Proyecto de Grado

Presentado ante la ilustre Universidad de Los Andes como requisito parcial para obtener el Título de Ingeniero de Sistemas

Un método para el ajuste de la velocidad de transmisiones cortas de datos en redes TCP/IP: Un enfoque por simulación

Por

Br. Eduardo A. Granados L.

Tutor: Dr. Andrés Arcia-Moret

Septiembre 2014

Un método para el ajuste de la velocidad de transmisiones cortas de datos en redes TCP/IP: Un enfoque por simulación

Br. Eduardo A. Granados L.

Proyecto de Grado — Sistemas Computacionales, 117 páginas

Resumen: En las redes de computadores, la simulación es utilizada para poder evaluar ciertas configuraciones (de cualquier nivel de complejidad) de las cuales se desea obtener información a detalle sobre el comportamiento de uno o más sistemas en particular. Existen simuladores de redes conocidos como NS-2 u Omnet++, cuya curva de aprendizaje puede ser difícil de abordar. Por ello, en este trabajo, proponemos el diseño y la implementación de un simulador sencillo de redes enfocado a tener un diseño fácil de comprender, mantener y escalar. Este simulador ha sido diseñado con propósitos didácticos, que a la vez permita modelar una red sobre la cual se puedan implementar cualquier protocolo de red con el propósito de explicarlo en el aula de clases. Como punto de partida, el simulador presentado, llamado microSim ($\mu\mathcal{SIM}$), permite realizar pruebas en redes con cuellos de botella para estudiar el comportamiento de los algoritmos de control de congestión que existen dentro de TCP, así como también desarrollamos la implementación de un método para acelerar la velocidad de transmisión de datos.

Palabras clave: Simulación de redes, TCP, Control de Congestión, divACKs, modelado de redes.

Este trabajo fue procesado en LaTeX.

Índice general

Índice de figuras

Capítulo 1

Introducción

El Protocolo de Control de Transmisión (TCP[1]), según [1] es utilizado en más del 90 % del tráfico que circula por Internet, el cual es generado por datos provenientes de aplicaciones como correos electrónicos, páginas web o intercambio de archivos, entre otros. TCP es además considerado como un elemento fundamental para Internet, pues es el protocolo que garantiza los servicios de intercambio de datos (transmisiones confiables) y un uso equitativo de los recursos de red.

Siendo TCP un protocolo con tal presencia en la Internet, es considerado fundamental en los cursos de redes impartidos en las universidades. A través del estudio de TCP se comprenden aspectos de diseño de redes tales como políticas de uso de recursos, tratamiento de pérdida de datos, establecimiento de sesiones entre dos puntos, ingeniería de protocolos, y seguridad, entre otros.

De esta manera, en un ambiente de simulación con propósito didáctico, se puede lleva a cabo la implementación de protocolos o simples mejoras a un algoritmo distribuido. Por esto, se ha convertido en buena práctica, modelar redes, y simular entornos, para estudiar su comportamiento bajo ciertas variantes previsibles en un ambiente real. Mientras más detallado sea el modelo de la red, más comparable será a la respuesta real.

Existen varios simuladores de redes en la actualidad, la utilidad de cada uno depende de la necesidad que se tenga, por ejemplo, NS-2 es lo suficientemente completo y versátil

[1] Transmission Control Protocol

para poder realizar pruebas de nuevas tecnologías, teorías o protocolos. Pero la curva de aprendizaje para propósito didáctico puede llegar a ser una tarea compleja [2]. Recientemente se ha propuesto NS-3, con un diseño completamente nuevo y con un estilo de programación más moderno. Sin embargo, la penetración de NS-3 es todavía relativamente baja respecto a NS-2, quizás debido a las diferencias en cantidad de protocolos implementados en NS-2. Otros simuladores como GloMoSim o SimulX son con propósito específico, invitándonos a desarrollar nuestro propio simulador de propósito didáctico y creado en base a la experiencia académica de los cursos de redes dictado en la escuela de ingeniería de sistemas.

En este documento proponemos un simulador con propósito didáctico, donde calculamos las conductas intermedias con los constructos mínimos necesarios para entender el rol de la capa transporte. Para ello hemos implementado TCP tomando en cuenta los algoritmos que dictan su comportamiento y especificados en [3], como también una variante sobre TCP para acelerar las transmisiones, basado en la técnica de *divacks* [4].

Proponemos entonces $\mu\mathcal{SIM}$ con la finalidad de ser un simulador simple de entender y lo suficientemente versátil para poder extender a otros escenarios y protocolos de red. Nosotros nos enfocamos en el estudio del control de congestión y de una variante para el aumento de la velocidad en las transmisiones cortas (ver Capítulo 4) que en ocasiones mejora el rendimiento de TCP, y estudiada a profundidad por Arcia-Moret en [5].

1.1. Antecedentes

TCP ha sufrido cambios incrementales desde su concepción a finales de los años 80 [RFC 793][3]. El protocolo tiene como función el control de la transmisión (confiable) de datos. Es decir, que lleguen de extremo a extremo en estricto orden. Para ello, el protocolo debe asegurar el orden de los datos o perder alguno de ellos. Además, el protocolo debe garantizar un uso equitativo del servicio de comunicación de datos, pues la concurrencia en el uso del servicio es fundamental para aspectos como la neutralidad de la Internet [6].

Para llevar a cabo la transmisión de datos, el protocolo cuenta con dos fases bien definidas. *Slow Start* para descubrir el ancho de banda disponible en la red y *Congestion Avoidance* para mantener la tasa de transmisión y coexistir (adaptarse) junto con otros flujos circulando en los distintos cuellos de botella de la red.

Hoy en día, debido al aumento del ancho de banda disponible en la red [4], compañías como Google, están proponiendo esquemas de envío de archivos pequeños (como las páginas web) de manera significativamente más rápida [7] aumentando el tamaño inicial de la ventana de congestión. Por otro lado, en redes de gran capacidad como las redes satelitales [8], se necesita un envío más agresivo de datos al principio de la conexión. Esto, tanto para descubrir más rápidamente la capacidad a utilizar de la red, como para ahorrar tiempo en los envíos de datos de gran capacidad y retardo muy largo.

En [5][4][9], Arcia-Moret y colaboradores, han estudiado un mecanismo basado en el aumento significativo de los paquetes de control (ACK[2]) de tal manera que se pueda regular la velocidad el envío de datos a voluntad. Este método de aumento de envío de ACKs, se conoce como *divacks* o ACKs divididos, los cuales se envían para reconocer de manera parcial un segmento recibido por el destino [10]. Teóricamente estos divacks tienen las mismas propiedades de un ACK completo, por lo tanto cuando un divack es recibido por el emisor, este hace crecer la ventana de congestión como si fuese un ACK completo, acelerando el crecimiento de la ventana de congestión. Esto es posible debido a la dualidad de la interpretación de los paquetes ACKs, aunque ellos reconocen la llegada de bytes, son interpretados como unidades completas y su verificación en este sentido es la mínima.

Hay entornos o contextos donde hace falta poner a prueba este método con propósito específico de estudiar su conducta. En la literatura, hay casos donde se hace uso de simulación de sistemas. Dhraief et al. [11] realizan un estudio para medir, comparar y evaluar el desempeño en Sistemas de Distribución Inalámbricos (WDS)[3] usando la herramienta NS-2, obteniendo resultados bastante fiables sin utilizar equipos reales. Por otro lado, Hachana et al. [12], realizan un estudio comparativo de dos simuladores

[2]Acknowledgement Packet: Paquete de reconocimiento enviado por el receptor de un segmento enviado por el emisor

[3]Interconexión de puntos de acceso por medios inalámbricos

conocidos (NS-2 y OMNeT++) y propone el desarrollo de una nueva herramienta para simular entornos inalámbricos llamado SimulX, un simulador por eventos discretos. En [13], Lezama et al. desarrollan SIMDROP, un simulador de redes completamente ópticas, en donde se modela una malla de red óptica y donde se compararan diversos algoritmos de enrutamiento por deflexión, aplicados específicamente en el dominio de las redes ópticas. También en Arcia-Moret [5] analiza el impacto de los *divacks*, exponiendo un análisis detallado de resultados obtenidos a través de simulaciones realizadas con NS-2, cuantificando así beneficios y problemas que podría existir usando *divacks* en una red con un único cuello de botella (altamente concurrido).

1.2. Justificación

Muchos de los simuladores de redes existentes, son sistemas con curva de aprendizaje lenta y con un nivel de detalle elevado, y en ocasiones para hacer llegar los conceptos básicos de las redes a un público de poca experiencia, se necesita explicar de manera más simple las dinámicas propias del envío de información en las redes. Más todavía, cuando se desea estudiar el comportamiento de las nuevas tecnologías en diversos escenarios, de manera fácil y rápida.

También, como vemos en el capítulo 5 hemos realizado estudios sobre las particularidades de TCP y los ACKs divididos para la aceleración de crecimiento de cwnd[4] en una red y su comportamiento en el cuello de botella.

1.3. Objetivos

1.3.1. Objetivo General

Diseñar, implementar y probar una herramienta de simulación, que contemple la capa transporte y un elemento intermedio que funja de cuello de botella para el estudio de los fenómenos de congestión a través de los algoritmos de TCP.

[4]Ventana de congestión de TCP

1.3.2. Objetivos Específicos

- Realizar una revisión bibliográfica sobre *divacks* y estudios previos relacionados los mecanismos de arranque rápido en TCP.
- Proponer una herramienta para observar el comportamiento al variar la cantidad de *divacks* a usar en una transferencia TCP, basado en [4].
- Modelar matemáticamente el impacto de los *divacks* en la transmisión de datos.
- Usar la técnica de simulación por eventos discretos, para el desarrollo del simulador llamado MicroSim: $\mu\mathcal{SIM}$
- Diseñar, implementar y probar una herramienta de simulación, que calcule y ajuste la velocidad de transmisión para archivos cortos.
- Realizar estudios comparativos de diversos escenarios simulados.

1.4. Método de Desarrollo

Hemos creado una herramienta que corresponde a la expresión esencial de la congestión una red TCP/IP, es decir, que contenga todos los elementos necesarios para modelar el comportamiento de una red. Su estructura fue construida de la manera más simple y comprensible ya que el objetivo es que siga siendo usado como útil didáctico, basado en la experiencia de los cursos de redes de pregrado de impartidos en el departamento de computación de la escuela de sistemas (ULA).

Como trabajo futuro quisiéramos que la herramienta pueda crecer y soportar el envío de múltiples flujos para hacer estudios comparativos.

Debido a la naturaleza de este proyecto se utilizará como guía el **Modelo de Desarrollo en Espiral**, ya que el enfoque que se da al desarrollo del producto, es iterativo e incremental. Este modelo fue inicialmente propuesto inicialmente por Boehm [14], y se basa en organizar todas las actividades esenciales para el desarrollo de software en una espiral, dividida por ciclos, que representan una fase o iteración del producto.

En cada ciclo de la espiral, existen cuatro regiones bien definidas y recorridas generalmente en el siguiente orden:

1. **Definición de objetivos**: En esta fase de trabajo, se encuentran los objetivos que se deben cumplir al finalizar el ciclo que se está transcurriendo, hay que tomar en cuenta todas las conclusiones que se hicieron en la fase anterior (solo si no está en primer ciclo), definir restricciones e identificar los riesgos del producto que se va a elaborar en esta iteración.

2. **Análisis de riesgos**: Donde se estudia con cuidado los riesgos observados anteriormente y se descubre la manera de reducirlos o buscar alternativas para que no existan.

3. **Desarrollo y pruebas**: Básicamente se implementa y cumple con todos los objetivos y requerimientos planificados. Luego se pone a prueba el producto o prototipo realizado, para verificar su correcto funcionamiento.

4. **Planificación**: El proyecto entra en fase de planificación si requiere de un ciclo nuevo, se evalúan los resultados y se planifica el próximo ciclo.

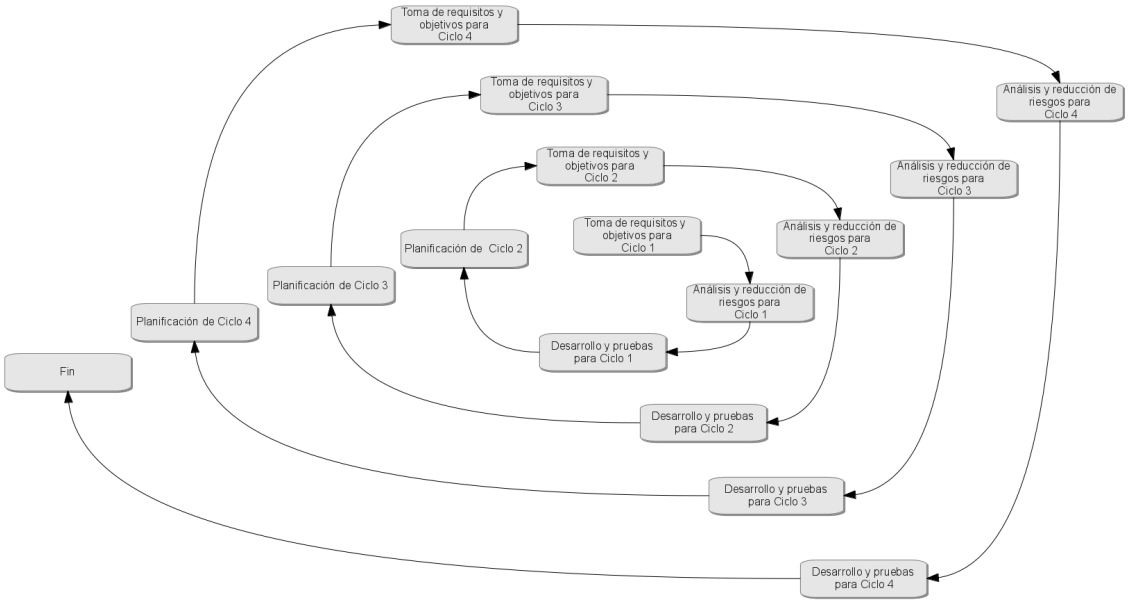

Figura 1.1: Diagrama de Modelo de Desarrollo en Espiral

El simulador fue desarrollado en cinco regiones bien definidas y presentadas cada

una en un ciclo de la espiral. A continuación describimos cada una de estas fases de trabajo:

1. **Primer ciclo**: Simulación de la propagación de paquetes a través de dos nodos.

   a) Creación de las estructuras básicas: Nodo y enlace (*Link*)

   b) Creación del Planificador (*Scheduler*) e introducción de su funcionalidad. Dentro de esta fase también definimos la clase Evento, fundamental para el planificador.

   Pruebas: Se realizaron pruebas de transmisión de paquetes en un solo sentido. Se puede observar una prueba realizada en la figura del Capítulo 3 en la Sección 3.1.2

2. **Segundo ciclo**: Agregación de Capa Transporte y Capa Aplicación al simulador.

   a) Implementación de la noción de Capa y separación entre nodos y costo de comunicación entre nodos.

   b) Determinación del orden y precisión de los tiempos asignados a cada evento. Identificación de los componentes básicos de un evento: nodo origen y destino (acción fundamental para la separación de los entes intermedios de la red).

   c) Implementación del *Logger* y generación de archivos con formato NAM[5].

   Pruebas: Se realizaron pruebas de transmisión de paquetes entre capas y se capturó trazas generadas en archivos con formato NAM, verificando la correctitud de los tiempos asignados a cada evento. Un ejemplo de traza en archivo NAM se puede observar en el capítulo 4 en la sección 4.5.

3. **Tercer ciclo**: Implementación el nodo intermedio (Router).

   a) Estudio de funcionamiento de nodo intermedio básico.

   b) Implementación del envío de paquetes en dirección opuesta.

   c) Implementación de pérdida de paquetes en el nodo intermedio.

[5]Network Animator, herramienta para animar redes desarrollado para NS-2

d) Verificación de las consecuencias en la pérdida de paquetes

Pruebas: Se realizaron pruebas de transmisión de paquetes que entran y salen de nodo intermedio y el funcionamiento del *buffer* del nodo intermedio. En el capítulo 5 se pueden observar pruebas realizadas al nodo enrutador, en la sección 5.3.

4. **Cuarto ciclo**: Implementación de TCP en capa transporte según [3].

   a) Implementación de recuperación de pérdidas basándose en algoritmo New Reno[15].

   b) Noción de crecimiento de ventana de congestión, ventana deslizante y ventana anunciada.

   Pruebas: Se realizaron pruebas de crecimiento de ventana de congestión en la capa transporte en emisor y pruebas de envío de ACKs en la capa transporte del receptor. En el capítulo 5 se pueden observar diferentes pruebas que se hace a la implementación de TCP

5. **Quinto ciclo**: Implementación de la técnica de *divacks* en $\mu\mathcal{SIM}$.

   a) Implementación de envío de *divacks* por parte de la capa transporte del receptor

   Pruebas: Se realizaron pruebas de aceleración de crecimiento de ventana de congestión en la capa transporte en emisor. En la sección 5.4 del capítulo 5 se puede observar varias pruebas realizadas a los *divacks* en $\mu\mathcal{SIM}$

En el capítulo 3 se detalla como se implementó el simulador, en el capítulo 5 se puede observar pruebas realizadas al simulador.

Capítulo 2

Marco Teórico

Abordaremos en este capítulo algunos conceptos básicos de redes, necesarios para la comprensión de algunas técnicas expuestas en el documento. Con el propósito de aclarar las ideas expuestas y facilitar la comprensión del propósito subyacente en $\mu\mathcal{SIM}$.

2.1. El Modelo OSI

Modelo de interconexión de sistemas abiertos, abreviado **OSI** (proviene del ingles para, *Open System Interconnection*). Estandarizado bajo el codigo ISO/IEC 7498-1 [16], creado por la Organización Internacional de Normalización (**ISO**) en conjunto con la Comisión Electrotécnica Internacional (**IEC**), durante la década de los 80s, con el objetivo principal de modelar el intercambio de información entre *sistemas abiertos*[1].

Entonces, modelo OSI se puede definir como una serie de reglas que se aplican a sistemas para estandarizar su interconexión y comunicación.

OSI se caracteriza por ser un modelo a capas, donde cada una de estas capas tienen asociadas una serie de conceptos y reglas específicas, y están diseñadas para representar el funcionamiento de una serie de elementos bien caracterizados que forman parte de la vida de sistemas que se comunican entre si. Hoy en día se le atribuye el éxito de la escalabilidad de la Internet a este modelo. Cada capa es un problema aparte, al punto

[1]ISO define sistemas abiertos como una serie de dispositivos que forman un todo autónomo, capaz de realizar procesamiento y/o transferencia de información, y además, cumple con los estándares requeridos por OSI en su comunicación con otros sistemas abiertos[16].

que cada una de estas puede simplificarse mediante una interfaz [17] que presta servicio a la capa precedente (en orden ascendente o descendente).

2.1.1. Capas de OSI

Figura 2.1: Capas de Modelo OSI y OSI Simplificado

Según [16] el modelo OSI está compuesto por 7 capas a saber:

- **Capa 1, Capa física**, engloba cada uno de los atributos para conectarse físicamente con el medio por el cual se va a transmitir la información entre dos entidades. El medio corresponde la vía que conecta físicamente 2 extremos a

comunicar. Capa 1 de OSI, debe proporcionar los estándares que indican como activar, mantener y desactivar una conexión física. En resumen, es la capa que se encarga de transformar la información en señales físicas que se pueden transmitir por el medio alámbrico o inalámbrico y recibir señales que vienen del medio, para transformarla en **información** que se pueda usar en capas superiores.

