprovechada.

4.1.6. Topología lineal con conexión inalámbrica

Con este tercer escenario se va a dar un paso más. Hasta ahora, los dos anteriores escenarios han sido diseñados para ver cómo se comporta CUBIC ante redes cableadas. Con esta nueva topología se analizará cómo se comporta ante redes inalámbricas, realizando también una comparativa con protocolos como New Reno y Westwood.

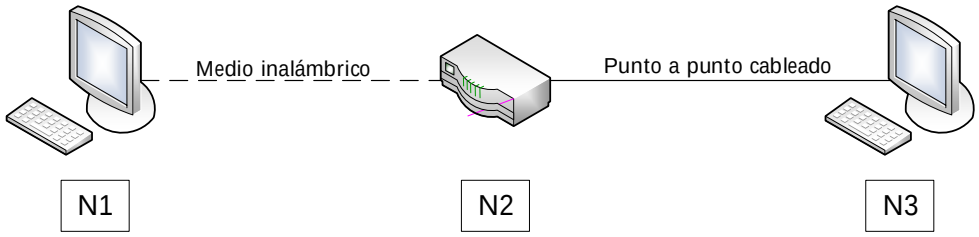


Figura 4.3: Topología 3 nodos con conexión inalámbrica

Como se observa en la Figura 4.3, la red está formada por 3 nodos (N1, N2 y N3). El nodo fuente (N1) y el nodo destino (N2) están comunicados mediante un nodo intermedio. Por un lado, N1 se conecta a él mediante una conexión inalámbrica, lo que simularía una conexión entre un ordenador y un punto de acceso. Por otro lado, tenemos una conexión cableada entre N2 y N3 que simula un entorno más real, donde el punto de acceso (N2) se comunica con un servidor lejano (N3). El nodo fuente, utiliza la aplicación Bulk Sender de ns-3, mientras que el nodo destino, usa Packet Sink, como en las anteriores topologías.

En relación a los parámetros utilizados, la conexión cableada está diseñada con un RTT de 200ms y un ancho de banda de 500 Mbps. Mientras que la conexión inalámbrica está basada en 802.11b, proporcionando una velocidad de 11 Mbps en la red. Como en el anterior escenario, se calculan los resultados con y sin un modelo de errores. Para este escenario, el modelo de errores implica una pérdida del 0.01 % y 0.02 % de los paquetes.

Para realizar un estudio más profundo, se realiza también una comparación con los protocolos New Reno y Westwood. En la Sección 2.1.6 se ha descrito que Westwood actúa de manera a ante redes inalámbricas, por lo que es interesante estudiar qué diferencias tiene con CUBIC.

4.2 Simulaciones

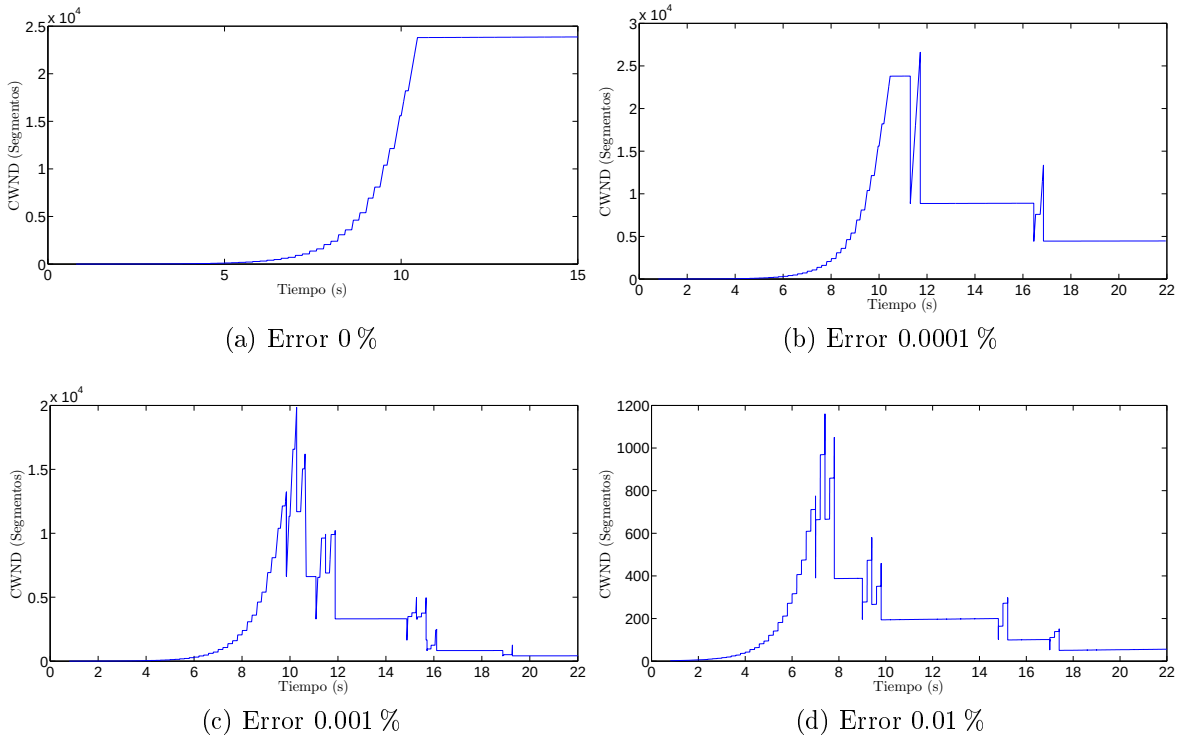
Para profundizar el estudio de las características del protocolo CUBIC, se han realizado diferentes simulaciones, partiendo de los escenarios definidos en la Sección 4.1.

4.2.1. Análisis de TCP-CUBIC

Esta sección hace referencia a la topología *Point to Point* de la Figura 4.1. El conjunto de parámetros que se van a proponer toman un valor de 500 Mbps para el ancho de banda, 400ms de RTT y un intervalo de [0 0.01 %] de error por paquete.

Ventanas de congestión

Como CUBIC está diseñado para redes con un alto ancho de banda y retardo, primero se evalúa la topología *punto a punto* sobre un enlace de 500 Mbps y 400ms de RTT. La ventana de congestión para diferentes tasas de error esta representada en la Figura 4.4.