- **Capa 2, Capa enlace de datos**, se encarga de verificar la integridad de la información que va a ser enviada entre sistemas conectados de manera local o directa, se encarga de realizar control de acceso al medio (MAC), para esto, se divide la información en *tramas*, a las cuales se asigna un ID que será único localmente, considerado como dirección MAC. Estas tramas facilitará gestionar la corrección de errores y ocuparse del control de flujo entre sistemas.

- **Capa 3, Capa de red**, se considera una Red, a una serie de terminales (sistemas) interconectados de manera local (directa) para trasmitir información entre sí. Si deseamos comunicar terminales que están ubicados topológicamente en redes distintas, es necesario tomar en consideración una serie de reglas. la Capa 3 del modelo OSI se encarga de proponer estas reglas. Básicamente la información se dividirá en paquetes, y cada paquete tendrá como información adicional, de donde sale y hacia dónde va, esto para implementar algoritmos de decisión de ruta más corta (considerado enrutamiento), para llegar al destino de manera óptima.

- **Capa 4, Capa transporte**, una vez establecida la conexión entre dos terminales o extremos, la capa 4 de modelo OSI, se encargará que transportar entre dos aplicaciones remotas todos los segmentos de información (libre de errores), además deberá garantizar que exista un flujo constante de datos en movimiento por la red. La Capa transporte tiene dos protocolos principales, UDP y TCP. UDP (User Datagram Protocol) se caracteriza por no estar orientado a conexión y TCP por estar orientado a conexión, cada uno con atributos y rendimiento según el tipo de aplicación a la que sirven.

- **Capa 5, Capa de sesión**, crea y mantiene el enlace lógico que existe durante la vida de una conexión entre 2 terminales donde se transmite datos de cualquier

tipo. Mantener una conexión es muy útil, ya que se garantiza que se pueda reanudar en caso de existir alguna interrupción. RPC [18] es un ejemplo de protocolo que trabaja en capa 5.

- **Capa 6, Capa de presentación**, realiza la función de ser una interfaz entre terminales que representan o manipulan la información de manera distinta, en este caso no es necesario que existan estándares para los dos extremos, entonces básicamente lo que la capa 6 hace es recibir la información y reinterpretarla de manera que el sistema lo pueda procesar (haciendo las veces de traductor). Algunos protocolos que trabajan en capa 6 son NCP, Telnet, NDR, XDR, PAD.
- **Capa 7, Capa de aplicación**, maneja aquellos servicios lógicos que generan la información a ser transmitida, es dictada por protocolos definidos a nivel de capa aplicación, y estos protocolos son los que permiten al usuario final realizar envío de información en forma de mensajes a través de una red. Algunos protocolos que trabajan en capa 7 son FTP, DNS, HTTP, SSH.

En la figura 2.2 se puede observar como un mensaje generado por capa aplicación, es picado y vuelto trama.

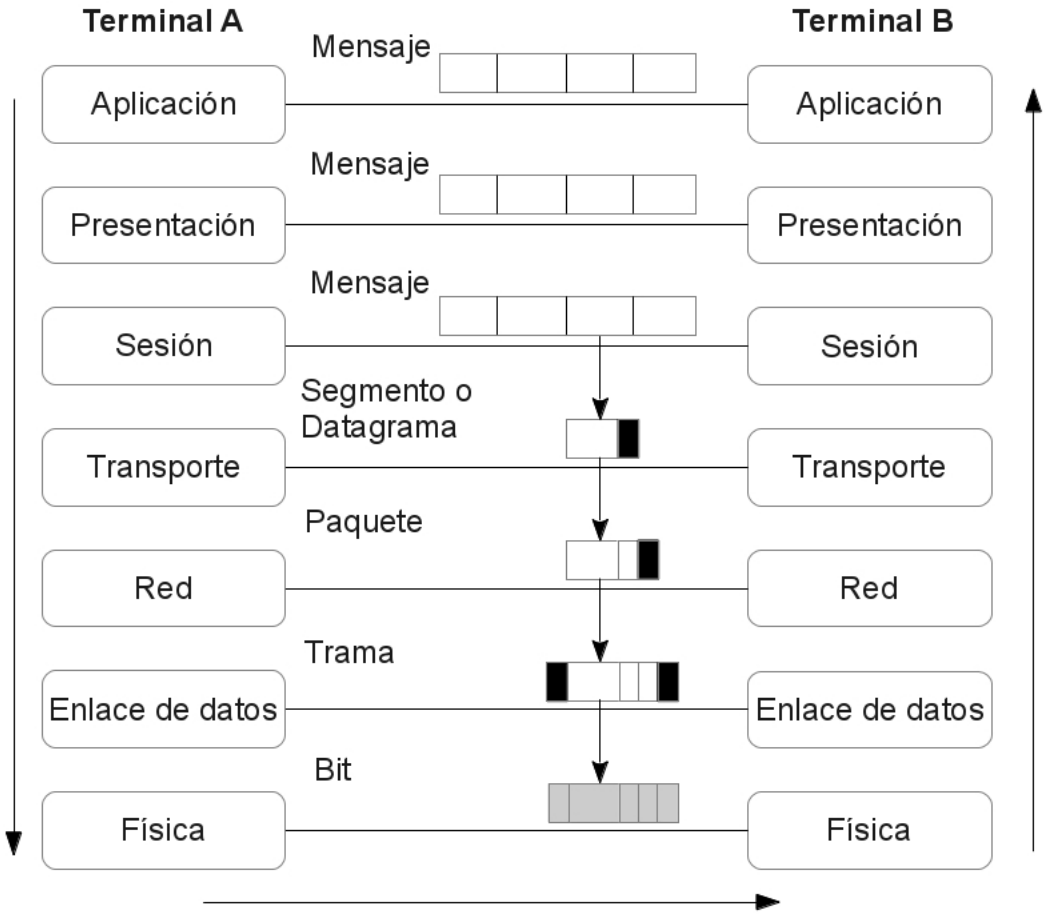

Figura 2.2: Capas de Modelo OSI con respectivos PDU

2.1.2. La Unidad de Datos de los Protocolos (Protocol Data Unit)

Comúnmente denominado PDU, es el elemento esencial el cual va a contener los datos que van a ser transmitidos, la información no puede se transmitida en un solo bloque por la red, tiene que ser dividida en partes más pequeñas, para que la transmisión de esta sea manejable fácilmente por todas las capas del modelo OSI. Una PDU básicamente va a transportar una porción del mensaje entre los dos extremos e información concerniente al protocolo que opera sobre el mensaje; de allí su nombre: *unidad de dato del protocolo.*

Ya que cada capa controla una parte distinta de la red, a los PDU se le

añadirá información de control correspondiente a cada capa, esta información esta contenida en una sección denominado cabecera o *header*. La cabecera va a encapsular los datos a transmitir en cada capa del modelo OSI, y cumple con la función de facilitar la comunicación efectiva entre capas, además de contener información útil para la capa homóloga a ser procesada en el terminal destino.

Se puede observar en la figura 2.2, los nombres de cada uno de los PDU asociados a cada capa del modelo OSI:

- Para la capa 1, el PDU asociado es el **bit**, también denominado símbolo.
- Para la capa 2, el PDU asociado es denominado **trama**.
- Para la capa 3, el PDU asociado es denominado **paquete**.
- Para la capa 4, el PDU asociado es denominado **segmento** para TCP, o **datagrama** para UDP.
- Para la capa 5, 6 y 7, el PDU asociado es denominado **mensaje**.

2.2. El Protocolo de Control de Transmisión (TCP)

Según [3], el Transmission Control Protocol (en español Protocolo de Control de Transmisión) o TCP, es uno de los protocolos fundamentales en Internet, pues es el de mayor uso. Un 90 % del trafico de Internet es controlado por este protocolo [1]. Muchos programas dentro de una red de datos compuesta por entes comunicantes, pueden usar TCP para crear conexiones entre ellos a través de las cuales puede crearse un flujo a través del cual circula un envío de datos confiable. El protocolo garantiza que los datos serán entregados en su destino sin errores y en el mismo orden en que se transmitieron. También proporciona un mecanismo para distinguir distintas aplicaciones dentro de una misma máquina, a través del concepto de puerto.

TCP es un protocolo de transporte de datos entre dos extremos (un emisor y un receptor), y se basa en la transferencia organizada y segura de datos, es decir, que si un segmento de información se pierde durante el proceso de envío, el protocolo se encarga de la retransmisión de dicho segmento; para continuar con la transferencia de los datos.

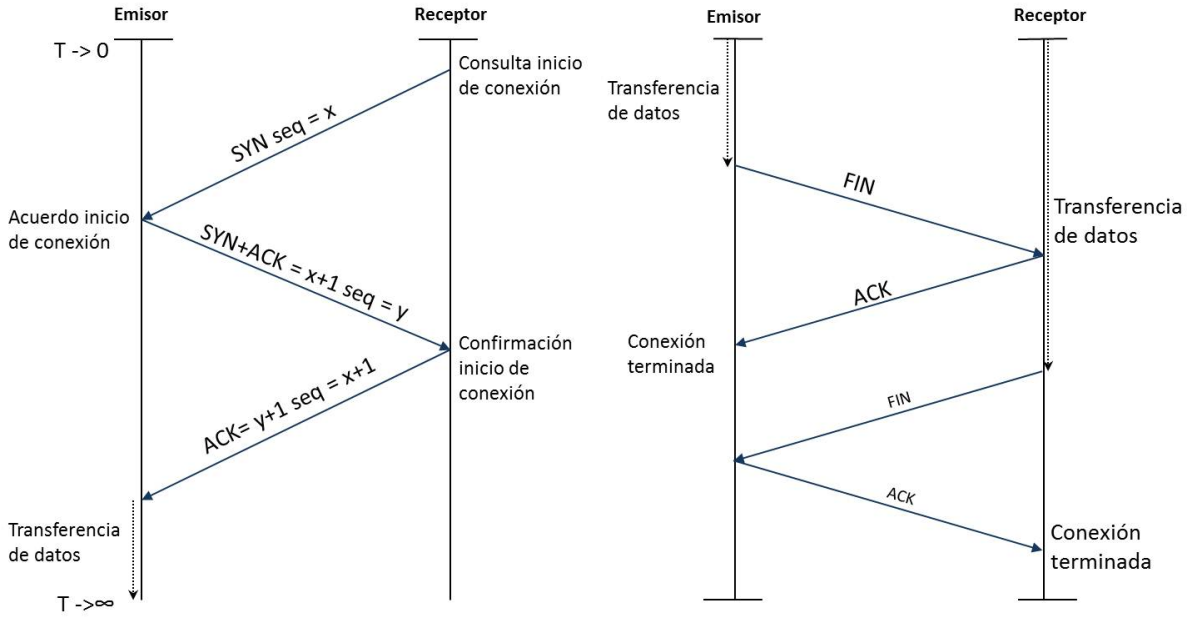

Figura 2.3: Establecer y cerrar una conexión

El comportamiento natural de TCP consiste en enviar la información deseada dividida en segmentos de forma ordenada y en ráfagas, es decir, el emisor enviará una ráfaga de segmentos y esperara que el receptor confirme la llegada completa de estas ráfagas, la cantidad de segmentos enviados en una ráfaga esta definido por el tamaño de ventana de congestión (***cwnd***). El receptor confirmará la llegada correcta de los segmentos, enviando segmentos de reconocimiento (**ACK**). Los segmentos ACK también son una forma de indicar que el canal de datos esta disponible para recibir un flujo más grande, por lo tanto, la llegada de ACKs al emisor indica el crecimiento de la ventana de congestión.

TCP es orientado a conexión. Esto quiere decir que antes de comenzar la transmisión de datos los *hosts* involucrados deben establecer un acuerdo, en el cual intercambian y establecen los parámetros para asegurar la transferencia de datos. Esta conexión se realiza (figura 2.3a) mediante un proceso conocido como acuerdo de tres fases (*three-way handshake*): En la primera fase, el *host* cliente (receptor) envía un segmento especial que no contiene datos (segmento SYN) al *host* servidor (emisor) para hacer la petición de inicio de la conexión. En la segunda fase, el emisor responde con un

segmento reconociendo la llegada del SYN del receptor (SYN+ACK). Finalmente, en la última fase, el receptor envía un ACK al emisor y de esta manera queda establecida la conexión entre ambos. En este punto se han acordado los números de secuencia iniciales (identificación de los paquetes) y se ha reservado espacio para los *buffers* y para las variables de la conexión. Luego de esto (figura 2.3b), cualquiera de los dos *hosts* puede terminar su conexión mediante el envió de un segmento FIN y la recepción del reconocimiento mediante un ACK, una vez que ambos *hosts* hacen esto, todos los recursos son liberados.

2.2.1. La Ventana de Congestión

Como se dijo antes, la ventana de congestión (*cwnd*) tendrá como valor el numero de paquetes (en bytes) que puede llegar a ser transmitido (por el emisor) en un instante de la dinámica de una transmisión. Conocer su valor en todo momento es muy importante para comprender el ancho de banda disponible en la red para una conexión TCP. En la figura 2.4 se puede observar un modelo gráfico detallado de (*cwnd*).

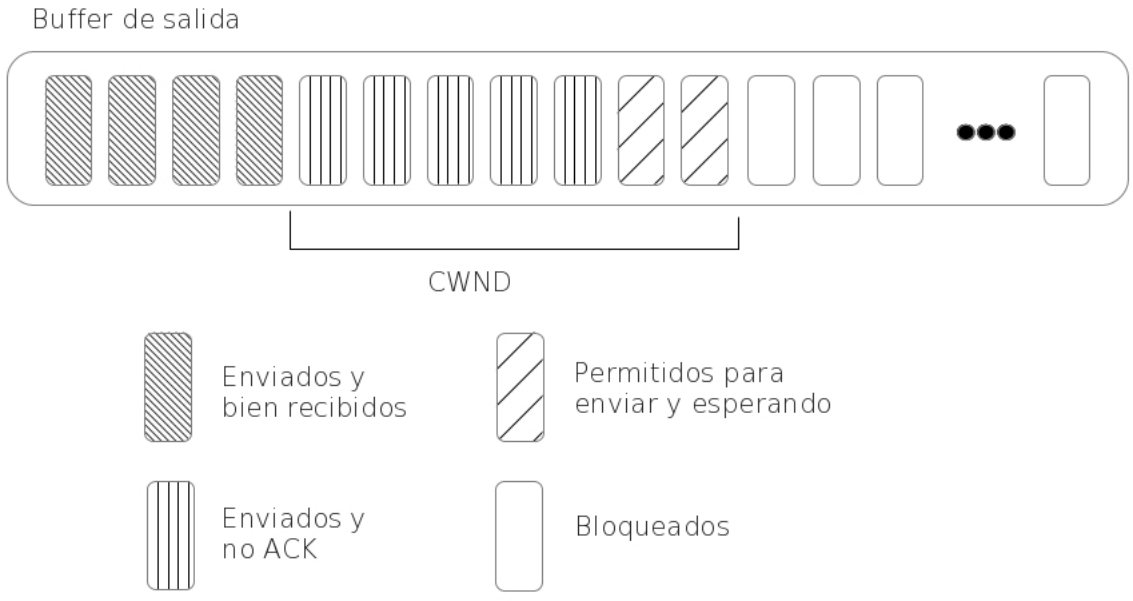

Figura 2.4: Ventana de Congestión

Existe un comportamiento de (*cwnd*) llamado deslizamiento de ventana o **ventana**

deslizante, como se observa en la figura 2.4, (*cwnd*) va a ser una variable indicadora que se va a mover (o deslizar) a lo largo de la cola de paquetes que van a ser transmitidos, este movimiento va a ocurrir solo si llega un ACK reconociendo un paquete enviado anteriormente. Así por ejemplo, como se ve en la figura 2.5, a la llegada del ACK, la ventana crece y se desliza para así aumentar el flujo de datos en la red.

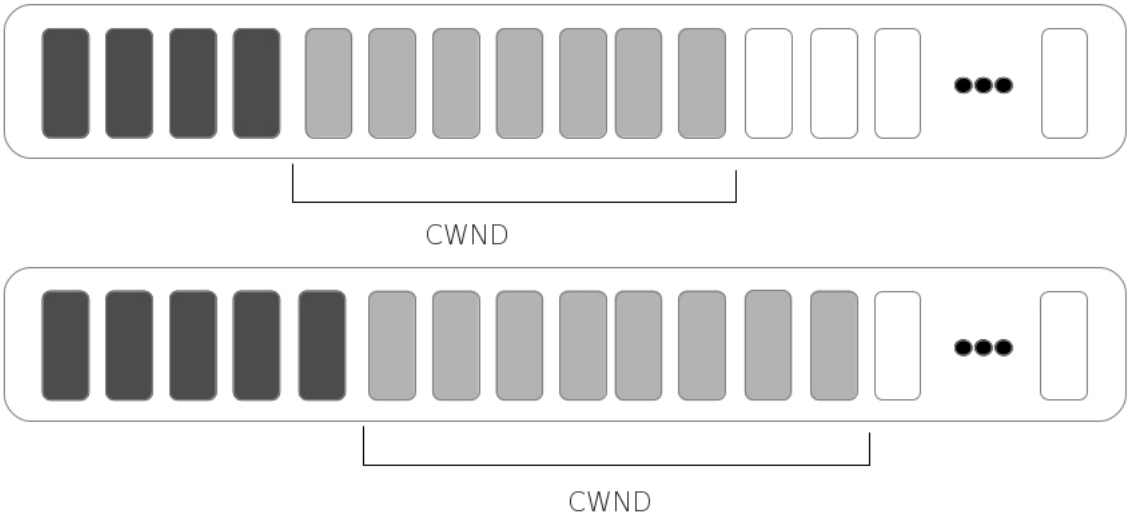

Figura 2.5: Deslizamiento de Ventana de Congestión

2.2.2. Control de Errores

Dependiendo del entorno[2], existe la posibilidad de que ocurran pérdidas de datos en una conexión. La manera más simple de detectar una pérdida, es llevando un temporizador denominado *timeout*, el cual actualizara su valor cada vez que la ventana de congestión se mueve. Si se cumple el tiempo de *timeout*, entonces se considera la existencia de una pérdida de paquete.

Al ocurrir una pérdida, el emisor debe retransmitir el paquete perdido, para que esto ocurra, existen tres protocolos que garantizan la correcta retransmisión de paquetes perdidos, denominados **ARQ** (del inglés *Automatic Repeat-reQuest*):

- **Stop-and-wait**, consiste en enviar un paquete y esperar a que llegue su reconocimiento (ACK) para poder enviar el próximo paquete de la cola. Si existe

[2]Condiciones variantes que definen el ambiente donde se realiza la transmisión

una pérdida de paquete se espera a que se cumpla el *timeout*, y se reenvía el paquete correspondiente. Su comportamiento es bastante simple y se considera como un caso especial donde *cwnd* no crece y su valor es igual a 1. El problema de este algoritmo es que no se saca provecho de la capacidad total de la red cuando es mucho más grande que un segmento de tamaño máximo y, en consecuencia, la transmisión es muy lenta.

- **Go-Back-N**, mejora de stop-and-wait, donde *cwnd* puede ser de tamaño variable y mayor o igual a 1, en este caso, se puede usar el concepto de ACKs acumulados, la cual dice que un ACK que reconoce un paquete con número de secuencia X, también reconoce a todos los paquetes con número de secuencia inferiores a X. Hay que considerar que en Go-Back-N, el valor de *timeout* se establecerá al enviar el paquete más antiguo. Si existe una pérdida de paquete, el emisor esperará a que se cumpla el *timeout* y tendrá que reenviar todos los segmentos que han sido enviados y no reconocidos, osea, reenviar toda la ventana de congestión. Solo se envían paquetes nuevos cuando se reconozca toda la ráfaga de paquetes enviada anteriormente.

- **Selective Repeat**, en este caso el receptor debe enviar ACK por cada paquete recibido, se establecerá entonces el timeout por cada ACK recibido por el emisor y si existe una pérdida, solo se retransmite el paquete que se pierde. Se observa que con este protocolo, la transmisión va a ser continua, ya que con cada ACK recibido, la *cwnd* se deslizará y enviará paquetes nuevos. *Selective Repeat* es el protocolo ARQ implementado en el Simulador desarrollado.

2.2.3. El Control de Congestión

Entendiendo por conexión como el acuerdo entre dos extremos, cada conexión tiene capacidades de transmisión de datos distinta, es decir, hay redes (intermedias) más rápidas que otras. Esto implica que unas conexiones tienen un valor de *cwnd* distinto a otras (inclusive en ocasiones, partiendo del mismo origen y llegando al mismo destino), y que un valor inadecuado de *cwnd* puede causar congestión y por ende pérdidas de paquetes, que en general no se desea.

El Control de congestión definido en el RFC 5681 [19] establece que, para descubrir un valor adecuado de *cwnd*, existe varias fases que definirán su crecimiento. Sin embargo hay que considerar que siempre existirá un valor de ventana de congestión inicial.

TCP tiene dos fases principales durante la vida de la transmisión de la información, **Slow Start** y **Congestion Avoidance**. También existe una fase de transición media en caso de recuperación, denominada **Fast Recovery**.

2.2.3.1. Fase de Crecimiento Rápido (*Slow Start*)

Slow Start es la fase donde toda conexión TCP iniciará y es la que se encarga de descubrir el tamaño máximo de la ventana de congestión. Esta fase plantea que se debe iniciar enviando a la red desde el emisor, un volumen pequeño de datos, que se irá aumentando exponencialmente hasta que la red se sature.

El algoritmo define un crecimiento rápido de la *cwnd*, donde cada vez que el emisor recibe un ACK, la ventana de congestión crece en un MSS[3], además la ventana se deslizará abriendo así la posibilidad de enviar dos paquetes a la llegada de un ACK. Eq. 2.1 es la operación que se realiza al llegar un ACK al emisor durante Slow Start.

$$cwnd = cwnd + 1 \tag{2.1}$$

Esta ecuación resume la acción de un ACK sobre la percepción de la congestión que tiene un emisor. Es decir, que cuando un ACK llega al emisor, este indica que un paquete de datos ha salido de la red y entonces puede inyectar otro correspondiente al deslizamiento. más aún, como está en la fase de descubrimiento puede inyectar un otro *nuevo* paquete de datos a la red correspondiente al crecimiento de la ventana.

[3]Del ingles para Maximum Segment Size, es el tamaño máximo de un segmento de TCP.

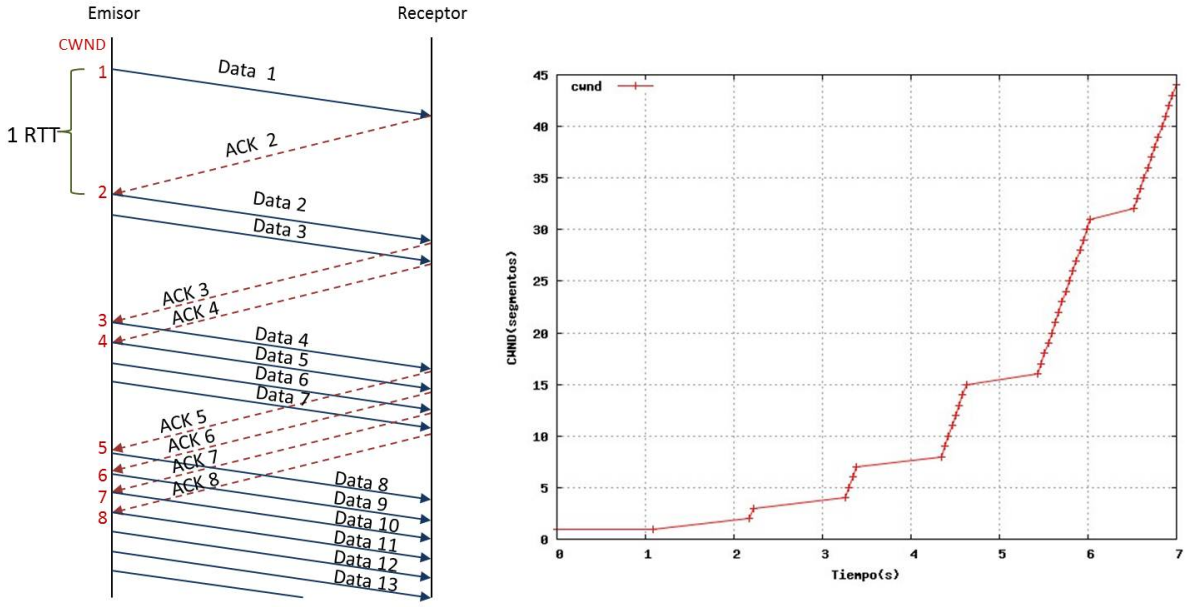

Figura 2.6: Comportamiento de Slow Start

En figura 2.6 se puede observar como crece *cwnd* por cada ACK recibido, en un caso donde no existe pérdida de paquetes. Existe un valor que determina el momento en el que la fase Slow Start culmina, llamado *SSThreshold* o el umbral de Slow Start, es mantenido y asignado internamente por TCP. *SSThreshold* (en cantidad de segmentos), es el valor máximo que puede tener *cwnd* para que se mantenga en Slow Start, si *cwnd* llega a ser mayor que *SSThreshold*, entonces TCP pasará a fase **Congestion Avoidance**.

2.2.3.2. La fase de control de la congestión (*Congestion Avoidance*)

Como su nombre lo indica, es la fase donde se trata de evitar o controlar la existencia de congestión[4], para esto, se sigue la dinámica de control *AIMD* (additive-increase/multiplicative-decrease), la cual propone un algoritmo de crecimiento conservativo y de reducción agresivo. Esta fase es la que garantiza la estabilidad y escalabilidad de la red. Dos o más usuarios haciendo transferencias simultaneas confían

[4]Recordemos que la congestión en una red es necesaria para mantener el uso de los enlaces lo más alto posible

mutuamente que en esta fase se garantiza que quien más congestione la red bajará su tasa de transmisión oportunamente para igualarse a los otros concurrentes.

El crecimiento de *cwnd* durante Congestion Avoidance se ve expresado en eq. 2.2, la cual se aplica por cada llegada de ACK al emisor.

$$cwnd = cwnd + \frac{1}{old\_cwnd} \tag{2.2}$$

En figura 2.7 se visualiza como crece *cwnd* por cada ACK recibido durante la fase Congestion Avoidance. Esta fase hace posible que se crezca de forma aproximadamente de 1 paquete de datos por RTT. Es decir que aproximadamente se tiene:

$$cwnd = cwnd + \frac{1}{cwnd} * cwnd \approx cwnd + 1 \tag{2.3}$$

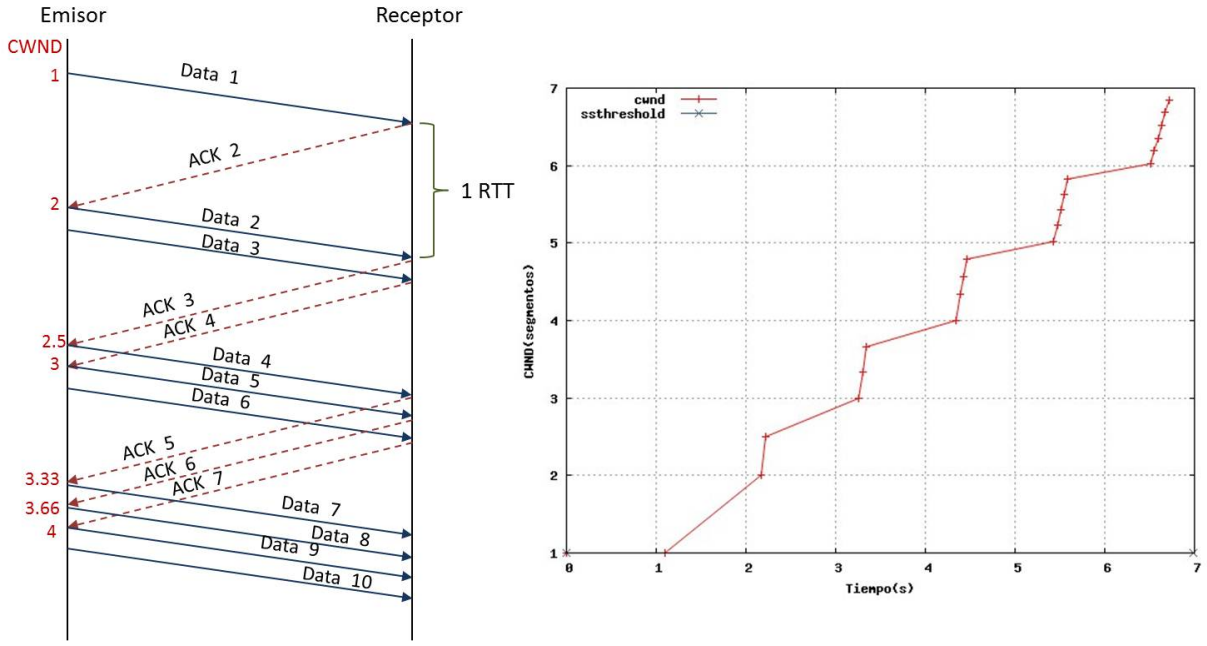

Figura 2.7: Comportamiento de Congestion Avoidance

2.2.3.3. Detección de pérdidas y Recuperación Rápida (Fast Recovery)

Cuando existe una pérdida detectada por *timeout* en el emisor, entonces se debe retransmitir desde el último paquete no reconocido e iniciar la ventana de congestión en la posición de este paquete, también entonces serán retransmitidos todos los paquetes enviados que van después del último paquete no reconocido. Las siguientes acciones serán tomadas en el emisor:

$$ssthreshold = \frac{cwnd}{2} \tag{2.4}$$

$$cwnd = cwnd(t = 0) \tag{2.5}$$

Como observamos en la eq. 2.4 y en la eq. 2.5, el valor de ssthreshold se asigna igual a la mitad de la ventana de congestión y se regresa al valor inicial de *cwnd*, y se pasa a fase *Slow Start* nuevamente.

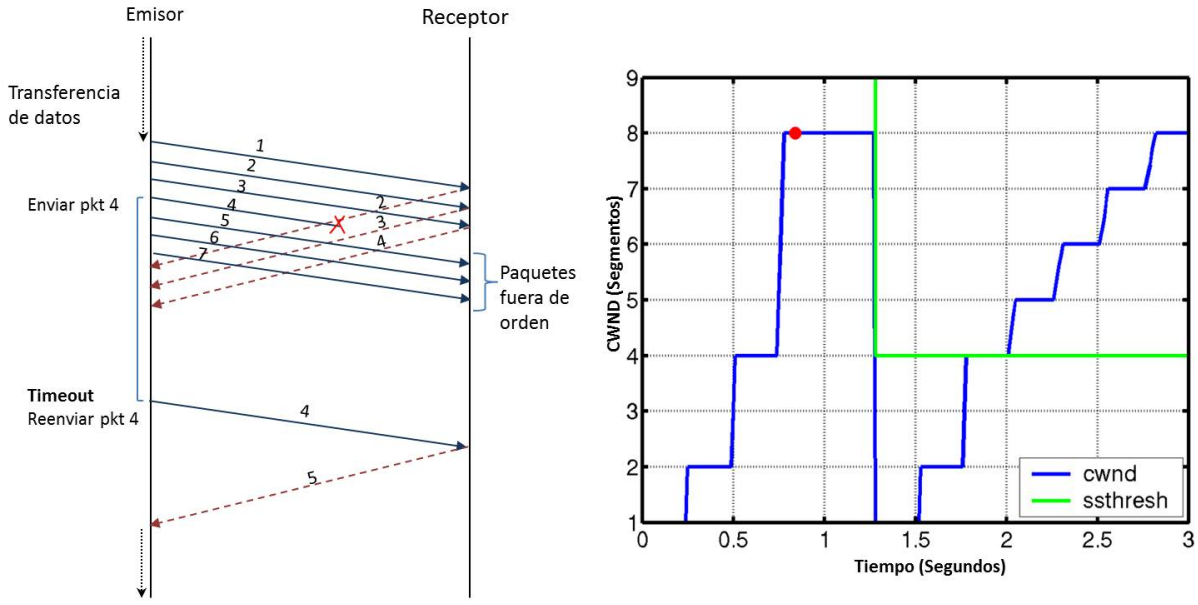

Figura 2.8: *cwnd* cuando ocurre un timeout

En el RFC 5681 [19], se define una nueva manera para detectar pérdidas, para garantizar la transmisión continua de datos, y así no perder tanto tiempo (como ocurre al esperar un timeout). El comportamiento del receptor ahora se redefine, ya que estamos agregando la posibilidad de una recuperación rápida y de eliminar el descarte de paquetes que llegan fuera de orden[5], para esto, se debe bufferizar todos los paquetes que llegan fuera de orden y hacer peticiones repetidas del paquete esperado, estas peticiones repetidas se denominan dupACKs o ACKs duplicados, y solo se dejarán de enviar si y solo si el paquete perdido llega al receptor.

Al recibir un dupACK el emisor considera que posiblemente hay congestión de red y no va a hacer crecer *cwnd* y va a esperar recibir un ACK normal. Si llega a recibir un tercer dupACK, entonces, se reenvía el paquete perdido y TCP pasa a fase Fast Recovery ya que existe pérdida de paquete.

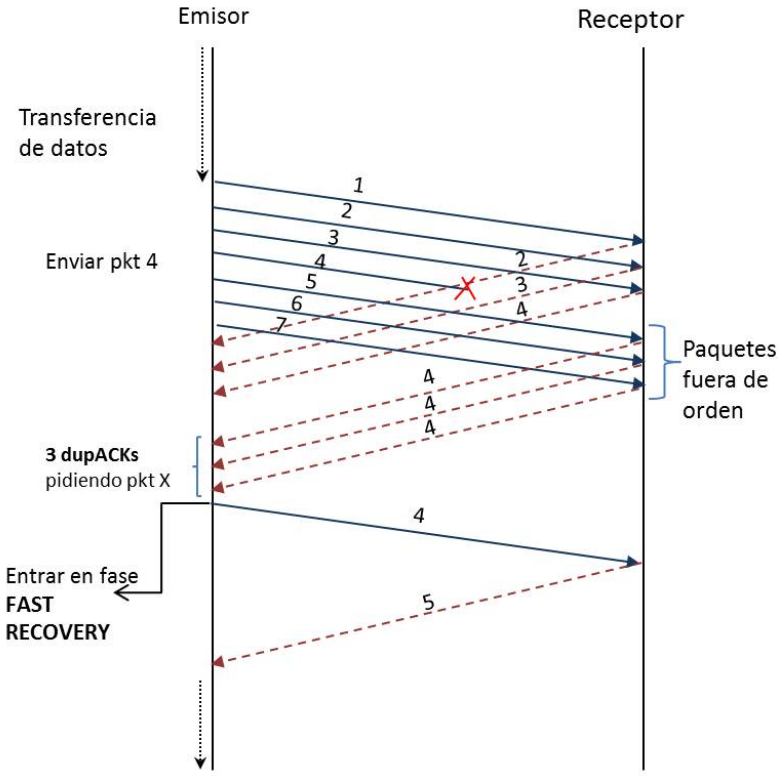

Figura 2.9: TCP durante la llegada de 3 dupACKs

[5]Si un paquete llega antes del paquete esperado se considera fuera de orden, esto ocurre cuando hay congestión en la red y existe pérdidas

TCP tiene definido varios algoritmos para el control de pérdidas de paquetes. En este trabajo nos basaremos en el algoritmo *New Reno* para control de congestión de TCP, definido en RFC 6582 [15], el cual dictará el comportamiento de TCP durante la fase Fast Recovery.

Cuando TCP entra en **Fast Recovery**, establece el valor de *cwnd* igual a su mitad y ssthreshold igual a *cwnd* como se observa en eq. 2.6 y eq. 2.7.

$$cwnd = cwnd/2 \tag{2.6}$$

$$ssthreshold = cwnd \tag{2.7}$$

Mientras se siga en Fast Recovery, puede ocurrir dos cosas, sigue llegando dupACKs o llega un ACK reconociendo el paquete reenviado. Si llegan dupACKs, entonces se hace crecer *cwnd* de 1 MSS por dupACK recibido y se envía 1 paquete de datos, esto se hace para mantener la transmisión de datos de manera continua. Si llega un ACK reconociendo el paquete reenviado, entoces la *cwnd* se regresa al valor calculado por la eq. 2.6 que tenía justo cuando entro a Fast Recovery, y se pasa a fase Congestion Avoidance. Más adelante se detallara la implementación de Fast Recovery en $\mu\mathcal{SIM}$.

2.2.4. Incremento de la Velocidad de Transmisión

Se hace referencia al mecanismo basado en el aumento de la frecuencia de envío paquetes ACK para incrementar la velocidad de una transmisión, enfoque estudiado por Arcia-Moret et al. en [5][4][9], método conocido como *divacks* o ACKs divididos, estudiada por primera vez por Savage [10] donde se divide un ACK en varios paquetes de control, para reconocer de manera segmentada la llegada de un paquete al receptor. Si tomamos en cuenta que cada paquete de control, sea ACK completo o *divack*, va a hacer crecer la *cwnd* (dependiendo de *Slow Start* o *Congestion Avoidance*), se puede obtener una considerable aceleración del crecimiento de *cwnd* por cada RTT.