(a) Error 0 %

(b) Error 0.0001 %

(c) Error 0.001 %

(d) Error 0.01 %

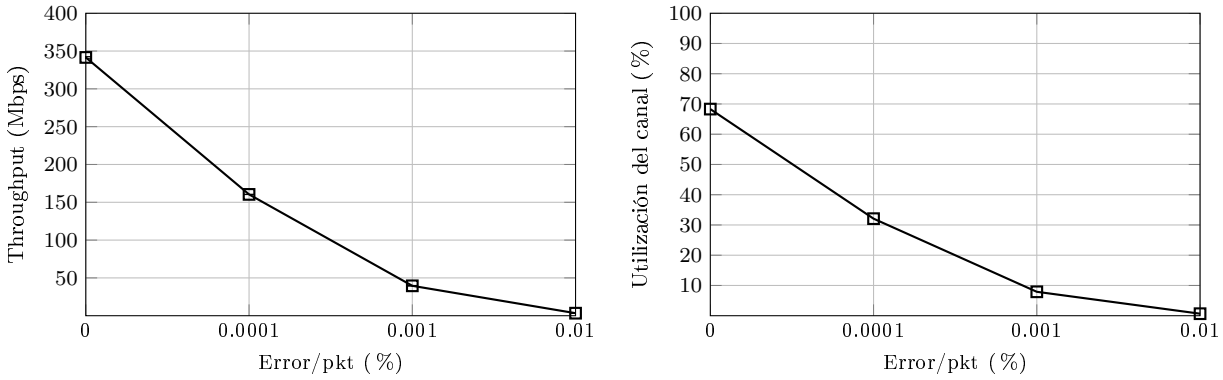
Figura 4.4: Ventanas de congestión con 500 Mbps y 400 ms de RTT utilizando topología Point to Point

En el caso específico de la Figura 4.4a, no hay modelo de errores y la ventana de congestión obtenida muestra un caso en el que no hay ninguna pérdida, ni siquiera por congestión de los buffer de la red, por eso no se produce ninguna caída. La interpretación de por qué sigue esta evolución se describe en la Sección 4.2.2 que muestra la comparativa de CUBIC con New Reno. En el caso con un error del 0.0001 % de los paquetes, Figura 4.4b, se aprecian caídas de la ventana de congestión, que explican el comportamiento típico de CUBIC. En la zona que abarca el intervalo de entre los 16 y los 18 segundos se ve claramente la forma cúbica que propone el algoritmo. En las demás situaciones (Figura 4.4c y Figura 4.4d), los resultados son parecidos, la forma de la ventana es similar, aunque el número de segmentos alcanza valores más altos en el caso del 0.0001 % de los errores.

Eficiencia de la red

Una vez analizadas las ventanas de congestión, se pasa a evaluar los resultados del rendimiento de la red y la utilización del canal. Por un lado, en la Figura 4.5a se observa cómo el throughput decae cuánto mayor es la probabilidad de error, lo cual pone de manifiesto el mal comportamiento de TCP sobre canales propensos a errores.

En relación a la utilización del canal (Figura 4.5b) se observa que el modelo sin errores, no ocupa todo el ancho de banda ya que se tiene un RTT muy alto y el crecimiento de la ventana es lento.



(a) Throughput por error

(b) Utilización del canal

Figura 4.5: Experimento: 500 Mbps, RTT 400 ms en una topología Point to Point

4.2.2. TCP-CUBIC vs TCP-NewReno

Una vez explicadas las características principales de CUBIC, se va a realizar un estudio más profundo comparándolo con la versión de TCP New Reno, ya que en anteriores

capítulos se ha comentado que comparte características con él, al ser uno de los algoritmos de congestión básicos del protocolo TCP. Para ello, se ha utilizado la topología Dumbbell explicada en la Sección 4.1.

Ambos protocolos empiezan transmitiendo paquetes a un modo conocido como es *Slow-Start* hasta que alcanzan un valor umbral (*ssthresh*), se produce una pérdida o un *timeout*. En la Figura 4.6 se puede ver esta característica, donde la función de New Reno superpone a la CUBIC. Al iniciarse la conexión, el valor de la ventana es igual a 2MSS (por defecto en el simulador), una vez llega el ACK correspondiente, la ventana tomará valor de 4MSS, cuando lleguen siguientes ACK's, la CWND se actualizará al doble y así sucesivamente hasta que se detecte una pérdida o se alcance el umbral de congestión mencionado. En general el valor *ssthresh*, es por defecto 65536 bytes, así que en redes con alto BDP este valor se alcanza en un tiempo muy corto, como se puede observar.

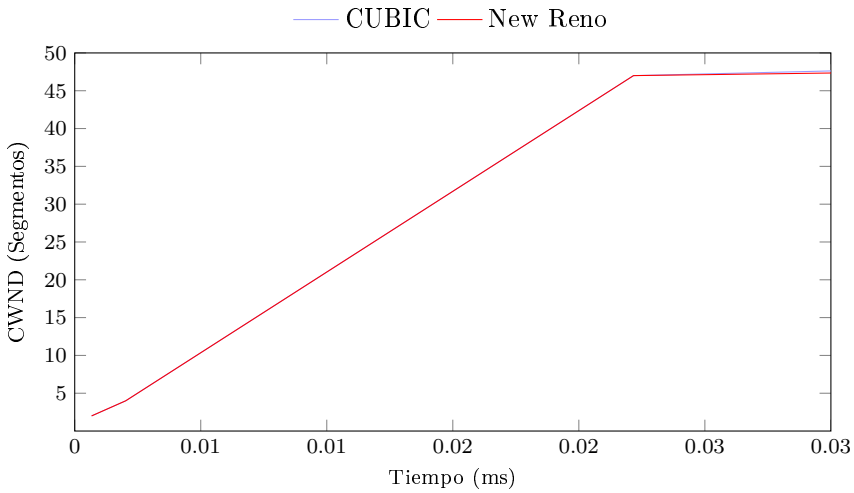


Figura 4.6: Análisis del modo *Slow-Start* de uno de los flujos en la topología Dumbbell.