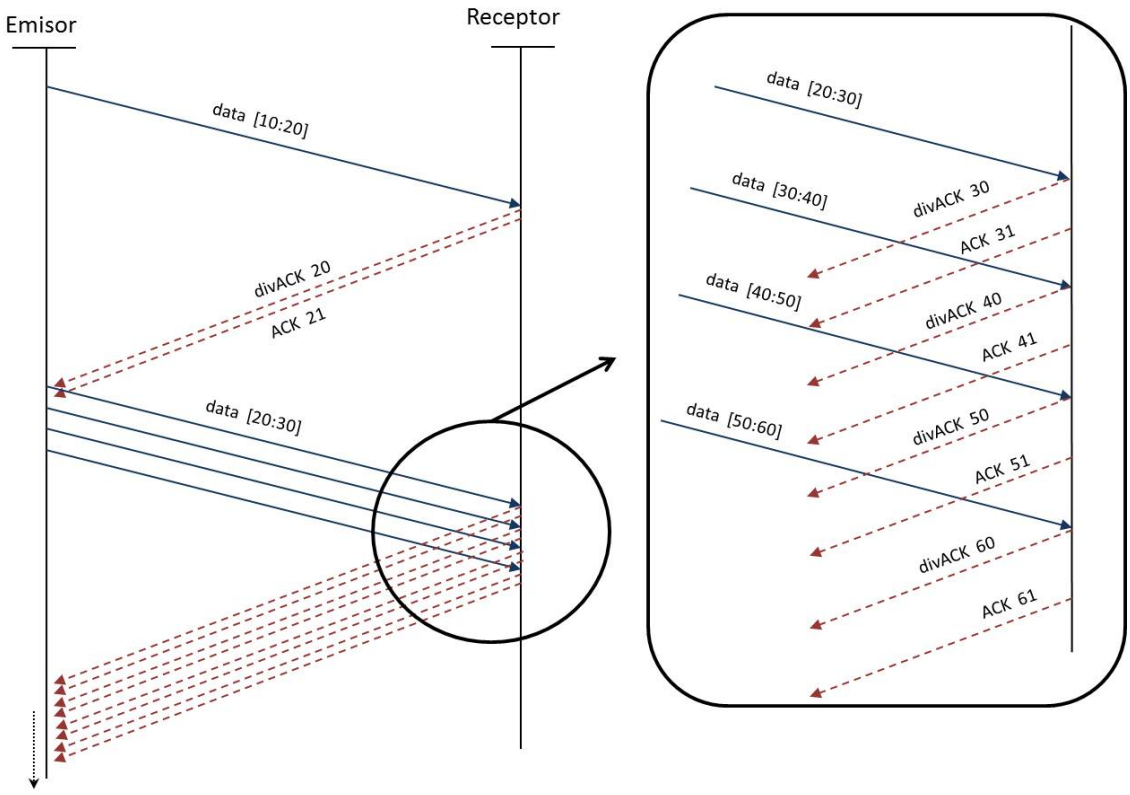

Figura 2.10: Técnica de división de ACKs: dos ACKs por cada paquete de datos

Se puede observar que en la figura 2.10, se realiza una transmisión donde el receptor enviará dos paquetes de control por cada paquete de datos recibido, si comparamos el crecimiento de la *cwnd* en transmisión con un solo ACK por paquete de datos, podemos ver que en fase de *Slow Start* estamos duplicando el crecimiento de la *cwnd* por cada RTT, solo si enviamos un ACK extra. Recordemos que aunque existe una aceleración en el crecimiento de la *cwnd*, la velocidad de la transmisión no solo va a estar dictada por el tamaño de la ventana de congestión, existen otros factores que afectan dicha transmisión, como por ejemplo, el ancho de banda disponible, tamaño de *buffer* en nodo enrutador, entre otros.

2.2.5. Consideraciones sobre *divacks* en Slow Start

Se tiene el interés inicial de desarrollar un modelo que defina el comportamiento de la fase Slow Start, en donde se aplica la técnica de *divacks* para acelerar la transferencia

de un archivo corto. El enfoque requerido es, encontrar una ecuación que relacione la cantidad de RTTs deseado para concluir la transferencia con la cantidad de *divacks* que se debe usar para concluir esta transferencia corta.

2.2.5.1. Definición de Modelo

El siguiente modelo está basado en los estudios realizados en [5], específicamente se hace uso del modelo propuesto (eq. 5.10 pag.87)

$$T_f = RTT \cdot \log_{(n+1)} \left(\frac{Fsize}{cwnd(0)} \right) \tag{2.8}$$

Esta ecuación refleja el tiempo que se tardaría en transmitir un archivo de tamaño $Fsize$ usando *divacks*. Podemos observar en eq. 2.8 que al aumentar la cantidad de $divacks(n)$ usados en la transferencia, existirá un decremento logarítmico en el tiempo que se realiza la transferencia (T_f).

De la eq. 2.8 podemos partir para crear el modelo, recordemos que el modelo que se busca, tiene como entrada la cantidad de RTTs que se quiere usar y como respuesta la cantidad de *divacks* que se deben usar para concluir dicha transferencia.

Definimos entonces,

$$T_f = r \cdot RTT \tag{2.9}$$

Donde r es la cantidad de RTTs requeridos para concluir la transferencia y RTT en el valor temporal promedio que tiene 1 RTT en esta transferencia. Con esto podemos remplazar en eq. 2.8 resultando

$$r \cdot RTT = RTT \cdot \log_{(n+1)} \left(\frac{Fsize}{cwnd(0)} \right) \tag{2.10}$$

Así, podemos eliminar la variable RTT de ambos lados de la igualdad. Resultando

$$r = log_{(n+1)}\left(\frac{Fsize}{cwnd(0)}\right) \tag{2.11}$$

Recordemos que la variable independiente del modelo es la cantidad de *divacks* (n), por lo cual tenemos:

$$(n+1)^r = \left(\frac{Fsize}{cwnd(0)}\right) \tag{2.12}$$

Y luego

$$n = \sqrt[r]{\left(\frac{Fsize}{cwnd(0)}\right)} - 1 \tag{2.13}$$

Obteniendo así en 2.13, el modelo deseado de Slow Start donde al ingresar la cantidad de RTTs deseados (r) para concluir una transferencia, el tamaño del archivo (*Fsize*), y el valor de la ventana de congestión inicial (*cwnd(0)*), podremos saber la cantidad de *divacks* necesarios para concluir la transmisión en dicho tiempo.

Ejemplo

Supongamos que queremos transmitir un archivo de 800 KB en 4 RTTs, y que el tamaño del paquete utilizado es de 1000 Bytes (además de usar una ventana inicial de 1 solo paquete). Entonces, utilizando el modelo anterior, los datos pueden ser presentados de la siguiente manera:

```
r = 4
Fsize = 800000 Bytes
cwnd(0) = 1000 Bytes
```

Si los usamos en la ecuación 2.13 como valores de entrada obtendremos lo siguiente:
$n = \sqrt[4]{\left(\frac{800000}{1000}\right)} - 1$
Obteniendo finalmente:

```
n = 1.99069 o aproximadamente 2 divacks
```

Lo que quiere decir que si usamos 2 *divacks* para transferir un archivo de tamaño 800000 Bytes y para una ventana de congestión inicial de 1000 Bytes, podremos concluir la transmisión en 4 RTTs.

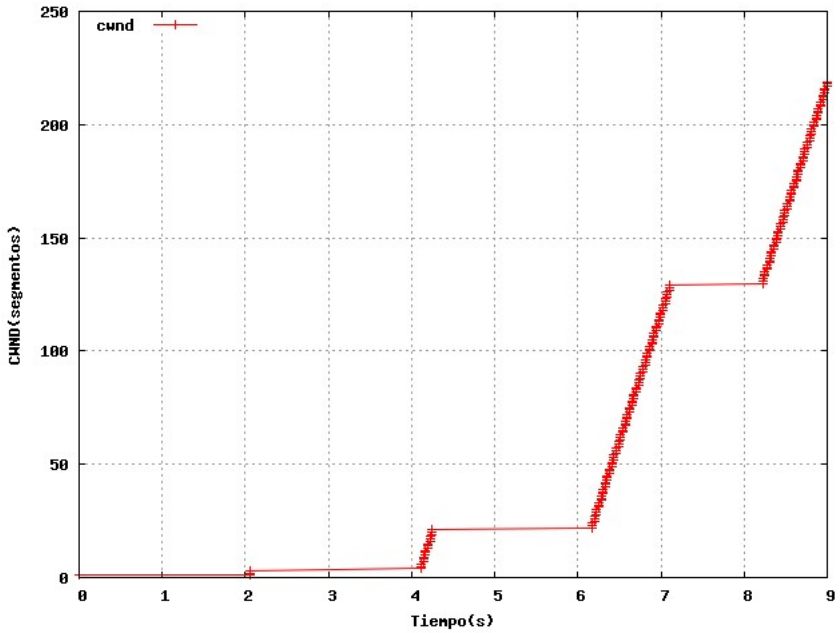

Figura 2.11: Crecimiento de cwnd con 2 divacks

En la figura 2.11, se muestra el crecimiento de la ventana de congestión durante la transmisión de un archivo de 800 KBytes usando 2 *divacks*. Se puede observar que la transmisión concluye en 4 RTTs.

2.3. Herramientas de Desarrollo y Visualización de Respuesta

Nuestra herramienta llamada $\mu\mathcal{SIM}$ se desarrolló en el lenguaje C++ y con interfaz en QT.

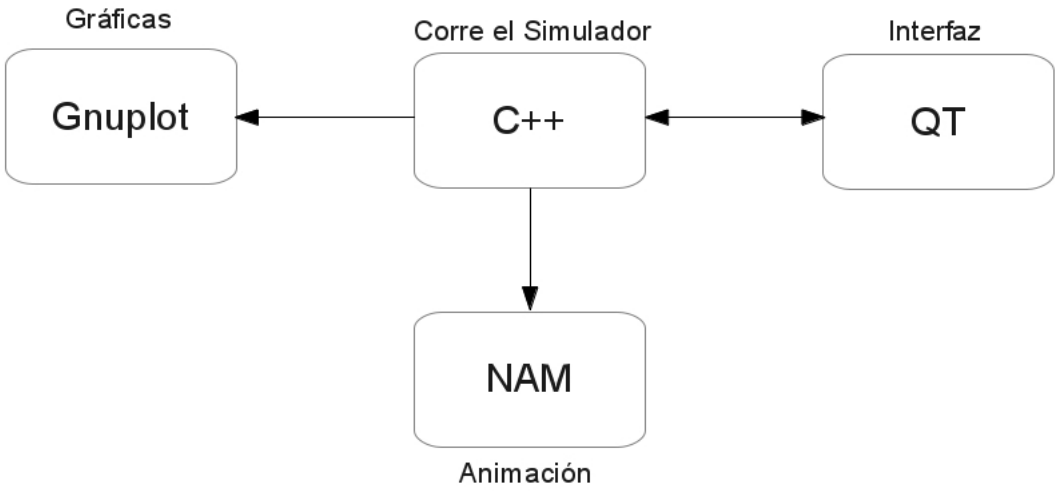

Figura 2.12: Herramientas de trabajo

2.3.1. El lenguaje C++

Lenguaje de programación que extiende el lenguaje C, el cual está implementado para adaptarse al paradigma orientado a objetos. Es muy extenso y lo suficientemente versátil para desarrollar todo tipo de proyecto, al ser un lenguaje compilado, se adapta bien a las necesidades de rendimiento que se exige para desarrollar programas de cálculo extenso, como por ejemplo un simulador por eventos discretos. Existen compiladores para la mayoría de los sistemas operativos modernos, como, Microsoft Windows, Linux, Mac OS X, Solaris, etc. Decir que hemos compilado el simulador en 3 plataformas: Mac Os, Windows y Linux. Su especificación detallada se encuentra en [20]. Siguiendo el paradigma a objetos, nosotros hemos planteado $\mu\mathcal{SIM}$ con un modelo de este tipo, detallado en el capítulo 3.

2.3.2. QT4

QT es un framework para el lenguaje C++ [21], utilizado para el desarrollo de interfaces gráficas de usuarios de aplicaciones, es bastante completo y altamente usado por la comunidad de desarrollo de usuarios de C++ actual, además de contar con herramientas básicas de creación de GUIs, añade una serie de bibliotecas que pueden

sustituir las bibliotecas estándares de C++, con el motivo de tener un entorno completo de desarrollo para la creación de aplicaciones totalmente multiplataforma. El kit de desarrollo es totalmente libre y compatible con Microsoft Windows, Linux y Mac OS X.

2.3.3. Network Animator (NAM)

Se usa NAM para visualizar ciertos eventos que ocurren en $\mu\mathcal{SIM}$, NAM es una herramienta basada en Tcl/TK de animación de simulaciones de redes, originalmente se usa para visualizar los resultados de una simulación realizada en NS (Networ Simulator). Dada la versatilidad de NAM, es totalmente posible visualizar trazas de $\mu\mathcal{SIM}$ fácilmente. NAM es capaz de mostrar esquemas gráficos de una topología de red personalizada, animación de paquetes que viajan a través de la red, y varias herramientas para observar detalles de la data.

2.3.4. Gnuplot

Gnuplot es una herramienta libre usada para generar gráficas de trazas obtenidas de $\mu\mathcal{SIM}$, es bastante simple se usar y compatible con los sistemas operativos más usados, como Mac Os, Windows y Linux y fácil de integrar en el lenguaje de programación C++ gracias a la técnica de *piping*[6].

[6]Es un mecanismo para comunicar de procesos en un sistema operativo. Los datos escritos en el *pipe* por un proceso, puede ser leído por otro proceso distinto.

Capítulo 3

El Simulador de Redes MicroSim ($\mu\mathcal{SIM}$)

Recordemos que el motivo inicial de crear una herramienta de simulación, es continuar realizando estudios sobre la técnica de incremento de frecuencia de envío de paquetes de reconocimiento (ACK) para acelerar el crecimiento de *cwnd*, enfoque propuesto por Arcia-Moret (2009) en [5] y continuos estudios descritos en Díaz et al. [22], en Ciminieri et al. [23] y publicados en [4] y [9] respectivamente.

3.1. Simulador por eventos discretos

$\mu\mathcal{SIM}$ fue construido con el propósito de establecer un estudio comparativo en diversos escenarios de transmisión de datos. Para esto se ha desarrollado una herramienta simple que simula una red con un único cuello de botella. Se ha creado usando el framework **QT**, lenguaje **C++** , **Gnuplot** y **NAM**, como esta descrito en el capítulo 2.

La respuesta mostrara el comportamiento durante toda la transmisión, realizando una traza que mostrara el crecimiento de la ventana de congestión por parte del emisor, el *Throughput*, el *Goodput* y además un log (registro) donde se muestra detalladamente los valores por cada RTT.

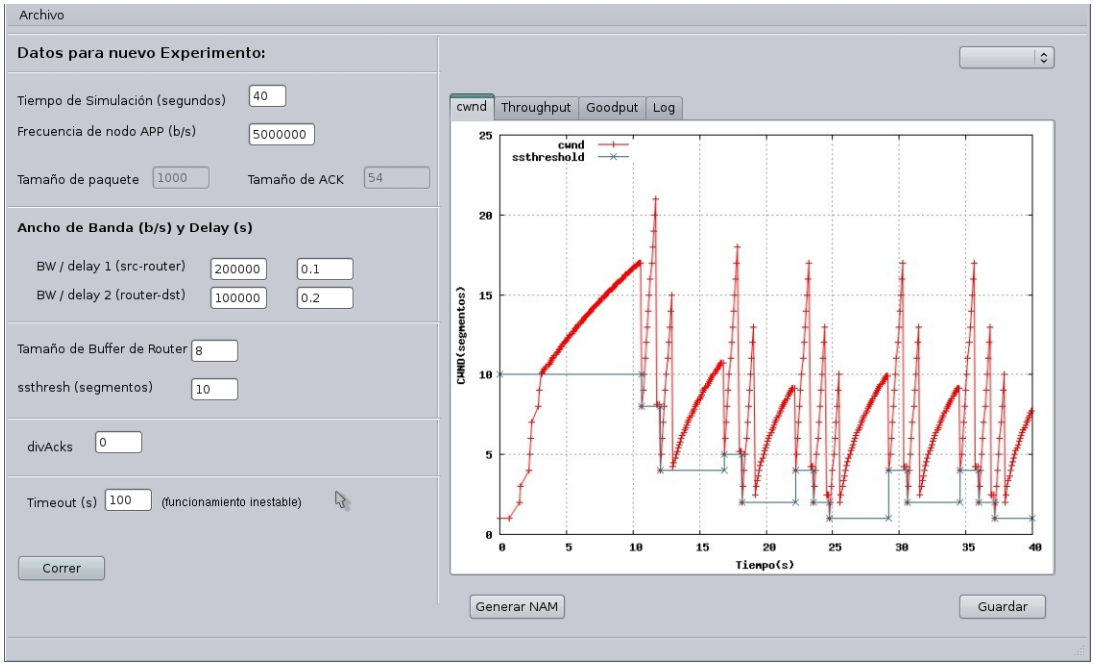

Figura 3.1: Interfaz gráfica de usuario del simulador $\mu\mathcal{SIM}$

3.1.1. MicroSim ($\mu\mathcal{SIM}$)

MicroSim ($\mu\mathcal{SIM}$) es un simulador de redes a eventos discretos, creado con fines didácticos, para mostrar el comportamiento detallado en instantes de tiempo de la vida de una transmisión de datos en una red, estos instantes son marcados por la ocurrencia de un evento en la simulación. El motor de $\mu\mathcal{SIM}$ es la clase **ListScheduler**, que se encarga de llevar una lista de eventos de forma organizada temporalmente, además de planificar la futura ejecución de otros eventos.

además de los eventos y el planificador de eventos, existen otros elementos que forman parte de la simulación en $\mu\mathcal{SIM}$, el objeto fundamental que viaja en una transmisión corresponde al segmento de dato que desde ahora se llamará PDU (Unidad de Datos de Protocolo). Al viajar a través de la red, las distintas capas modificaran la estructura y agregaran información, para poder ser comunicada por las diferentes capas de modelo OSI.

En una red existen objetos que generan PDUs y objetos que reciben PDUs, a estos objetos los llamaremos en general **Nodos**, en una red simple siempre debe existir un

nodo fuente y un nodo destino. Cada nodo contiene un **buffer**, para mantener los PDUs que entran al nodo.

Los **nodos** estarán conectados entre si por un **Enlace** o medio, por el cual será propagados los PDUs. El enlace tiene dos atributos principales, el ancho de banda (**BW**[1]), que es medido en bits por segundo (b/s), y el retardo (**delay**) que es medido en segundos (s).

Los **eventos** están compuestos por dos nodos (fuente y destino), PDU transmitido, el tipo de evento que está ocurriendo y el tiempo en segundos (s) que ocurre el evento.

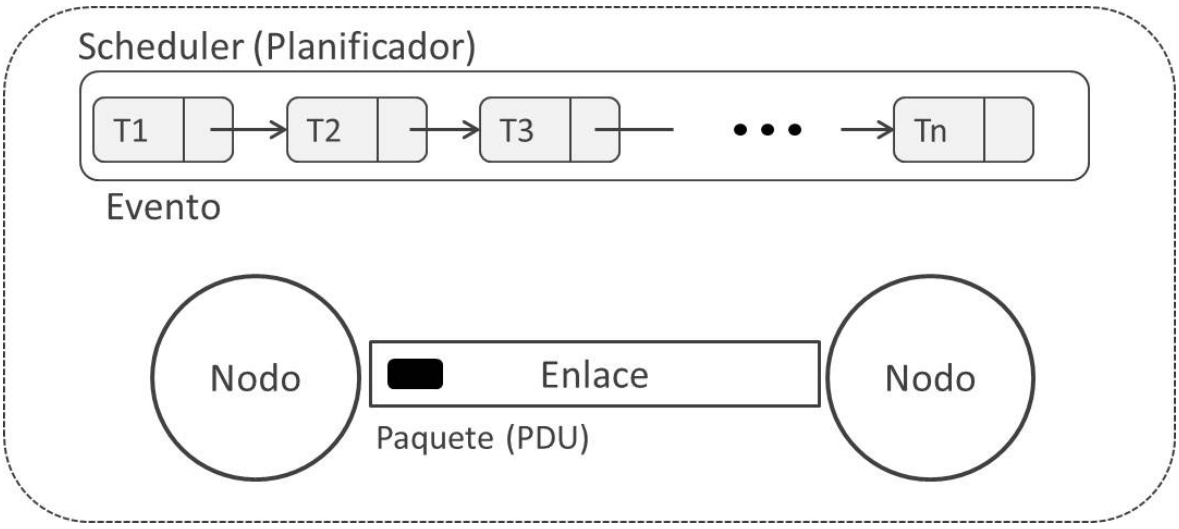

Figura 3.2: Componentes de $\mu\mathcal{SIM}$

3.1.2. Manejo de Capas

Después de definir bien cada uno de los elementos principales de $\mu\mathcal{SIM}$, pasamos a definir como van a estar organizados cada uno de ellos. Para cumplir con el modelo a capas OSI, recordemos que cada capa tiene su función independiente dentro del modelo, pero cada una de ellas está conectadas para formar un sistema.