Una vez finalizado este modo, por cada ACK recibido se calcula el valor de W_{cubic} y los sucesivos de la forma $W_{cubic}(t + RTT)$. Dependiendo del resultado obtenido se puede trabajar en uno de los 3 modos restantes, como se explica en la Sección 2.2. En caso de tener una red con un valor de BDP bajo, CUBIC va a adaptar el crecimiento y decrecimiento de su ventana lo más parecido a New Reno para obtener rendimientos aceptables ante diferentes topologías de red. Esta situación conlleva al modo *TCP Friendly* de CUBIC, explicado en la Sección 2.2.1).

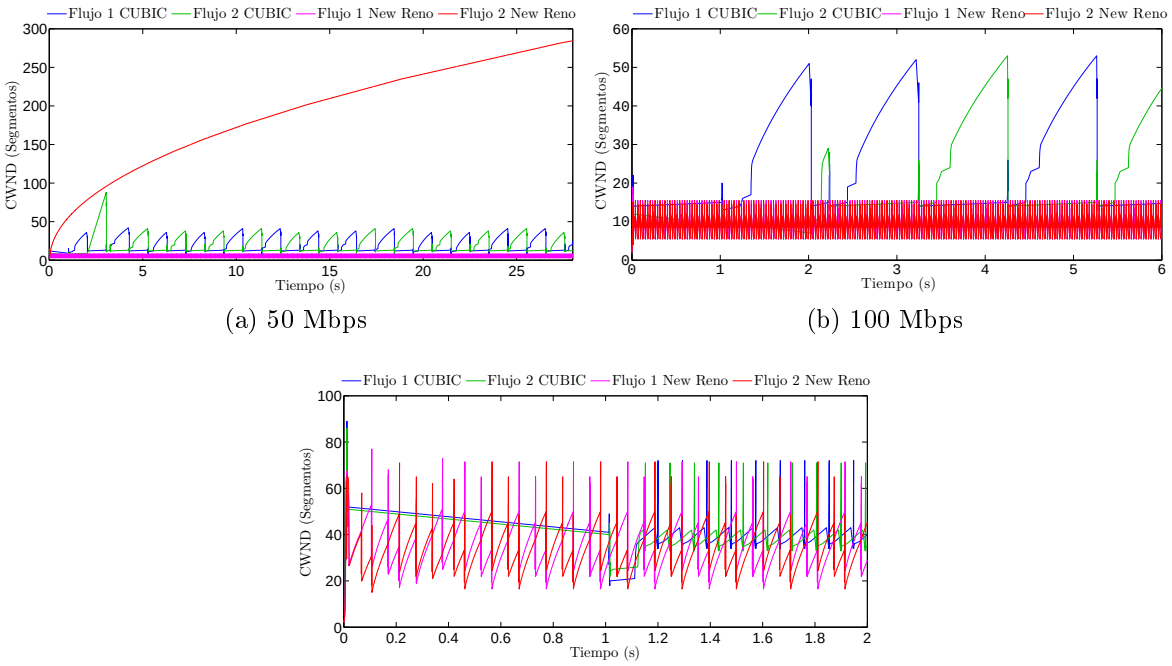
Otra de las semejanzas con New Reno es si se da la situación de recibir un triple ACK con el mismo número de secuencia o se ha producido un *timeout*. Cuando sucede esto, se indica que hay que realizar una retransmisión, para ello se activa el modo *Fast-Recovery*. Lo que hace es iniciar la retransmisión de todos aquellos paquetes previos al número de secuencia de los 3 ACK's recibidos. Una vez iniciado el *Fast-Recovery* podemos

recibir dos tipos de ACK. $ACK_{partial}$ es aquel que llega de los paquetes que aún no han sido reconocidos, pero todavía no es el que tiene el número de secuencia que indicaba la congestión. Este simplemente indica que aún hay que seguir retransmitiendo porque no se ha llegado al definitivo. Por otro lado, ACK_{full} indica que todos los paquetes previos que se tenían que retransmitir, se han enviado de forma correcta, poniéndose fin a este modo.

Sin embargo, los demás modos de operación, no funcionan igual. Como se ha explicado en la sección 2.2, CUBIC propone una evolución de la ventana totalmente diferente. Como se puede ver en la figura 2.4.

Resultados con RTT's bajos

Al igual que en la sección anterior, se han modificado 3 parámetros esenciales, en este caso el RTT es de 0.2ms en todos las conexiones, formando en total un RTT total en la red de 0.6ms. El ancho de banda es un parámetro que varía entre 50, 100 y 500 Mbps. El tercer parámetro a tener en cuenta son los errores. En este caso, se obtienen resultados sin ningún modelo de error y posteriormente se aplica un error del 0.01 % de los paquetes.



(a) 50 Mbps

(b) 100 Mbps

(c) 500 Mbps

Figura 4.7: Ventanas de congestión con RTT 0.6ms - Sin Errores ante una topología Dumbbell

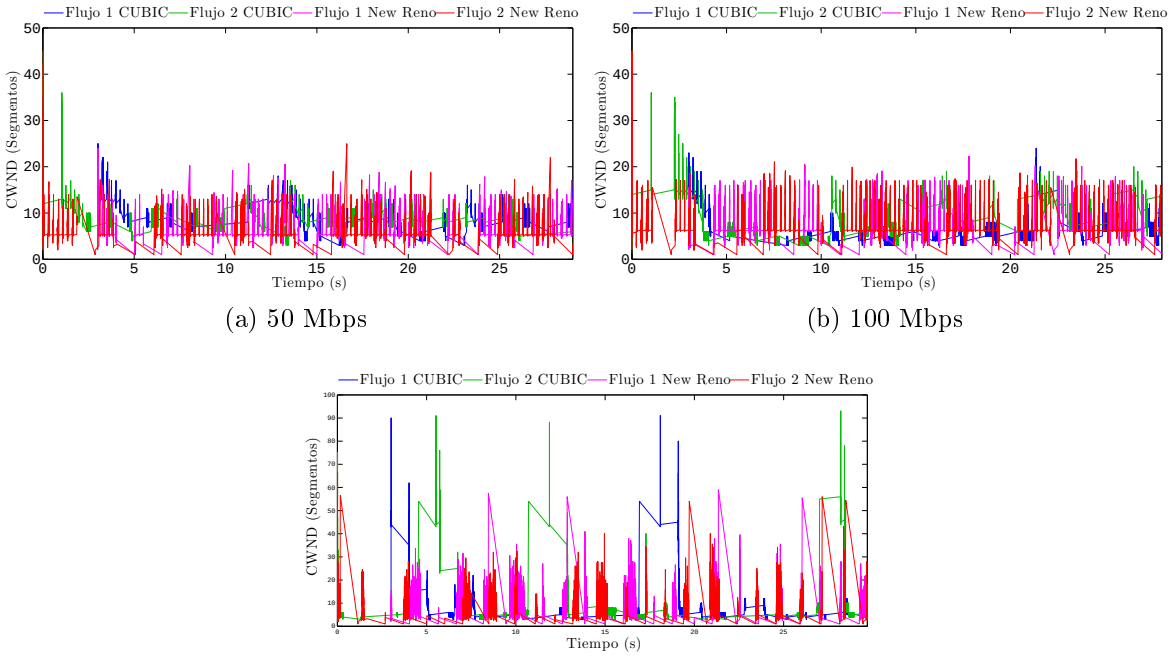
Ante el modelo sin errores representado en la Figura 4.7, las únicas pérdidas que pueden surgir vienen puramente de la congestión de los buffer de la red. En este caso, al