Podemos aprovechar la versatilidad en la definición de los elementos de $\mu\mathcal{SIM}$ para adaptarlos al modelo OSI fácilmente.

[1]Del ingles bandwidth.

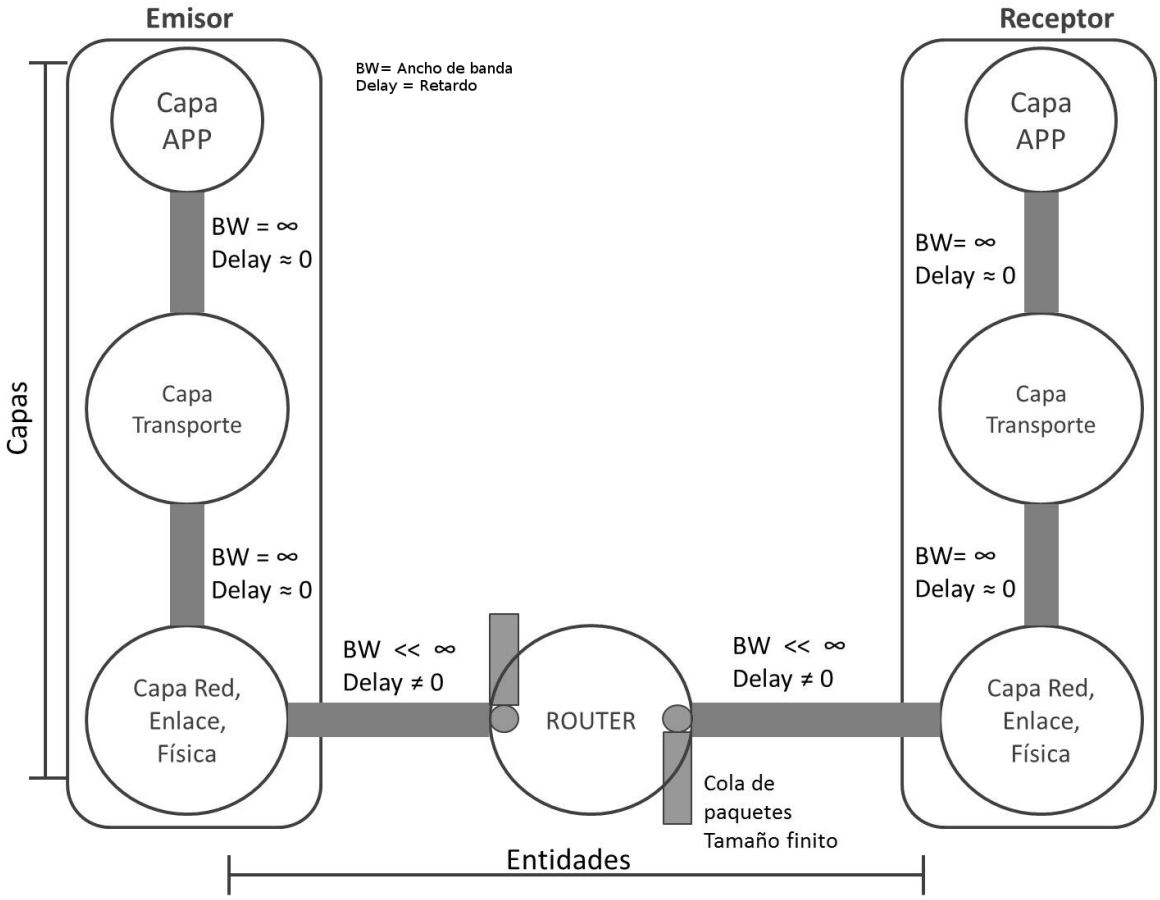

Figura 3.3: Capas en $\mu\mathcal{SIM}$

Como se observa en la figura 3.3, podemos definir nodos de $\mu\mathcal{SIM}$, como instancias de capa de OSI, los cuales van a tener asociados funciones bien detalladas y correspondientes a su cargo.

Así en el código C++ bastaría con una simple declaración como se muestra a continuación:

```
Node * a = new Node(1, APP, "app", QUEUESIZE);
Node * b = new Node(2, TRANSPORT, "transport layer", QUEUESIZE);
```

Primero se instancia los nodos que corresponde a los extremos de una red, en el ejemplo se crea un nodo aplicación que genera datos y un nodo transporte que va a servir como sumidero de datos, tenemos dos nodos instanciados con los siguientes parámetros: id, tipo de nodo, etiqueta y tamaño de cola igual a QUEUESIZE.

```
Link * a_to_b = new Link(true, BW, DELAY, a, b, TYPE);
a->add_interface(a_to_b);
```

Luego se instancia el medio que va a conectar a los dos nodos (a y b) con ancho de banda igual a BW y retraso igual a DELAY. Después se asigna el medio al nodo emisor de datos.

Para iniciar el flujo de datos, se llama a la función, la cual es una rutina que genera mensajes desde nodo APP.

```
start_app(APP_RATE, a, b);
```

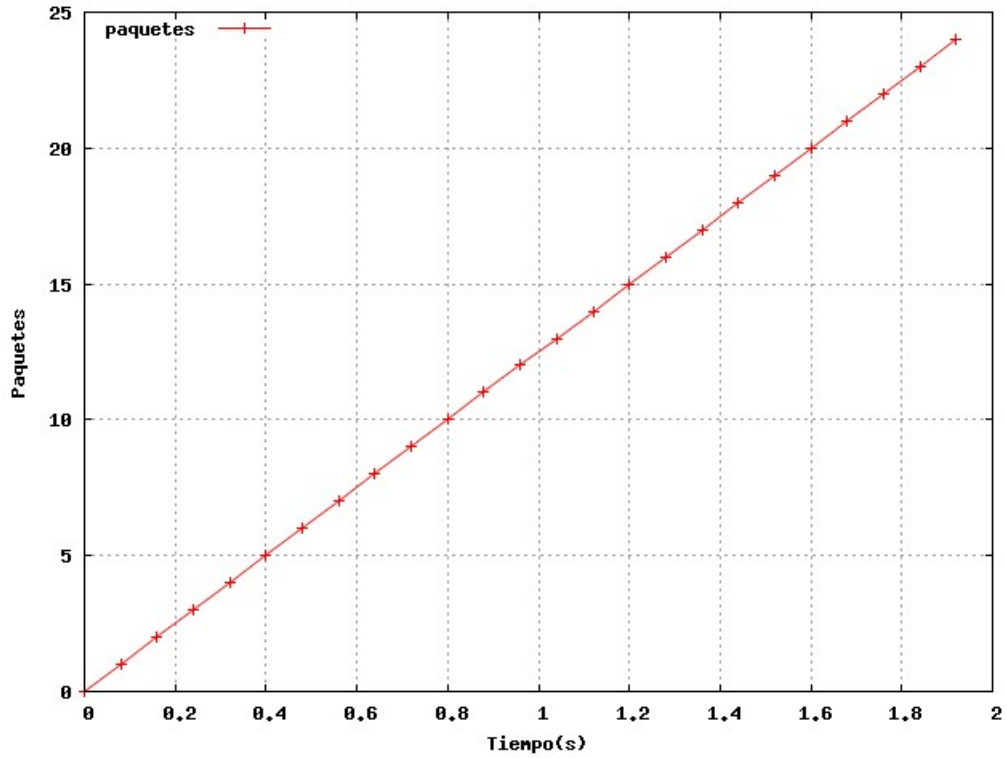

Figura 3.4: Envío simple de datos entre dos nodos

Para el caso que se ve en la figura 3.4, el valor de APP_RATE es 500 Kbps, para un tamaño de paquete igual a 1000 Bytes se puede ver que en aproximadamente 2 segundos el nodo receptor va a recibir 24 paquetes, recordemos que estos valores dependerá del ancho de banda y delay definido anteriormente.

La comunicación entre capas (nodos organizados verticalmente en la figura 3.3) se llevará a cabo con la conexión de un enlace en ambas direcciones, en donde se define un ancho de banda muy grande y un retraso cercano a 0. De esta manera modelamos OSI en $\mu\mathcal{SIM}$, aprovechado cada una de sus funcionalidades estándares, tanto para modelar el pase de PDU vertical, como el cambio de entes en horizontal. Debemos recordar que la misión principal de $\mu\mathcal{SIM}$ es mantener su estructura lo más simple posible.

3.2. Documentación

3.2.0.1. Diagrama de Clases

Diagrama de clases detallando el modelo de datos y como se relacionan entre ellos.

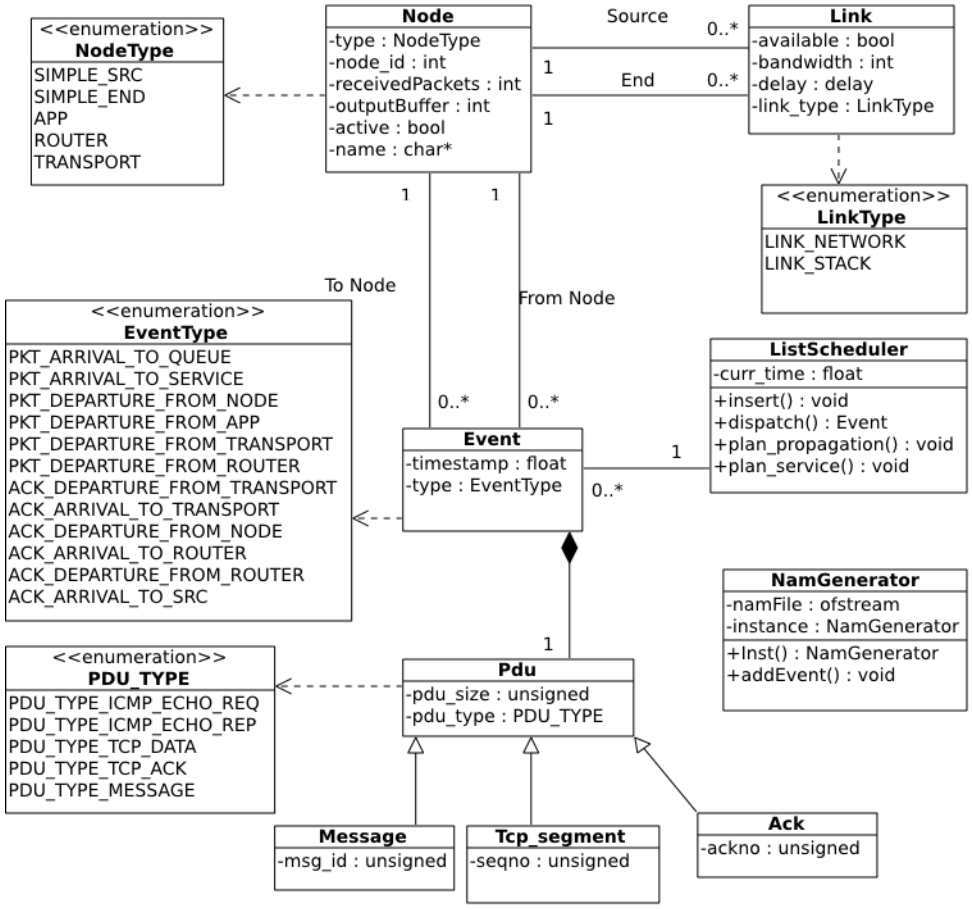

Figura 3.5: Diagrama UML de Clases

3.2.1. Casos de Uso

Los requerimientos básicos de funcionalidad necesarios para la interfaz gráfica de usuario, los detallaremos en cada uno de los casos de usos que se mostraran a continuación.

Se considera actor usuario a toda persona que tenga acceso a $\mu\mathcal{SIM}$, y aplicará los casos de usos que se definen a continuación.

3.2.1.1. Definición de casos de uso

Código CU	CU-01
Nombre CU	Crear nueva Sesión
Descripción	Crea nueva sesión de experimentos
Precondición	
Actor	Usuario
Flujo Normal	1. El usuario selecciona Archivo 2. El usuario presiona Abrir 3. Aparece una nueva ventana y el usuario selecciona la ruta donde desea crear nueva sesión 4. El usuario escribe el nombre de la nueva sesión 5. El usuario presiona Guardar
Postcondicón	

Código CU	CU-02
Nombre CU	Cargar Sesión
Descripción	Cargar sesión creada y guarda anteriormente
Precondición	Tener una sesión creada
Actor	Usuario
Flujo Normal	1. El usuario selecciona Archivo 2. El usuario presiona en Abrir 3. Aparece una nueva ventana 4. El usuario selecciona la ruta donde esta el archivo de sesión guardado 5. El usuario presiona guardar 6. El Sistema inmediatamente llenará la caja de selección de experimento. 7. Para visualizar un experimento cargado, el usuario debe presionar el ítem deseado de la caja de selección de experimento
Postcondicón	

Código CU	CU-03
Nombre CU	Guardar sesión
Descripción	Guarda el estado de la sesión actual
Precondición	
Actor	Usuario
Flujo Normal	1. El usuario selecciona Archivo 2. El usuario hace presiona Guardar 3. El Sistema Guardará todos los datos necesarios en un archivo local
Flujo Alternativo	Si el usuario presiona Guardar y la sesión está vacía 1. Aparece una nueva ventana que dice no poder guardar experimento vacío 2. Usuario hace presiona el botón OK
Postcondicón	

Código CU	CU-04
Nombre CU	Ingresar parámetros de Simulación
Descripción	Para ingresar cada uno de las variables que afectan a la simulación
Precondición	
Actor	Usuario
Flujo Normal	1. En el panel de variables, el usuario debe llenar cada uno de los campos existentes con valores correspondientes
Postcondicón	

Código CU	CU-05
Nombre CU	Ejecutar Simulación
Descripción	Ejecuta la simulación con valores actuales
Precondición	Tener todos los campos llenos en el panel de variables
Actor	Usuario
Flujo Normal	1. El usuario presiona el botón ejecutar 2. Se muestra en el panel de resultados la respuesta de la simulación
Postcondicón	

Código CU	CU-06
Nombre CU	Observar Gráfica de crecimiento de ventana de congestión
Descripción	Para visualizar la gráfica de crecimiento de ventana de congestión, durante la vida de la simulación
Precondición	Haber ejecutado la simulación
Actor	Usuario
Flujo Normal	1. El usuario presiona la pestaña *cwnd* ubicada en la sección de la derecha de la ventana principal. 2. Se muestra en el recuadro principal la gráfica de crecimiento de ventana de congestión.
Postcondicón	

Código CU	CU-07
Nombre CU	Observar Gráfica de Rendimiento de la red
Descripción	Para visualizar la gráfica de rendimiento de la red
Precondición	Haber ejecutado la simulación
Actor	Usuario
Flujo Normal	1. El usuario presiona la pestaña Throughput ubicada en la sección de la derecha de la ventana principal. 2. Se muestra en el recuadro principal la grafica de rendimiento de la red o throughput.
Postcondicón	

Código CU	CU-08
Nombre CU	Observar registro de ejecución de simulación
Descripción	Para visualizar el registro o log de ejecución de la simulación
Precondición	Haber ejecutado la simulación
Actor	Usuario
Flujo Normal	1. El usuario presiona la pestaña Log ubicada en la sección de la derecha de la ventana principal. 2. Se muestra en el recuadro principal el registro de ejecución de la simulación.
Postcondicón	

Código CU	CU-09
Nombre CU	Generar traza con formato NAM
Descripción	Para generar traza completa de la simulación para poder ser visualizada en NAM
Precondición	Haber ejecutado la simulación
Actor	Usuario
Flujo Normal	1. Usuario presiona el botón Generar Archivo NAM. 2. Aparece una nueva ventana donde se seleccionara la ruta destino para el archivo compatible con la herramienta NAM, para visualizar la simulación. 3. Usuario presiona el botón guardar.
Postcondicón	

Código CU	CU-10
Nombre CU	Observar Gráfica de Goodput de la red
Descripción	Para visualizar la gráfica de Goodput de la red
Precondición	Haber ejecutado la simulación
Actor	Usuario
Flujo Normal	1. El usuario presiona la pestaña Goodput ubicada en la sección de la derecha de la ventana principal. 2. Se muestra en el recuadro principal la gráfica de Goodput.
Postcondicón	

3.2.1.2. Diagrama de casos de uso

Diagrama correspondiente para los casos de uso definidos.

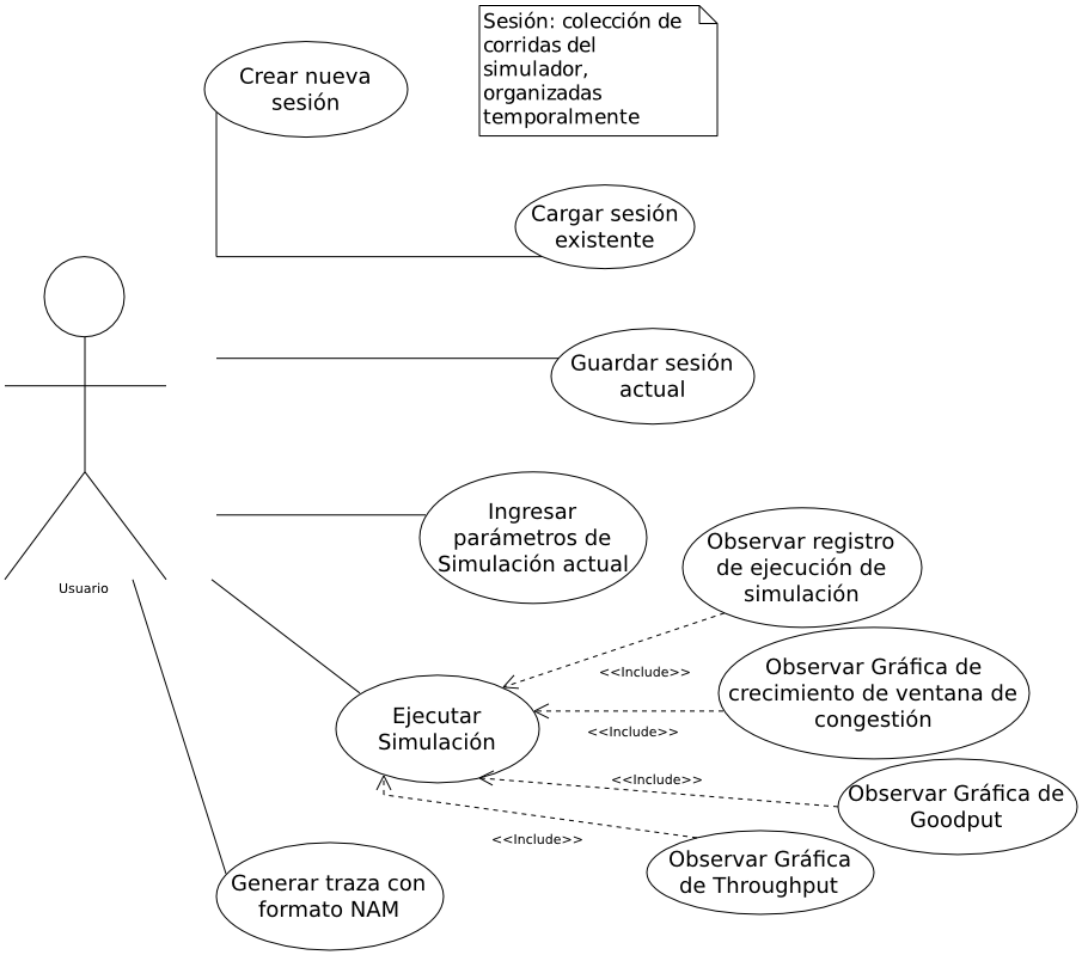

Figura 3.6: Diagrama de casos de uso

3.2.2. Especificación del flujo de datos

Para modelar el comportamiento del pase de mensajes en una red, debemos comprender a detalle el flujo de trabajo que define sus dinámicas y como actúan los elementos dependiendo de su entorno.