solo tener pérdidas por congestión, se ve claramente lo justo que es CUBIC a la hora de compartir el canal como se ha explicado en la Sección 2.2. En la Figura 4.7a se observa cómo CUBIC al sufrir alguna pérdida por congestión permite adaptarse al segundo flujo a la red. En cambio, el algoritmo New Reno no le ocurre lo mismo, el flujo 2 no sufre ninguna pérdida por lo que el crecimiento de la ventana es continuo, impidiendo que el flujo 1 alcance valores superiores, por eso apenas se puede observar la evolución del primero. Esto es un claro ejemplo de cómo se comportan ambos algoritmos de congestión en el caso de añadir más de un flujo a un mismo canal. Sin embargo, en una situación real, lo ocurrido en New Reno no va a prolongarse hasta el infinito y el primer flujo podrá adaptarse mejor al canal.

En la Figura 4.7b se observa la evolución con un ancho de banda de 100 Mbps. Las características observadas en CUBIC son similares a las de 50 Mbps, aunque el algoritmo New Reno sí que sufre pérdidas por congestión y ambos flujos alcanzan valores similares. En el caso de 500 Mbps, se ha hecho una reducción de la ventana de simulación a 2 segundos, puesto que ambos protocolos siempre seguían la misma forma que en la figura 4.7c pero de una forma más prolongada, donde no se apreciaban casi las diferencias entre flujos. Con este experimento se puede concluir que CUBIC no se diferencia en gran medida a New Reno, puesto que se está frente a una red con BDP bajo, lo que indica que se utiliza el modo *TCP Friendly* para obtener mejores características. Además cabe destacar que a mayor ancho de banda, más similar es la forma de su ventana a New Reno, como se puede observar en la figura 4.7c. Cabe destacar que CUBIC trabaja bien en redes de alta velocidad y con RTT altos como se ha mencionado en secciones anteriores, con el uso de un ancho de banda de 500 Mbps, el valor el producto BDP comienza a tomar valores grandes, sin embargo, hay que tener en cuenta la combinación de ambos factores, es decir, por tener un valor muy alto de ancho de banda que compense el RTT no va a hacer que CUBIC salga del modo *TCP Friendly* para usar el de su propio algoritmo.

Por otro lado, se ha utilizado un modelo de errores con una pérdida del 0.001 de los paquetes, representado en la Figura 4.8. Los resultados obtenidos son esperados, se está ante una situación con un BPD bajo, por lo que TCP CUBIC va a seguir una evolución parecida a la de TCP New Reno, como se ha explicado en la sección 2.2.1. Para obtener mejores resultados, se realiza el cálculo del valor *cnt* con la ventana de congestión de New Reno (W_{tcp}), en vez de a forma cúbica que sigue la CWND de CUBIC (W_{cubic}).

Tras analizar las ventanas de congestión de ambos protocolos, se va a estudiar el rendimiento de la red en las anteriores situaciones. Los resultados obtenidos se muestran en dos gráficas, por un lado el modelo sin errores y por otro, el modelo con errores. Cada gráfica muestra el throughput de la red para un RTT de 0.6ms y los tres valores de ancho de banda propuestos anteriormente. Los datos obtenidos en relación a las ventanas de congestión estudiadas previamente son esperados. Ante el modo sin errores representado en la Figura 4.9, la eficiencia del canal es máxima en ambas situaciones (Figura 4.9a), ocupando el máximo del canal posible. En la utilización del canal (Figura 4.9b), se puede



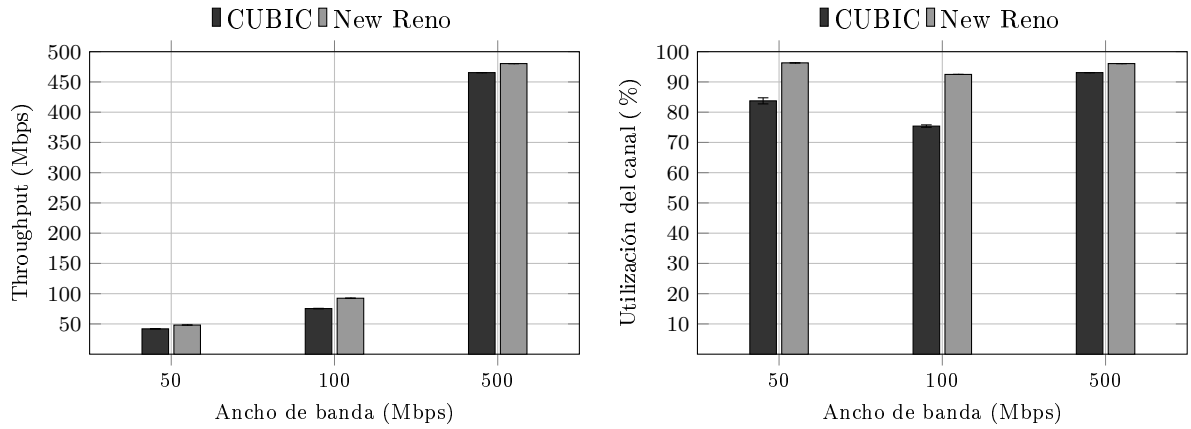
(a) 50 Mbps

(b) 100 Mbps

(c) 500 Mbps

Figura 4.8: Ventanas de congestión con RTT 0.6ms - Error 0.01 % ante una topología Dumbbell

ver nuevamente que a mayor ancho de banda, más se parecen ambos protocolos, por ello en a los 500 Mbps, el valor del throughput es prácticamente el mismo.



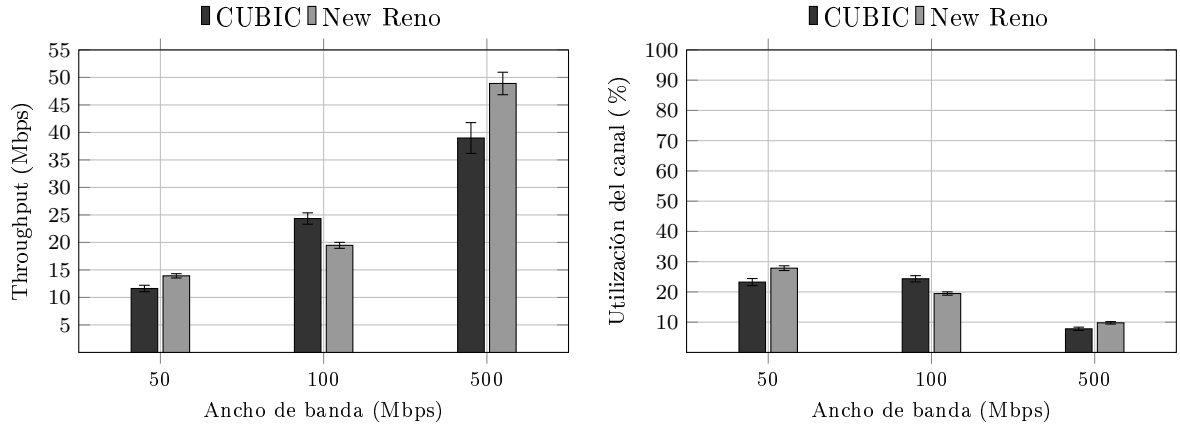
(a) Throughput

(b) Utilización del canal

Figura 4.9: Experimento.s RTT 0.6 ms - Sin Errores ante una topología Dumbbell

La figura 4.10 muestra los valores de rendimiento de la red con un 0.01 % de pérdida de paquetes. Los resultados obtenidos no son muy dispares, puesto que como se ha explicado en el análisis de la ventana de congestión, el producto del RTT y el ancho de banda es aún bajo y CUBIC no utiliza su propio algoritmo que permite obtener mejores rendimientos. Aunque cabe destacar que ente esta situación, a CUBIC le afecta por el igual los errores, independientemente del ancho de banda, por eso en la Figura 4.10a, la eficiencia sigue una línea recta, a diferencia de New Reno. Además, es importante destacar que con un modelo de errores y con RTT bajos, CUBIC se ve ligeramente más afectado que New Reno. Otro dato importante a observar, es que el intervalo de confianza en CUBIC es mayor, esto es debido a que este algoritmo depende principalmente del tiempo en el que se produce la última congestión, en cambio New Reno, aumenta siempre la ventana por cada RTT recibido, independientemente de la última caída.

En cuanto a la utilización del canal (Figura 4.10b), los resultados obtenidos ante el modelo de errores, se ven mayormente afectados para anchos de banda mayores, ya que la tasa de envío es mayor y por lo tanto la probabilidad de perder más paquetes aumenta también.



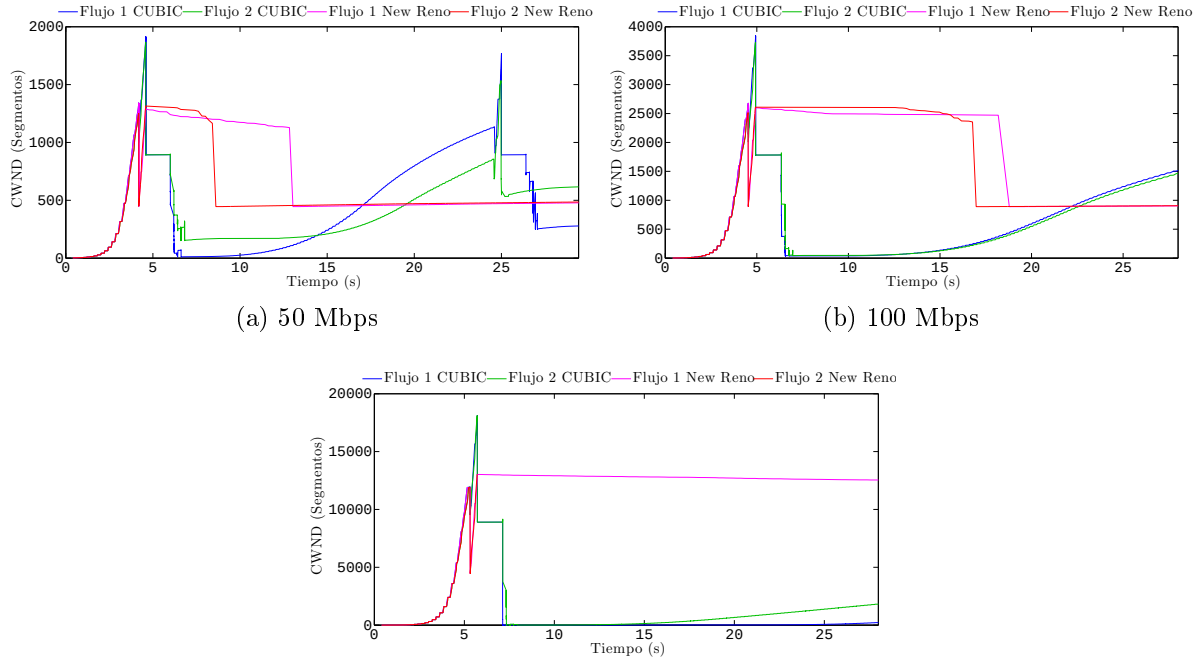
(a) Throughput (b) Utilización del canal

Figura 4.10: Experimento: RTT 0.6 ms - Error 0.01 % ante una topología Dumbbell

Resultados con RTT's altos

Una vez propuesto una comparativa de ambos protocolos ante valores bajos de RTT, se va a realizar un nuevo experimento, el cual sigue la misma topología, pero en este caso se incrementa el valor del RTT en el enlace, entre los nodos N3 y N4 que forman el cuello de botella. Es en esta parte donde se puede observar las principales diferencias entre New Reno y CUBIC, puesto que TCP CUBIC está ideado para este tipo de topologías. Para este experimento se ha utilizado un RTT de 200.4ms (200ms del cuello de botella y 0.4ms de

las conexiones extremas). El ancho de banda varía entre 50, 100 y 500 Mbps, proponiendo de nuevo un modelo con y sin errores para todos los casos descritos.