3.2.2.1. Máquina de Estados para un Paquete

El paquete es la partícula esencial de nuestro modelo, pues es el elemento que da sentido y forma a los eventos de $\mu\mathcal{SIM}$, definimos la conducta que tiene un paquete durante su vida dentro de una transmisión por una red.

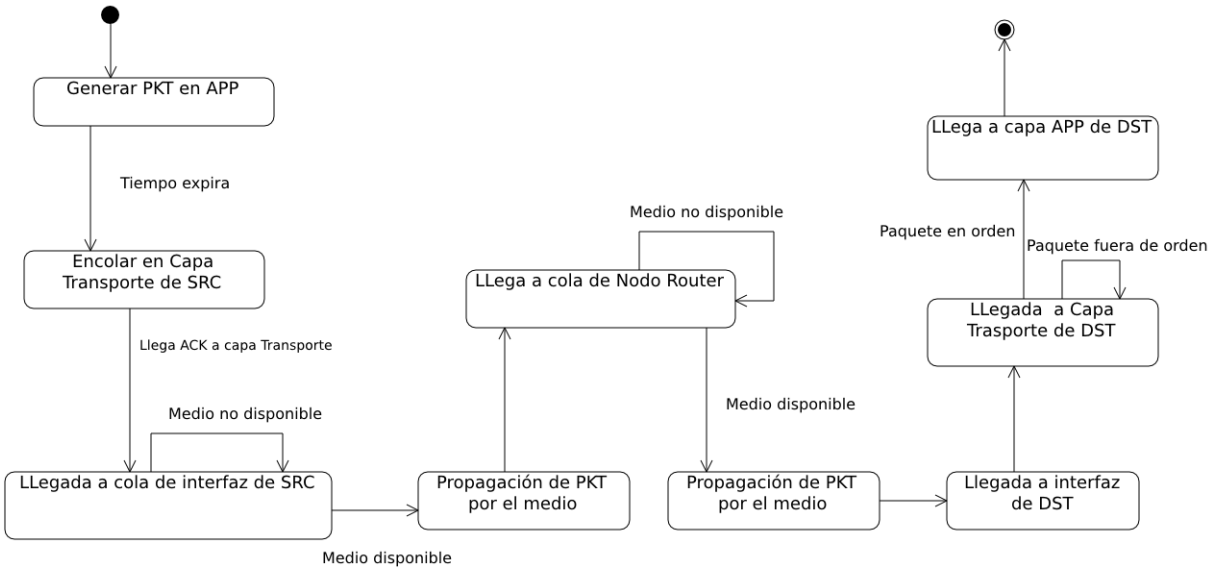

Figura 3.7: Estados de un paquete de Datos

3.2.2.2. Máquina de Estados para un ACK

Estados de un paquete ACK, recordemos que un ACK va a permitir que exista continuidad en el flujo de datos dentro de la red.

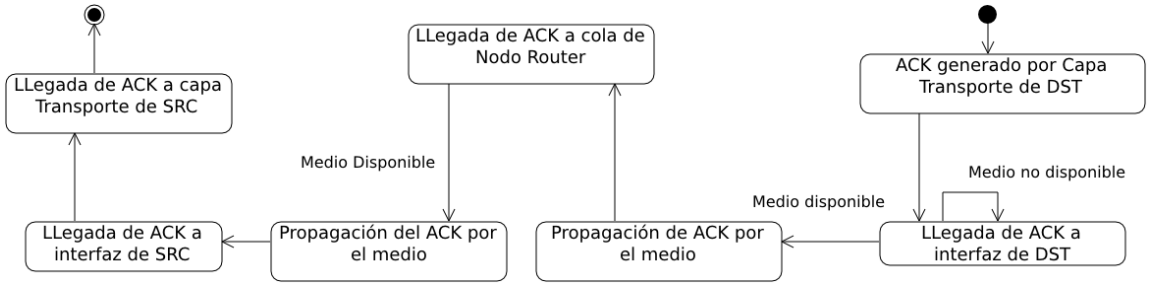

Figura 3.8: Estados de un ACK

3.2.2.3. Diagrama de Secuencia de $\mu\mathcal{SIM}$

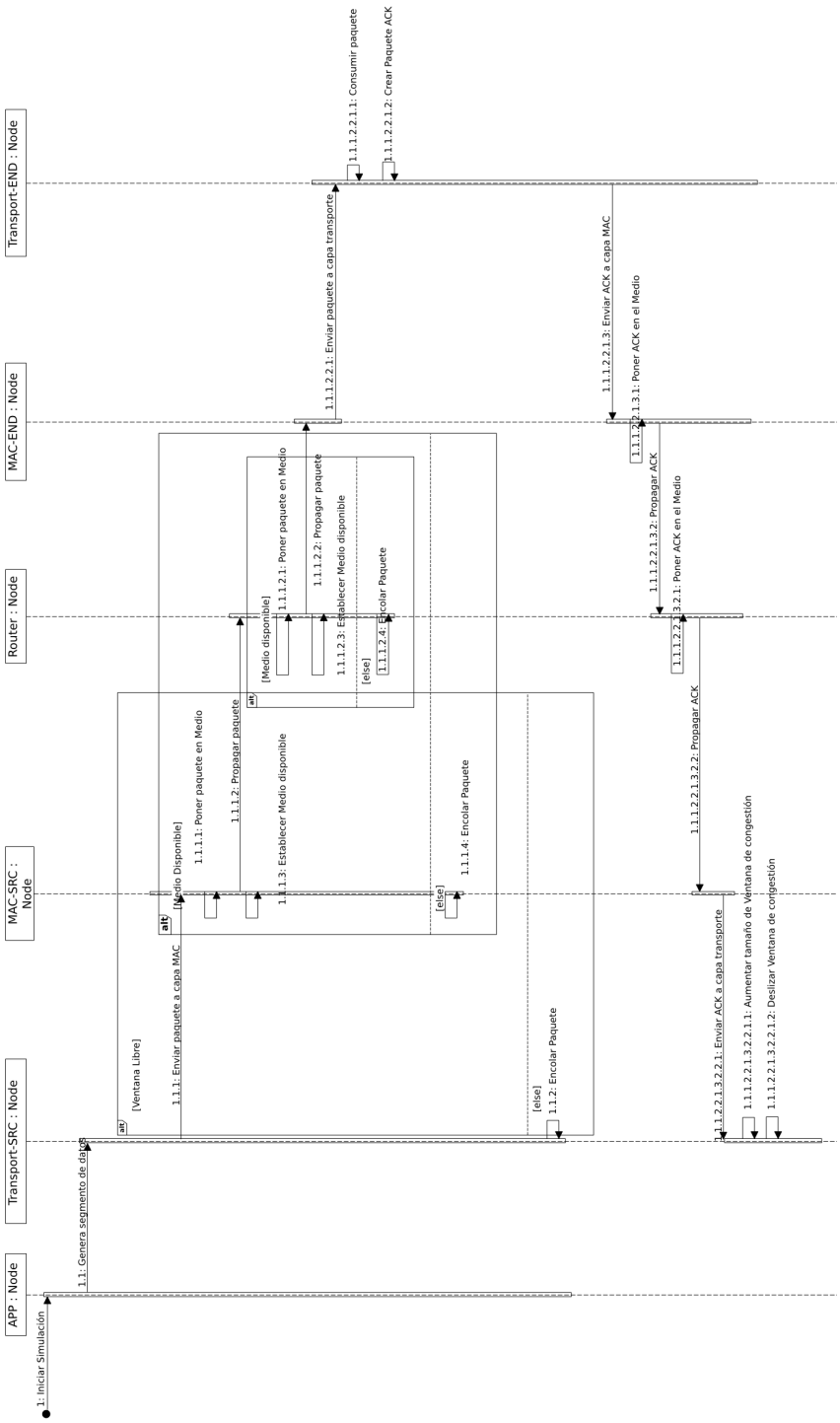

Figura 3.9: Diagrama de Secuencia

3.2.2.4. Máquina de Estados para TCP

Hay que recordar que el punto principal del trabajo a realizar, es tener un modelo fiel de la dinámica de TCP, ya que los estudios comparativos estarán concentrados en dicha dinámica. Por esto se engloba toda la teoría referente a TCP (detallada en el capítulo 2 del documento), en un diagrama de máquina de estados, donde cada estado corresponde a cada fase principal de control de congestión de TCP.

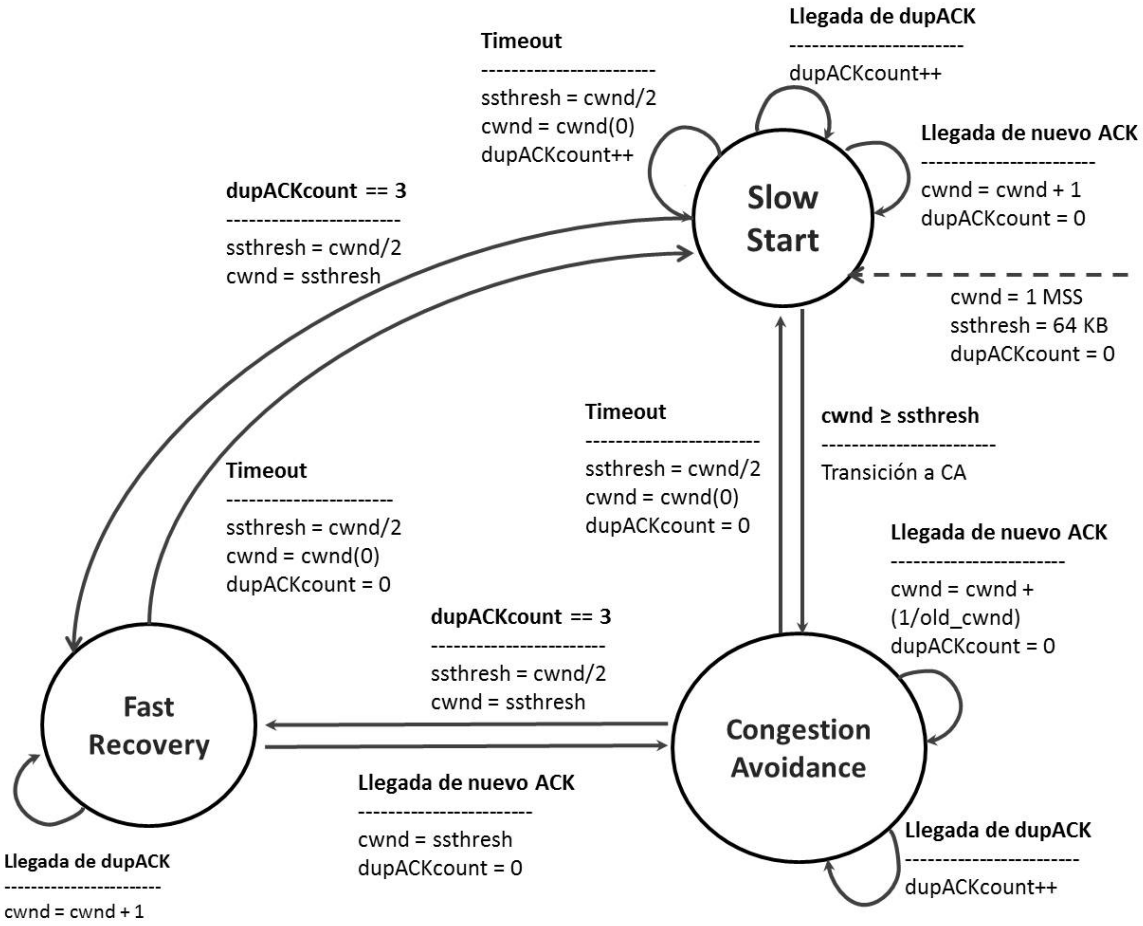

Figura 3.10: Diagrama de Máquina de Estados para TCP en el emisor

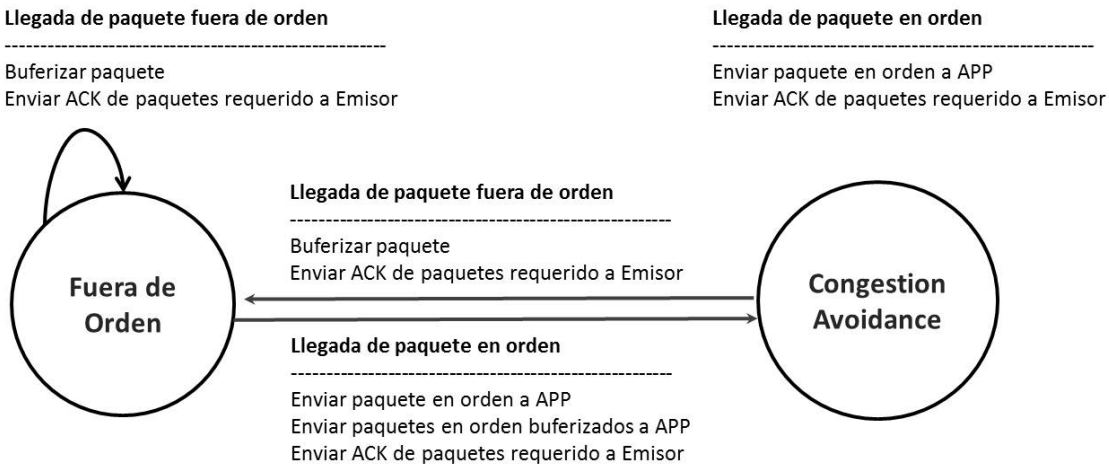

Figura 3.11: Diagrama de Máquina de Estados para TCP en el receptor

Capítulo 4

Implementación de $\mu\mathcal{SIM}$

4.1. Manejo de Eventos

En $\mu\mathcal{SIM}$ los **eventos** marcaran todos los puntos críticos que forman parte de la vida de una simulación y esencialmente van a describir el estado en el que se encuentra un PDU, en un instante de tiempo. Vale recordar, que como vimos en el capítulo 3, cada evento está compuesto por el **tiempo** en el que ocurrirá, el **PDU** asociado, el **nodo origen** y **nodo destino**, detallaremos cada uno de estos eventos, y como se manipulan.

En esencia un evento es un PDU o una llamada a función para ejecutar una tarea especifica producto de un temporizador o una llamada periódica.

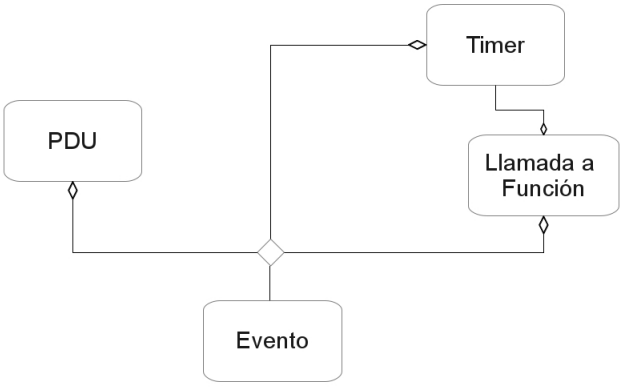

Figura 4.1: Detallando un Evento

4.2. Eventos desde el Emisor hacia el Receptor

En esta sección trataremos el envío de datos desde la capa Aplicación hasta el receptor final.

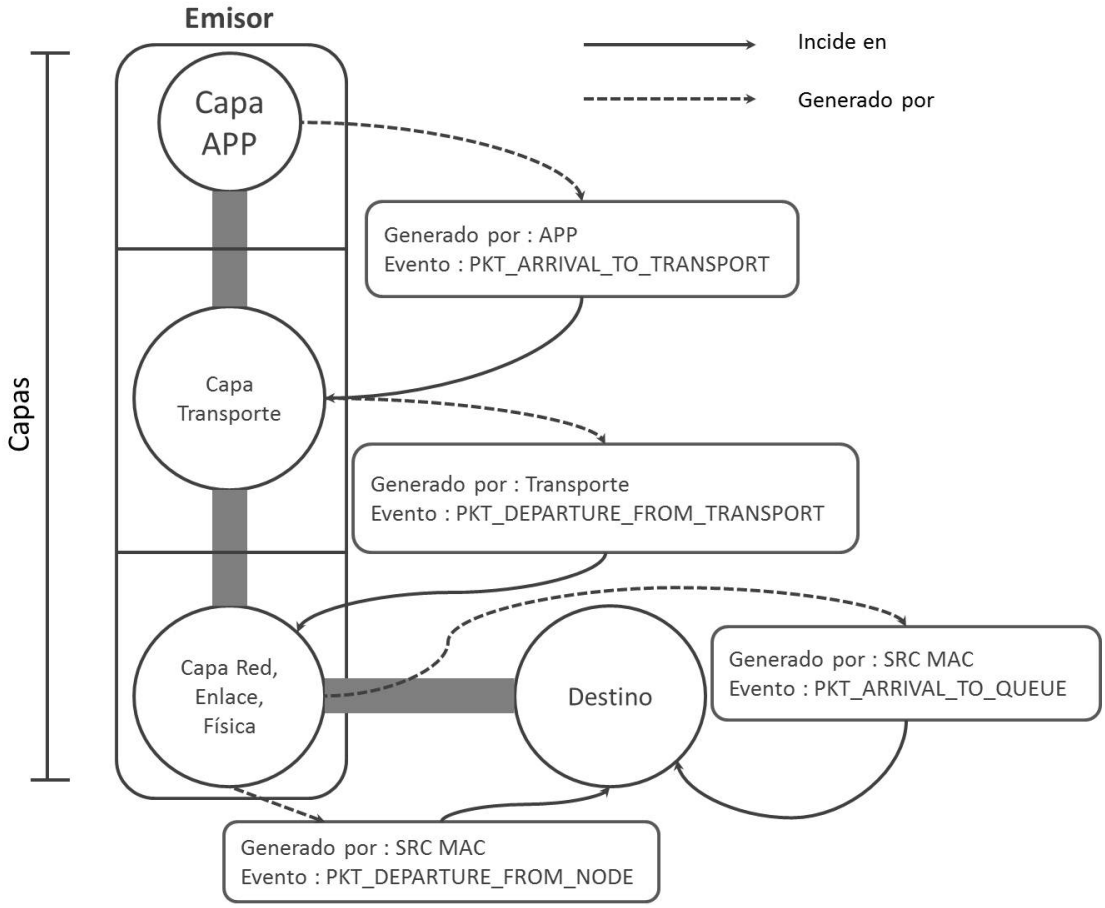

Figura 4.2: Eventos que ocurren durante la transmisión de un paquete

4.2.1. Desde la Aplicación hacia la capa Transporte

El **Algoritmo** 1, asociado al evento PKT ARRIVAL TO TRANSPORT, manipula los mensajes enviados desde capa aplicación con destino a capa transporte. Observamos que se debe segmentar el mensaje en paquetes de tamaño menor o igual al máximo tamaño del paquete declarado al inicio de la simulación. Si suponemos que ya se ha establecido una conexión entre origen y destino (3-way handshake), y no se ha enviado

todavía un paquete de datos, entonces podemos afirmar que estamos en la primera iteración de la instancia de simulación. Si estamos en la primera iteración entonces en este momento, se debe enviar cantidad de paquetes igual al tamaño de ventana de congestión inicial, a capa física, para planificar su puesta en el medio y luego ser propagada.

Algorithm 1 Desde la Aplicación hacia la capa Transporte

Require: msg = pdu asociado evento actual

1: $msgSize$ = tamaño de msg
2: **while** $msgSize > 0$ **do** ▷ Segmentamos msg, y lo encolamos
3: $msgSize = msgSize - PACKETSIZE$
4: $segmento = nuevoTcp\_Segment(PACKETSIZE, seqno)$
5: Encolar $segmento$ en Nodo Transporte
6: **end while**
7: **if** es primera iteración **then**
8: $queueSize$ = obtener tamaño cola transporte
9: **for** $i = 1$ a $cwnd\_inicial$ **do**
10: **if** $queueSize > 0$ **then**
11: $paquete$ = cabeza de cola nodo transporte
12: $Planificar\_Evento$ $(PKT\_DEPARTURE\_FROM\_TRANSPORT$, paquete, de capa Aplicación, hacia capa Transporte)
13: **end if**
14: **end for**
15: **end if**.

4.2.2. Desde la capa Transporte hacia la capa MAC de emisor

La capa Transporte es la capa del modelo **OSI** más elaborada en $\mu\mathcal{SIM}$, ya que es el principal objeto de estudio en este trabajo, para modelar su comportamiento, nos basamos en el algoritmo **New Reno**[15] de TCP. Capa transporte se divide en dos partes principales, la salida de TCP, que controla el flujo de salida de paquetes del emisor de datos y la entrada de TCP, que se encarga de, verificar y dar a conocer al emisor la llegada correcta de los paquetes a su destino. En los **Algoritmos** 2, 3 y 4, se detalla como se manipula el evento ACK ARRIVAL TO TRANSPORT, el cual es generado por la llegada de un ACK a capa transporte, desde capa MAC de SRC. El algoritmo que genera los ACKs será especificado más adelante.

Recordemos que TCP tiene dos fases principales que dictaran el crecimiento

Algorithm 2 Desde la capa Transporte hacia la capa MAC de emisor, parte 1 (Slow Start)

Require: ack enviado

if Fase Slow Start **then**
  if $cwnd == SSthreshold$ **then**
    Activa fase Congestion Avoidance
  end if
  bool $reconocido$ = reconocer(ack enviado)
  if reconocido **then**
    $cwnd++$ ▷ aumentar tamaño de ventana
    Liberar espacio en Ventana
    while Espacio disponible en Ventana **do**
      $sending\_segment$ = Desencolar siguiente paquete a transmitir
      guardar $sending\_segment$ en buffer de paquetes no reconocidos
      $Planificar\_Evento$ ($PKT\_DEPARTURE\_FROM\_TRANSPORT$, $sending\_segment$, de capa Transporte, hacia capa MAC)
    end while
  else ▷ Posible pérdida de paquete
    if $current\_ack == last\_ack$ **then**
      contador ACK duplicado += 1
      if contador ACK duplicado == 3 **then** ▷ Pérdida de paquete detectado
        $lost\_segment\_id$ == número de segmento perdido
        Buscar segmento perdido en buffer de paquetes no reconocidos
        $Planificar\_Evento$ ($PKT\_DEPARTURE\_FROM\_TRANSPORT$, $lost\_segment$, de capa Transporte, hacia capa MAC)
        $cwnd = cwnd/2$
        $SSthreshold = cwnd$
        Activar fase Fast Recovery
      end if
    end if
  end if
end if

de la ventana de congestión de una conexión [19], **Slow Start** y **Congestion Avoidance**, además existe una fase intermedia llamada **Fast Recovery**, donde TCP se recuperará de una pérdida de paquete.

Detallamos el comportamiento de TCP en $\mu\mathcal{SIM}$, durante fase Slow Start, en el **Algoritmo** 2. Durante Slow Start, la ventana de congestión crecerá en 1 por cada ACK recibido y se abrirá un espacio en la ventana para enviar nuevos segmentos de datos. En total, se podrá enviar 2 segmentos cada vez que llegue un ACK a capa Transporte del emisor, esto ocurrirá si y solo si, el ACK que llega, reconoce uno o más paquetes enviados anteriormente. En el caso de reconocer un paquete enviado, este se descartará del buffer de paquetes no reconocidos y se procederá a hacer el envío de nuevos paquetes, si no, es posible que un paquete se haya perdido. En este caso, no crecerá la ventana de congestión, ni se enviará nuevos paquetes, solo se aumenta un contador las veces que llegue el mismo ACK. Los ACK repetidos son llamados **DupACKs**, y forman parte del mecanismo para control y verificación de paquetes perdidos. Según [19], al recibir tres dupACKs seguidos, se considera que el paquete indicado en el ACK se ha perdido, y debe hacerse la retransmisión del mismo de inmediato. Hecho esto, se pasa a fase de recuperación o Fast Recovery, en este momento, el tamaño de ventana de congestión se reduce a la mitad, junto al umbral de Slow Start. Si el tamaño de Ventana de Congestión llega a ser igual o mayor que el valor de umbral de Slow Start(SSthreshold), entonces, se pasa automáticamente a fase Congestion Avoidance.

El **Algoritmo** 3 describe la implementación de la fase **Fast Recovery** basada en **New Reno**[15], hay que tener en cuenta, que se debe tratar de mantener la transmisión de paquetes de una manera continua, por eso, al reenviar el paquete perdido y mientras se mantenga en Fast Recovery, el envío de paquetes seguirá siempre y cuando llegue un ACK a capa transporte, en este momento pueden ocurrir dos casos. Primero, que el ACK que llega sigue pidiendo el reenvío del paquete recientemente reenviado, en este caso, cada uno de los dupACKs hará crecer la ventana de congestión en 1 MSS.

Segundo, si por el contrario, el ACK que llega, reconoce el paquete perdido y pide un paquete nuevo, entonces la ventana de congestión se hace nuevamente igual al valor

Algorithm 3 Desde la capa Transporte hacia la capa MAC de emisor, parte 2 (Fast Recovery)

```
30: if Fase Fast Recovery then
31:     if current_ack == lost_segment_id then
32:         cwnd = SSthreshold
33:         Activar fase Congestion Avoidance
34:     else
35:         cwnd + +
36:     end if
37:     while Espacio disponible en Ventana do
38:         sending_segment = Desencolar siguiente paquete a transmitir
39:         Guardar sending_segment en buffer de paquetes no reconocidos
40:         Planificar_Evento (PKT_DEPARTURE_FROM_TRANSPORT,
41: sending_segment, de capa Transporte, hacia capa MAC)
42:     end while
43: end if
```

que tenía cuando entró a Fast Recovery y entra automáticamente a fase Congestion Avoidance.

En fase Congestion Avoidance el crecimiento de la ventana de congestión es bastante más lento que en Slow Start. Este se caracteriza por su dinámica basada en **AIMD (additive-increase/multiplicative-decrease)**. En el**Algoritmo** 4 se muestra como se implementa el crecimiento de *cwnd* durante la fase Congestion Avoidance, según [19], en Congestion Avoidance, cada vez que llega un ACK, la ventana de congestión crecerá en $1/cwnd$, es decir, cada **RTT**, *cwnd* crecerá en 1 segmento. Se sigue entonces enviando la cantidad de paquetes igual al espacio disponible en *cwnd*, y se usa la técnica de los 3 DupAcks para detectar pérdidas de paquetes.

El evento es PKT DEPARTURE FROM TRANSPORT, y tiene atribuido como nodo fuente capa transporte de SRC, nodo destino capa MAC de SRC, siendo el PDU asociado el nuevo paquete que se transmitirá.

4.2.3. Desde la capa MAC hacia el enlace

Módulo de $\mu\mathcal{SIM}$ donde se desarrolla el funcionamiento de la capa MAC. Hay que recordar que en el estado actual de $\mu\mathcal{SIM}$, cada host solo tiene una interfaz de salida y entrada, a esta interfaz se le atribuye una cola de salida de paquetes ordenados por

Algorithm 4 Desde la capa Transporte hacia la capa MAC de emisor, parte 3 (Congestion Avoidance)

```
44: if Fase Congestion Avoidance then
45:     bool reconocido = reconocer(ack enviado)
46:     if reconocido then
47:         Liberar espacio en Ventana
48:         cwnd = cwnd + (1/old_cwnd)
49:         if cwnd == old_cwnd then
50:             Liberar espacio en Ventana
51:             old_cwnd = cwnd
52:         end if
53:         while Espacio disponible en Ventana do
54:             sending_segment = Desencolar siguiente paquete a transmitir
55:             Guardar sending_segment en buffer de paquetes no reconocidos
56:             Planificar_Evento (PKT_DEPARTURE_FROM_TRANSPORT,
57: sending_segment, de capa Transporte, hacia capa MAC)
58:         end while
59:     else                                          ▷ Posible pérdida de paquete
60:         if current_ack == last_ack then
61:             contador ACK duplicado += 1
62:             if contador ACK duplicado == 3 then ▷ pérdida de paquete detectado
63:                 lost_segment_id == Número de segmento perdido
64:                 Buscar segmento perdido en buffer de paquetes no reconocidos
65:                 Planificar_Evento (PKT_DEPARTURE_FROM_TRANSPORT,
66: sending_segment, de capa Transporte, hacia capa MAC)
67:                 cwnd = cwnd/2
68:                 SSthreshold = cwnd
69:                 Activar fase Fast Recovery
70:             end if
71:         end if
72:     end if
73: end if
```

tiempo de llegada y se le denomina *forward queue*. El algoritmo que implementa el funcionamiento de este estado del simulador es el **Algoritmo** 5. Su funcionamiento se bastante simple, si *forward queue* está vació significa que el paquete nuevo puede ponerse en el medio para ser transmitido. Si no, se encolaría en *forward queue* para esperar su futura puesta en el medio. Para planificar la puesta en medio se usa la ecuación 4.1.

$$T_e = t + \frac{Psize}{BW} \tag{4.1}$$

Algorithm 5 Desde la capa MAC hacia el enlace

1: **if** Cola de salida vacía **then**
2: *Planificar_Servicio* ($PKT\_DEPARTURE\_FROM\_NODE$, paquete asociado al evento, de capa MAC, hacia nodo destino)
3: **end if**
4: Encolar paquete en cola de salida

4.2.4. Propagar desde la capa MAC hacia destino

Algoritmo 6 es activado por el evento PKT DEPARTURE FROM NODE, significando que el paquete ya está puesto en el medio y ya está listo para ser propagado. Para ello, simplemente desencolamos el paquete de *forward queue* y planificamos su llegada al destino. A continuación solo si *forward queue* no está vacía, entonces se planifica su puesta en el medio.

Algorithm 6 Propagar desde capa MAC hacia destino

1: Desencolar paquete de cola de salida
2: Planificar propagación del paquete hacia el destino
3: **if** Cola de salida no vacía **then**
4: *Planificar_Servicio* ($PKT\_DEPARTURE\_FROM\_NODE$, paquete en la cabecera de la cola, de capa MAC, hacia nodo destino)
5: **end if**

4.3. Nodo Enrutador y sus Eventos Asociados

El nodo enrutador (o router) corresponder al nodo intermedio entre nodo origen y nodo destino. Actualmente su fin principal corresponde a generar el cuello de botella de la red. Normalmente el ancho de banda entre el nodo origen y el router es mayor que ancho de banda entre el router y el nodo destino. Esto es para hacer que la cola de paquetes que van a ser enviados de router a destino, crezca más rápido de lo que estos paquetes sean enviados, originando así un desborde de paquetes y por consecuencia, pérdida de paquetes.

El estado de desarrollo actual de $\mu\mathcal{SIM}$ no demanda un alto nivel de procesamiento en el nodo router, ni la existencia de complejidad alta en la implementación de la capa 3 del modelo OSI. Al existir 2 hosts en los extremos de la topología de la red y entre ellos va a haber comunicación continua, existirá enrutamiento estático donde:

Al Llegar	Al Salir
Todo PDU que llegue de Host SRC	Enviado a Host DST
Todo PDU que llegue de Host DST	Enviado a Host SRC

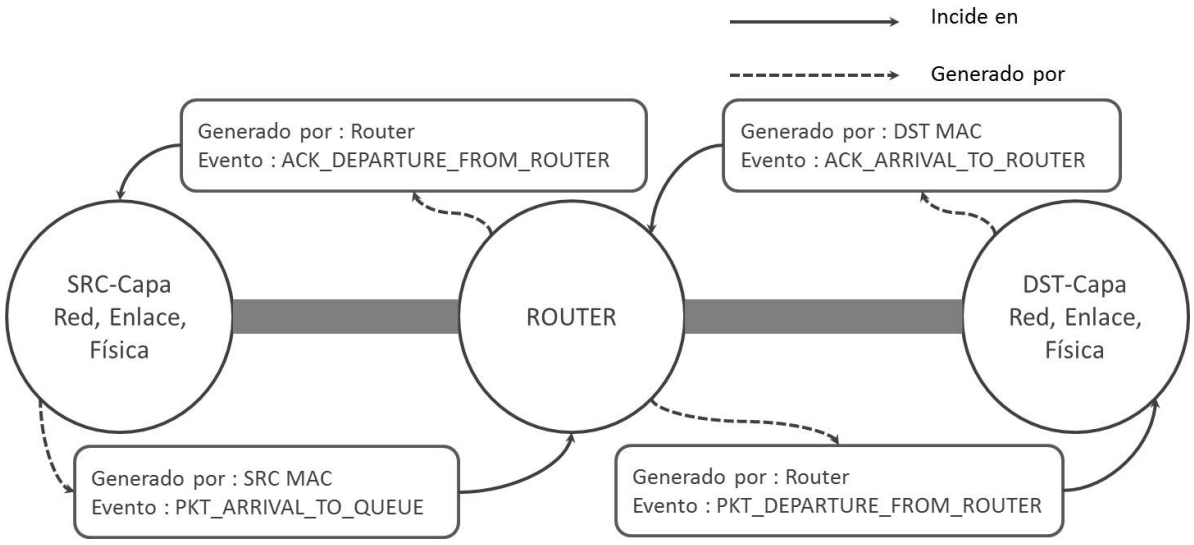

Figura 4.3: Eventos asociados a nodo Router

4.3.1. Desde Router hacia Enlace

Corresponde al algoritmo que genera el evento PKT DEPARTURE FROM ROUTER (**Algoritmo** 7). Si la *forward queue* del router está vacía, se planifica la puesta en el medio, si no, el paquete se encolará. En este momento hay que tener cuidado si la cola ha alcanzado su tamaño máximo, si es así, simplemente se desecha el paquete.

Algorithm 7 Desde Router hacia Enlace (de ida)

1: **if** Cola de salida vacía **then**
2: *Planificar_Servicio* ($PKT\_DEPARTURE\_FROM\_ROUTER$, paquete asociado al evento, de Router, hacia nodo receptor)
3: Planificar Servicio del evento
4: **else if** Cola de salida > tamaño máximo de cola **then**
5: Desechar paquete
6: **else**
7: Encolar paquete en cola de salida
8: **end if**

4.3.2. Propagar desde Router hacia la capa MAC de receptor

En el **Algoritmo** 8, repetimos la dinámica de propagación de paquetes puestos en el medio (explicados en el **Algoritmo** 6), esta vez planificando el evento PKT DEPARTURE FROM ROUTER.

Algorithm 8 Propagar desde Router hacia la capa MAC de receptor

1: Desencolar paquete de cola de salida
2: Planificar propagación del paquete
3: **if** Cola de salida no vacía **then**
4: *Planificar_Servicio* ($PKT\_DEPARTURE\_FROM\_NODE$, paquete en la cabecera de la cola, de capa MAC, hacia nodo receptor)
5: **end if**

4.4. Eventos asociados a los ACKs

Los ACKs reconocen la correcta llegada de cada uno de los paquetes de datos transmitidos por la red. Avisan al emisor de datos si se perdió algún paquete transmitido y permiten saber al emisor si la red está congestionada. Por esto, los ACKs son elementos importantes en una red y su comportamiento debe ser modelado detalladamente.

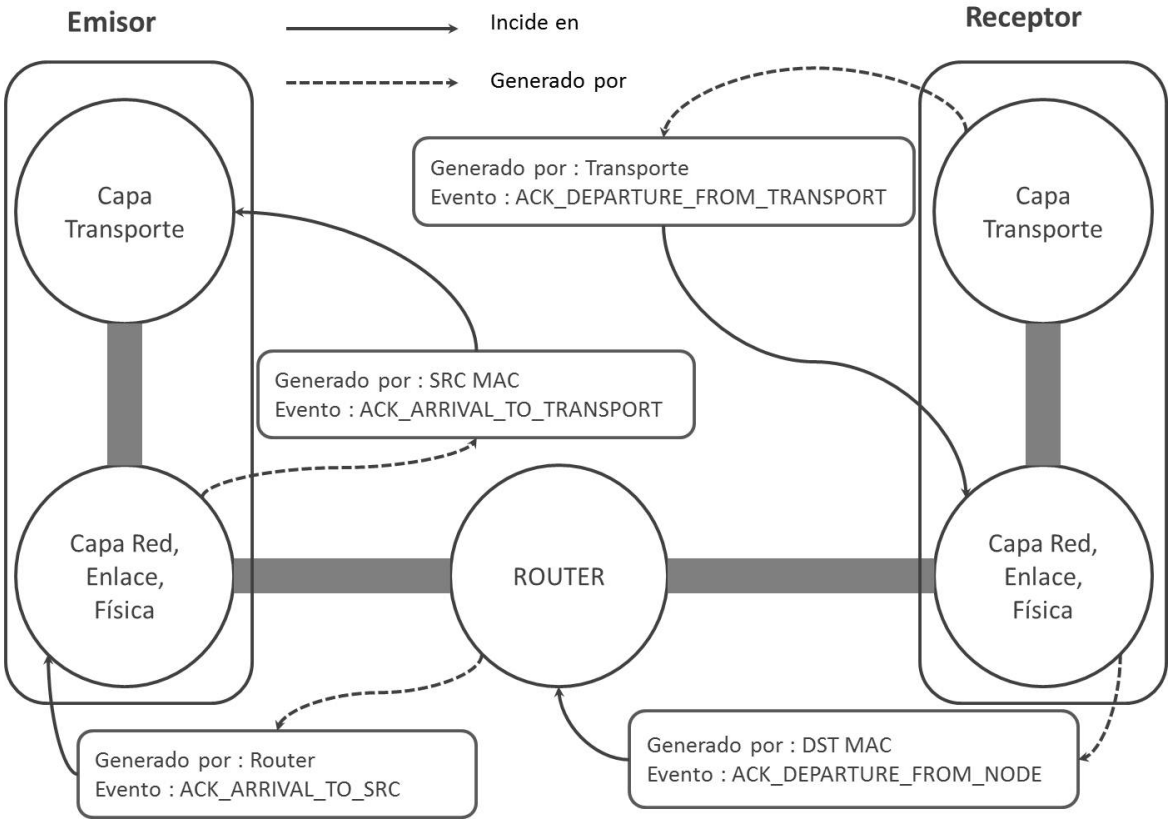

Figura 4.4: Eventos que ocurren durante la transmisión de un ACK

4.4.1. Desde capa MAC del receptor hacia capa Transporte del receptor

La creación de ACKs se inicia con la llegada de un paquete al host destino, cuando se captura la llegada de un paquete de datos, la capa MAC directamente lo envía a la capa Transporte para ser procesado.