(a) 50 Mbps

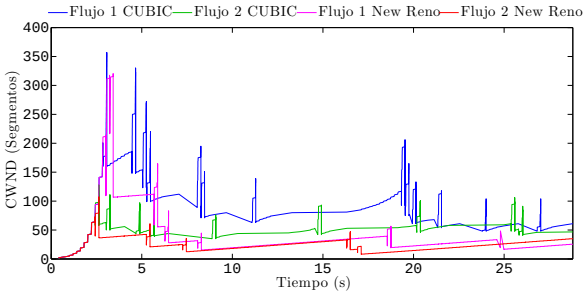
(b) 100 Mbps

(c) 500 Mbps

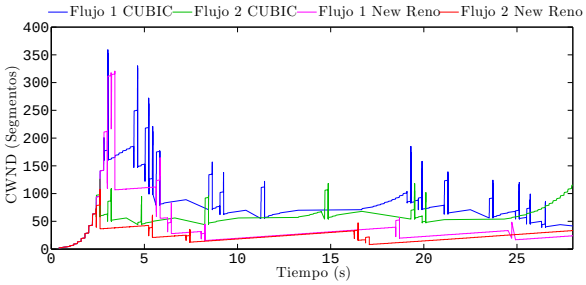
Figura 4.11: Ventanas de congestión con RTT 200.4ms - Sin Errores ante una topología Dumbbell

En el caso sin errores, las evolución de las ventanas de congestión están representadas en la Figura 4.11, donde se puede observar de una forma más visual la forma de CUBIC, puesto que el RTT es muy grande y la evolución de la ventana es más lenta. Además no ocurre como en la sección 4.2.1, las simulaciones mostradas ya presentan pérdidas por congestión en los buffer, permitiendo ver la evolución cúbica típica del algoritmo de congestión de CUBIC. Este proceso se puede ver muy bien a 50 Mbps, representado en la Figura 4.12a, ya que el tiempo de simulación permite ver desde que comienza la caída alrededor de los 6 segundos, hasta la siguiente, que se produce sobre los 24 segundos. El incremento del ancho de banda, a 100 Mbps (Figura 4.12b) y 500 Mbps (Figura 4.12b), no supone grandes diferencias, las evolución es prácticamente similar, salvo que a mayor ancho de banda, los valores de la CWND son más altos, como es lógico.

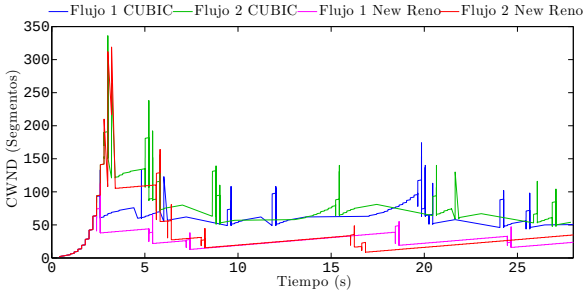
Por otro lado, el caso con errores, representado en la figura Figura 4.12, muestra una evolución de los flujos de CUBIC algo mayor a New Reno. Para los tres anchos de banda propuestos, los resultados apenas son diferenciables, las evolución de la ventana es prácticamente igual, concluyendo que el uso de un modelo de errores ante un cuello de botella de alta latencia afecta en gran medida cuanto mayor es el ancho de banda.



(a) 50 Mbps



(b) 100 Mbps



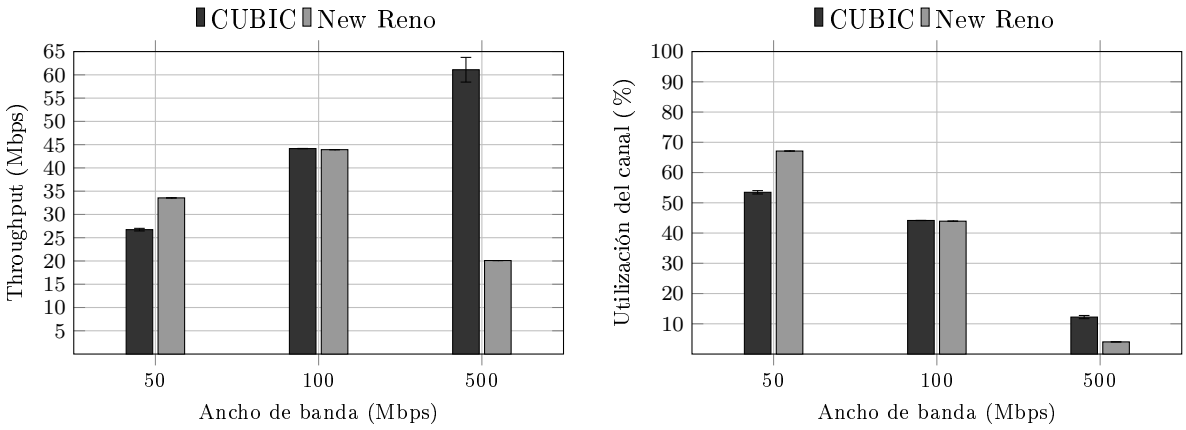
(c) 500 Mbps

Figura 4.12: Ventanas de congestión con RTT 200.4ms - Error 0.01 % ante una topología Dumbbell

En relación a los valores de throughput obtenidos, en modelo sin errores (Figura 4.13) se observa una clara diferencia a los 500 Mbps entre los dos protocolos, ya que se considera una red de alta velocidad y de alta latencia. Sin embargo, para valores de ancho de banda más bajos, las diferencias no son tan notorias, al igual que ocurre en las ventanas de congestión.

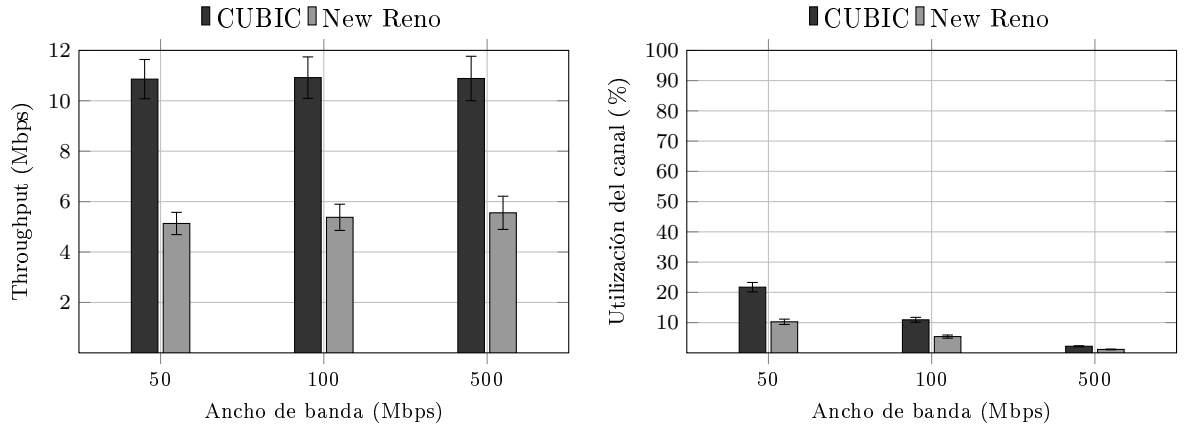
En la figura 4.14 se representa la eficiencia de la red ante el modelo con errores, con un 0.01 % de pérdida de los paquetes. En la Figura 4.14a se ve una clara diferenciación de throughputs entre New Reno y CUBIC para cualquier valor de ancho de banda. Tanto a 50 Mbps, como a 100 y 500 Mbps, la eficiencia de la red es casi prácticamente el doble con el uso de CUBIC.

A diferencia del modelo sin errores, para todos los valores de ancho de banda propuestos, se ve una mejora del protocolo CUBIC ante New Reno, esto es debido a que CUBIC fuera del modo *TCP Friendly* trata mejor los errores porque la evolución de su ventana depende del momento en el que se produjo la última pérdida. En cambio, New Reno no tiene esta característica, este disminuye su ventana al detectar una pérdida o se produce un *timeout* y sigue la evolución de la ventana al modo correspondiente. Por este mismo motivo los intervalos de confianza son mayores en CUBIC, ya que el tiempo entre una congestión y otra tiene un carácter más aleatorio.



(a) Throughput (b) Utilización del canal

Figura 4.13: Experimento: RTT 200.4 ms - Sin Errores ante una topología Dumbbell



(a) Throughput (b) Utilización del canal

Figura 4.14: Experimento: RTT 200.4 ms - Error 0.01 % ante una topología Dumbbell

Otra característica a observar es que el aumento del ancho de banda con un RTT alto afecta de diferente forma en el rendimiento de la red cuando se propone un modelo de errores. Como ocurre con el crecimiento de la ventana de congestión, para 50, 100 y 500 Mbps los rendimientos representados en la Figura 4.12 son semejantes, por lo que cuanto mayor ancho de banda se utilice, menor utilización del canal se obtendrá, como se puede observar en la Figura 4.14b.

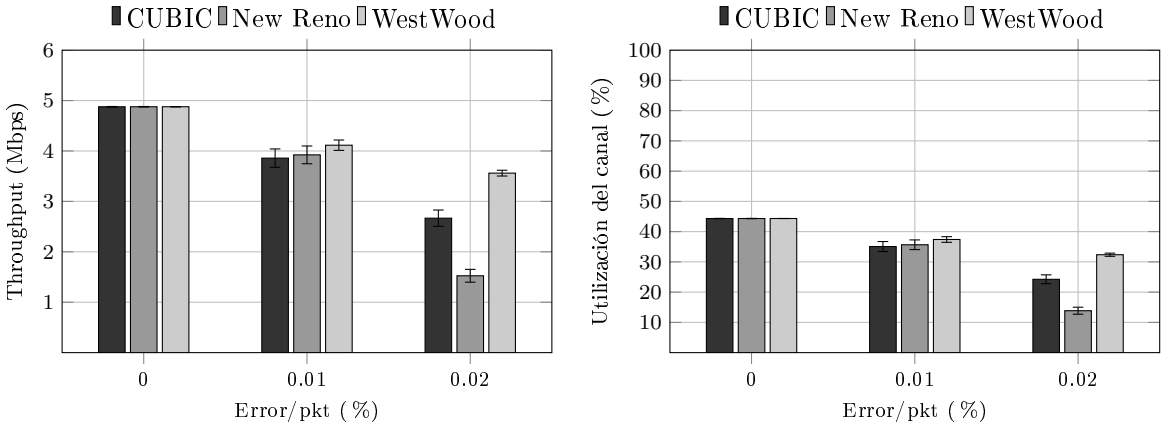
4.2.3. TCP-CUBIC ante medios inalámbricos

Para el análisis de resultados de esta parte, se ha utilizado la topología lineal explicada en la Sección 4.1.6. Las ventanas de congestión en este caso no suponen diferencias a los anteriores casos, por lo que los resultados se han centrado principalmente en obtener los valores de throughput y utilización del canal.

En la Figura 4.15 se pueden ver los rendimientos obtenidos. Ante un modelo sin errores, los valores obtenidos se sitúan en torno a 5 Mbps, que es el valor de throughput máximo obtenido en redes inalámbricas utilizando el protocolo TCP. Frente una pérdida del 0.01 %, las diferencias son apenas notorias, obteniendo throughputs semejantes. A medida que aumentan los errores por paquete, el throughput obtenido comienza a ser diferente entre los tres protocolos. Es en el caso de una pérdida del 0.02 % cuando se observa mayores diferencias. El protocolo CUBIC destaca sobre New Reno. Como ya se conoce, CUBIC se adapta mejor ante canales propensos a error, obteniendo mejores rendimientos debido a la evolución de su ventana dependiente del momento que se produjo la última pérdida.

Cabe destacar la eficiencia del protocolo Westwood. Si se compara con New Reno, las mejoras son claramente observable, obteniendo el doble de eficiencia con el uso de Westwood. En cambio, en relación al protocolo CUBIC, el throughput obtenido en Westwood supone una cuarta parte más que el obtenido por el propio CUBIC. Como consecuencia, hay que destacar que hay algoritmos de congestión que ofrecen métodos más inteligentes que CUBIC, como es en el caso de Westwood para redes inalámbricas.

En este escenario, la conexión limitante es la inalámbrica, por eso la utilización del canal, representada en la Figura 4.15b está en función de los 11 Mbps que ofrece la conexión



(a) Throughput

(b) Utilización del canal

Figura 4.15: Experimento: Inalámbrico 802.11b - Cableado: RTT 200 ms y 500 Mbps ante una topología lineal

802.11b. Como se ha comentado anteriormente, el uso del protocolo TCP reduce la eficiencia del canal WiFi, por eso ante los modelos representados, no se supera el 50 % del canal, aunque el que ofrece mejores características es Westwood para los tres modelo de errores propuestos.

5

Conclusiones y líneas futuras

5.1 Conclusiones

Este último capítulo concluye la memoria, destacando los aspectos más importantes que se han extraído tras analizar los resultados y simulaciones realizadas en capítulos previos.

El protocolo TCP es uno de los más utilizados en las redes actuales a nivel de Transporte. Como consecuencia de la evolución de las tecnologías, se han ido complementando a este protocolo nuevas características para lograr mejores rendimientos. Una de los principales aspectos que define TCP es el control de la congestión. A lo largo de los años, estos algoritmos han ido actualizándose para soportar diferentes topologías de red, destacando New Reno. Sin embargo, hay variantes que ofrecen características mejores, dejando atrás las versiones originales.

CUBIC responde de una forma óptima ante redes con altos valores de ancho de banda y latencia. Por otro lado, en una situación real, la probabilidad de pérdida está siempre presente, como una gran influencia en el protocolo TCP, como se ha podido comprobar. Si a las redes de alto ancho de banda y latencia se le suma una probabilidad de pérdida considerable, CUBIC también muestra claras ventajas y ofrece hasta un 60 % más de rendimiento ante algoritmos de congestión más antiguos, como New Reno.

Cabe destacar que CUBIC aumenta la ventana de congestión en función del momento en el que se produjo la última pérdida, y debido al carácter aleatorio de la probabilidad de pérdida, el intervalo de confianza de las medias del throughput en la mayoría de casos es mayor al de New Reno, que utiliza un método más sistemático en la evolución de su ventana.

De todos modos, el uso de TCP CUBIC no va a suponer siempre una ventaja. También hay una extensión importante de redes donde el ancho de banda es menor, al igual que la latencia. En este caso, TCP CUBIC se adapta de la mejor forma posible al protocolo

estándar New Reno, para obtener rendimientos aceptables ante cualquier tipo de red.No obstante, en la actualidad, el despliegue de redes de alta velocidad es cada vez mayor, siendo este un punto a favor para el protocolo CUBIC en desarrollos futuros.

Se han desarrollado comparativas también con otro tipo de protocolos más recientes, como es el caso de TCP Westwood, el cual propone un modelo que ofrece mejores rendimientos ante conexiones con una tasa elevada de errores en el canal, como es el caso de las conexiones inalámbricas. Así en canales propensos a errores, los valores de eficiencia son hasta un 25 % mejores en el caso de Westwood, en comparación con CUBIC. Esto demuestra que la evolución de los algoritmos de congestión, suponen en la mayoría de casos grandes ventajas, por lo que es conveniente fomentar el uso de estos nuevos protocolos y evitar el estancamiento en los más antiguos.

Con respecto al simulador ns-3, es importante destacar que el algoritmo de congestión CUBIC, no está implementado por defecto en su estructura, únicamente incorpora otros algoritmos básicos como son Reno y New Reno, aunque en versiones posteriores también se ha añadido TCP Westwood.

5.2 Líneas futuras

A raíz del trabajo realizado en este proyecto han surgido diferentes puntos que sería interesante analizar en futuras investigaciones.

- Se ha estudiado cómo se comporta CUBIC ante diferentes escenarios con conexiones cableadas, como la topología punto a punto o Dumbbell, sin embargo, convendría profundizar en el análisis en redes más complejas. Un ejemplo podría ser ver qué rendimientos se obtienen si se incrementa el número de flujos compartiendo el mismo canal o el uso de canales más propensos a error.
- Ampliando el estudio realizando y la correspondiente comparativa de TCP CUBIC con otras versiones ante redes inalámbricas, una opción propuesta sería también realizar un despliegue de un escenario más completo, utilizando versiones más actuales, como 802.11ac, añadiendo más nodos a la red o ampliando el intervalo de probabilidad de error en el canal.
- En relación al simulador ns-3, sería conveniente estandarizar el protocolo CUBIC e incluirlo oficialmente en su arquitectura. ns-3 proporciona una completa variedad de herramientas, pero desde el punto de vista de los algoritmos de congestión, es algo escaso, obligando a instituciones externas a implementar los algoritmos adaptados a las continuas versiones del simulador que van surgiendo.

Bibliografía

[1] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-host congestion control for tcp. *IEEE Communications Surveys Tutorials*, 12(3):304–342, Third 2010.

[2] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[3] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[4] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000.

[5] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP Fast Recovery Algorithm. RFC 6582, October 2015.

[6] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[7] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining tcp congestion control. *SIGCOMM Comput. Commun. Rev.*, 26(4):281–291, August 1996.

[8] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM.

[9] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001.

[10] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[11] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[12] Ning Cao and Wei Zhang. Cubic with faster convergence: An improved cubic fast convergence mechanism. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013.

[13] Brett Levasseur, Mark Claypool, and Robert Kinicki. A tcp cubic implementation in ns-3. In *Proceedings of the 2014 Workshop on Ns-3*, WNS3 '14, pages 3:1–3:8, New York, NY, USA, 2014. ACM.

[14] David A. Borman, Robert T. Braden, and Van Jacobson. TCP Extensions for High Performance. RFC 1323, March 2013.

Lista de acrónimos

RTT Round-Trip Time

RTO Retransmission Timeout

SACK Selective Acknowledgment

CWND Congestion Window

BDP Bandwidth-Delay Product

AIMD Additive-increase multiplicative-decrease

RTO Retransmission timeout

MSS Maximum Segment Size

FIFO First In, First Out

ns-3 Network Simulator 3

Tcl Tool Command Language

OTcl Object Tool Command Language