Algorithm 9 In DST MAC to Transport Layer

1: $Planificar\_Servicio$ ($PKT\_ARRIVAL\_TO\_TRANSPORT$, paquete asociado al evento, de la capa MAC, hacia la capa transporte del receptor)
2: contador de paquetes recibidos ++

4.4.2. Desde la capa Transporte del receptor hacia la capa MAC del receptor

En el **Algoritmo** 10 se implementa el comportamiento que tiene $\mu\mathcal{SIM}$ al momento de la llegada de un paquete a capa Transporte. Aquí es donde se crea el ACK que reconocerá el paquete que acaba de llegar, y pedirá la transmisión del siguiente paquete. Pero antes de eso, hay que recordar que TCP debe garantizar el envío **ordenado** de paquetes, y si falta uno, se debe guardar cada paquete nuevo, hasta que llegue el paquete faltante. Para esto se debe mantener siempre conocimiento del paquete que debe llegar, si el paquete que llega no es el que se espera, simplemente se guarda en un buffer de manera temporal, hasta que este llegue.

Cuando finalmente llegue el paquete esperado, se envía los paquetes contenidos en el buffer a capa aplicación, verificando su correcto orden. Si no existe un orden al vaciar, entonces se deja de enviar y se espera que llegue el paquete esperado.

Siempre que llegue un paquete, se generará un ACK que pedirá el paquete que se espera en ese momento.

4.4.3. Implementación de envío de *divacks*

Recordemos que lo que se propone en [5], [9] y [4], es usar el mecanismo de división de ACK para acelerar el crecimiento de *cwnd*, por lo tanto no nos interesa por ahora el hecho de que cada uno de los *divacks*, va a reconocer una porción de dato enviado, básicamente para reducir la complejidad del código. Fuera de esta consideración, los *divacks* en $\mu\mathcal{SIM}$ tendran el mismo efecto que tiene un full-ACK.

En el **Algoritmo** 11, se observa como implementamos la técnica, donde, $nDivacks$ es la cantidad de *divacks* que se va a enviar por cada full-ACK, este valor se asigna al iniciar la simulación. Existe un bucle que va de 1 a $nDivacks$, y en cada iteración se creará un ACK nuevo el cual pedirá el numero de paquete *-1*, TCP en el extremo del

Algorithm 10 Desde capa Transporte del receptor hacia capa MAC del receptor

Require: Pi : Paquete recibido, Pe : Paquete esperado

▷ Pe = 0, en primera iteración
if $Pe == Pi$ **then**
    Pe ++
    Enviar Paquete a capa Aplicación
    while Fin de buffer de paquetes recibidos **do** ▷ Buffer de paquetes que llegan en desorden
        if $Pe ==$ Número de paquete actual en buffer **then**
            Pe ++
            Enviar Paquete a capa Aplicación
            Contador de Paquetes recibidos por la capa APP ++
        else
            Romper Bucle ▷ Paquete faltante, debe ser retransmitido
        end if
    end while
else ▷ Paquete faltante, debe ser retransmitido
    Agregar Pi a buffer de paquetes recibidos
end if
▷ Crear y enviar *divacks*, ver Algoritmo 11
Nuevo ACK = ACK que pide Pe
$Planificar\_Servicio$ ($ACK\_DEPARTURE\_FROM\_TRANSPORT$, Nuevo ACK, de la capa Transporte, hacia la capa MAC del receptor)

emisor simplemente obviara el reconocimiento del divACK y pasará a crecer la ventana de congestión y enviar paquetes.

Algorithm 11 Implementación de envío de *divacks*

1: **for** i=0 to $nDivacks$ **do** ▷ Creando y enviando *divacks*
2: Nuevo ACK número -1
3: $Planificar\_Servicio$ ($ACK\_DEPARTURE\_FROM\_TRANSPORT$, Nuevo ACK, de la capa Transporte, hacia la capa MAC del receptor)
4: **end for**

4.4.4. Desde la capa MAC de receptor hacia enlace

Algorithm 12 Desde la capa MAC de receptor hacia enlace

1: $Planificar\_Servicio$ ($ACK\_DEPARTURE\_FROM\_NODE$, paquete asociado al evento, de capa MAC, hacia el Router)

4.4.5. Propagar ACK desde la capa MAC hacia el nodo Router

Algorithm 13 Propagar ACK desde la capa MAC hacia el nodo Router

1: Planificar propagación del ACK por el medio

4.4.6. Desde Router hacia el Enlace (de Regreso)

Algorithm 14 Desde Router hacia el Enlace

1: $Planificar\_Servicio$ ($ACK\_DEPARTURE\_FROM\_ROUTER$, paquete asociado al evento, de Router, hacia la capa MAC del emisor)

4.4.7. Propagar desde Router hacia la capa MAC de emisor

Algorithm 15 Propagar desde Router hacia la capa MAC de emisor

1: Planificar propagación del ACK hacia Capa MAC de emisor

4.4.8. Desde capa MAC del emisor hacia capa Transporte del emisor

Algorithm 16 Desde capa MAC del emisor hacia capa Transporte del emisor

1: *Planificar_Servicio* (*ACK_ARRIVAL_TO_TRANSPORT*, paquete asociado al evento, de capa MAC del emisor, hacia la capa Transporte del emisor)

4.5. Particularidades en la implementación de $\mu\mathcal{SIM}$

Existen consideraciones y detalles que llegaron a aparecer durante el desarrollo de la herramienta de simulación, que vale la pena exponer.

4.5.1. Implementando el Nodo Enrutador

En redes TCP/IP un enrutador se define como un ente intermedio que toma decisiones de camino óptimo para llevar un paquete de un punto a otro. Normalmente en sus interfaces de redes se le asocia dos o más redes diferentes, siendo el router, el distribuidor de la información. Pero también, no necesariamente de manera intencional, es un punto donde se va a crear el llamado cuello de botella, un fenómeno que causa congestión en la red. Es por eso que se decide traerlo a la simulación, como objeto fundamental.

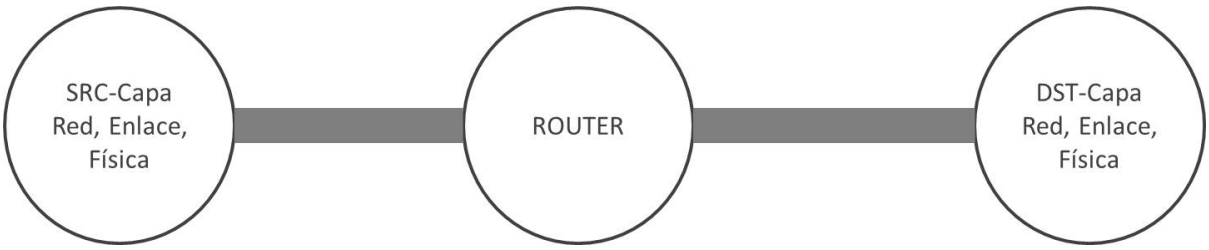

Figura 4.5: Nodo intermedio

El detalle que complica un poco su programación, es que, al ser un nodo intermedio, siempre existirá trafico que viajará en dos direcciones, trafico que debe

ser cuidadosamente manipulado solo en este punto, porque existen parámetros que afectan su transito a través del mismo. Para resolver este problema, se crearon colas que mantendrían un orden de los paquetes que viajaban desde y hacia ambas direcciones, y son atributos asociados a los nodos en general, esta colas se denominaran desde ahora, *forward queue* y *backward queue*.

4.5.2. Integración del Network Animator (NAM)

Como se ha mencionado, NAM es una herramienta muy útil y versátil, para la visualización del comportamiento de una red, tiene asociado como entrada un archivo de sintaxis y semántica definido para NAM, el cual, contiene toda la información estructural de la red y de todos los eventos que ocurren en un tiempo dado. Para integrar satisfactoriamente $\mu\mathcal{SIM}$ con NAM, fue necesario comprender el archivo con formato para NAM. Pero también lograr una manera de generar este archivo NAM de forma dinámica.

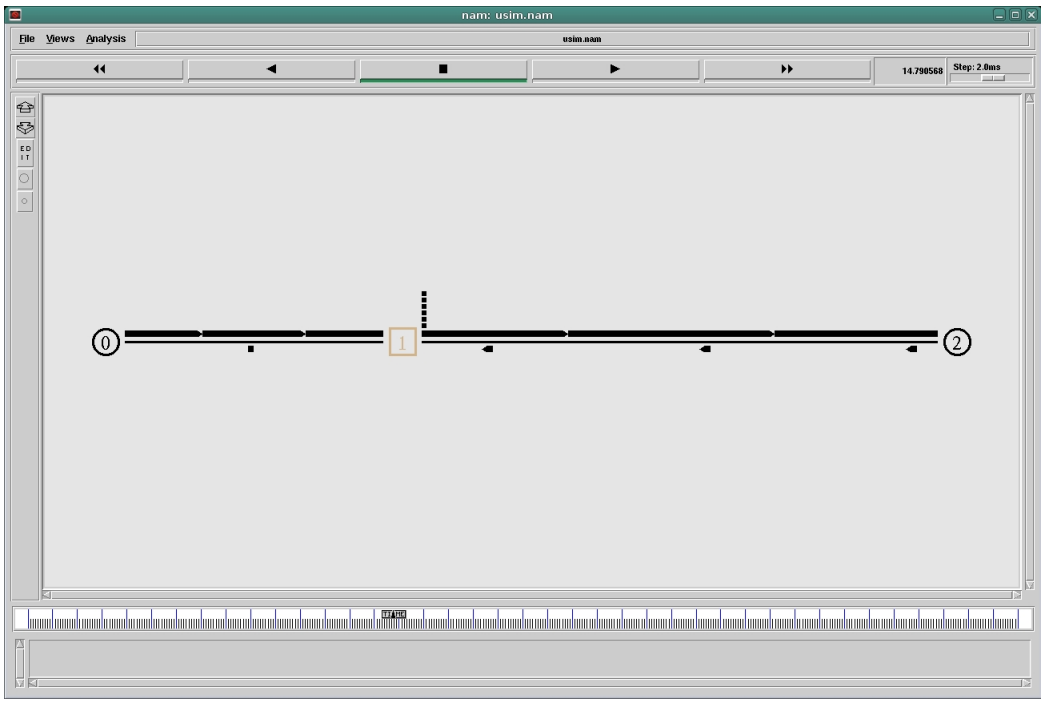

Figura 4.6: $\mu\mathcal{SIM}$ y NAM

Para esto, se requería una instancia global que exista durante toda la vida de la

simulación y pueda acceder a cualquier punto del simulador, y así poder recaudar la información necesaria para llenar completamente el archivo con formato NAM. Para implementar el generador de archivos NAM se utilizo el patrón de diseño *Singleton* para crear la instancia global.

Un ejemplo de archivo NAM se muestra a continuación:

```
V -t * -v 1.0a5 -a 0
n -t * -a 0 -s 0 -S UP -v circle -c black -i black
n -t * -a 1 -s 1 -S UP -v box -c tan -i tan
n -t * -a 2 -s 2 -S UP -v circle -c black -i black
l -t * -s 0 -d 1 -S UP -r 200000 -D 0.2500 -c black -o 0deg
l -t * -s 1 -d 2 -S UP -r 200000 -D 0.2500 -c black -o 0deg
q -t * -s 1 -d 0 -a 0.5
q -t * -s 0 -d 1 -a 0.5
q -t * -s 1 -d 2 -a 0.5
q -t * -s 2 -d 1 -a 0.5
+ -t 0.008000 -e 1000 -s 0 -d 1
- -t 0.008000 -e 1000 -s 0 -d 1
h -t 0.008000 -e 1000 -s 0 -d 1
r -t 0.298000 -e 1000 -s 0 -d 1
+ -t 0.298000 -e 1000 -s 1 -d 2
- -t 0.298000 -e 1000 -s 1 -d 2
h -t 0.298000 -e 1000 -s 1 -d 2
r -t 0.588000 -e 1000 -s 1 -d 2
h -t 0.590160 -e 54 -s 2 -d 1
r -t 0.840160 -e 54 -s 2 -d 1
h -t 0.842320 -e 54 -s 1 -d 0
r -t 1.092320 -e 54 -s 1 -d 0
+ -t 1.092320 -e 1000 -s 0 -d 1
- -t 1.092320 -e 1000 -s 0 -d 1
h -t 1.092320 -e 1000 -s 0 -d 1
+ -t 1.092320 -e 1000 -s 0 -d 1
```

```
- -t 1.132320 -e 1000 -s 0 -d 1
h -t 1.132320 -e 1000 -s 0 -d 1
```

Donde:

V: Es el inicio del archivo y la bandera -v indica que se usa la versión 1.0a5 de NAM.

n: Define cada uno de los nodos que existirán, los atributos para definir el nodo son: -t el tiempo en el que existe, -a dirección (por ejemplo dir IP), -s el id, -S el estado del nodo (UP o Down), -v la forma del nodo (circle, box, hexagon), -c el color, -i color secundario.

l: Define el enlace que conecta a los nodos, los atributos son: -t el tiempo en el que existe, -s el nodo origen, -d el nodo destino, -S el estado (UP o Down), -c el color, -o la orientación del enlace (x deg para orientación en grados), -r para ancho de banda, -D para delay o retraso

q: Define las colas asociadas a los nodos, los atributos son: -s el nodo origen, -d el nodo destino, -a la orientacion del dibujo de la cola en el animador (0.5 son 90 grados, es decir, se va a mostrar una cola vertical hacia arriba)

Los paquetes tienen asociados diferentes acciones en NAM:

h: Cuando el paquete inicia un salto desde un nodo origen. r: Cuando el paquete llega al nodo destino. +: Cuando el paquete entra a una cola. -: Cuando el paquete sale de una cola.

Los atributos de los paquetes son: -t es el tiempo en el que ocurre el evento, -s el nodo origen, -d el nodo destino, -e es el tamaño en bytes del paquete, -i (opcional) es el id del paquete, -P (opcional) es el tipo del paquete (TCP, ACK, NACK, SRM, etc.).

Para más información sobre el formato de entrada para NAM, revisar la documentación [2]

4.5.3. Orden en Scheduler

Definimos el Scheduler como el objeto que lleva una lista ordenada por tiempo, de eventos. El ordenamiento de los eventos se realiza siempre al insertar un nuevo evento a la lista, para ordenar solo se recorre la lista completa y se hace una consulta bastante

simple:

1: **if** Tiempo de evento nuevo < Tiempo evento iterado **then**
2: Insertar evento nuevo antes que evento iterado
3: **end if**

El problema existe cuando los valores de tiempos están dados por números en punto flotantes y más todavía cuando estos valores de tiempo son calculados por operaciones aritméticas básicas. Y es que el cálculo del tiempo de un evento, traerá acarreado (posiblemente), resultados de muchas operaciones aritméticas básicas, realizadas anteriormente, lo cual podría alterar la respuesta teórica de alguna operación.

Esto se convierte en un problema ocasionado, por la forma en la que un computador almacena los números flotantes en memoria, donde por ejemplo: 0.1 en realidad es:

```
0.1000000000000000000555111512312
```

Entonces podría llegar ocurrir algo curioso, supongamos que tenemos los siguientes valores:

```
a = 0.000000000000001000000
b = 0.000000000000000999999
```

Donde a puede ser el tiempo asociado al evento nuevo a insertar y b tiempo asociado a un evento en la lista de *Scheduler*, que por cálculos que traen un pequeño error acarreado, resulta aumentar un orden de magnitud al valor de a, y por ende, realizar una comparación no adecuada, ya que en realidad a debería ser igual a b.

Por esto no es recomendable realizar operaciones de comparación entre números flotantes sin antes tomar las precauciones convenientes. Para solucionar este problema se crea una pequeña rutina para comparar flotantes:

Algorithm 17 Rutina para comparar Flotantes

1: a = Tiempo de evento nuevo
2: b = Tiempo evento iterado
3: ϵ = Valor muy pequeño, se recomienda menores a 10^{-8}
4: **if** $|a - b| < \epsilon$ **then**
5: Son números iguales
6: **else**
7: Son números distintos
8: Comparar y retornar resultado de $(a < b)$
9: **end if**

4.5.4. Cálculo del Rendimiento de la red

Incluimos el concepto de rendimiento de la red basándonos en dos mediciones simples de capturar durante la simulación. Primero el *Throughput*, el cual es el flujo total de información que viaja por la red, se mide en bits por segundo, y se captura al momento de salir del nodo emisor de datos. La segunda medida es el *Goodput*, el cual es el total de información que llega de manera exitosa al destino, se mide en bits por segundo.

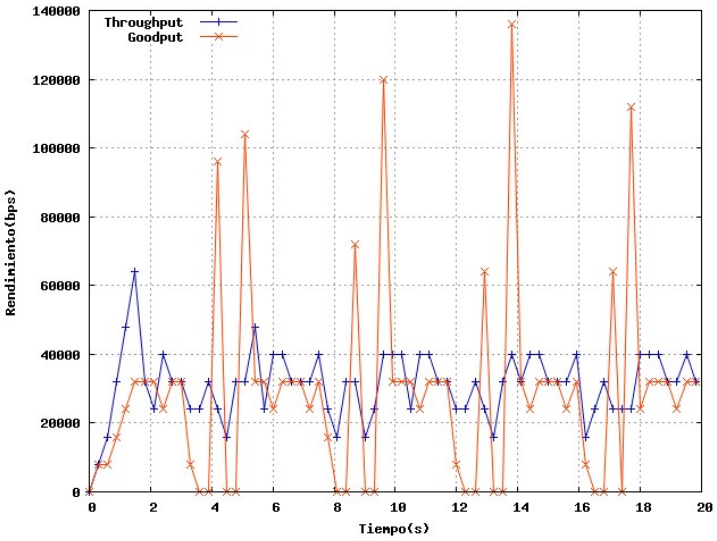

Figura 4.7: Medida del Rendimiento de una red

Observamos en la figura 4.7, que los puntos de cada gráfica están separados de

manera uniforme en el eje temporal, este valor se conoce como **delta**(δ), y es calculado antes de iniciar la simulación.

Para implementar la captura de la traza del rendimiento en $\mu\mathcal{SIM}$ se creó un nuevo evento llamado *trace*, el cual va a planificarse para que ocurra a los δ segundos para obtener los valores de *Throughput* y *Goodput* en ese instante, y luego planificar un evento *trace* nuevo que ocurrirá a los δ segundos siguientes.

4.5.5. Implementación del Temporizador

Recordemos que una manera de detectar pérdidas en una transmisión, es contando el tiempo que tarda en llegar un reconocimiento hasta que se cumpla un tiempo máximo, en este momento consideramos que existe un *Timeout* y se retransmite desde el ultimo paquete reconocido. Para implementar *Timeout* en $\mu\mathcal{SIM}$, debemos tener como entrada el valor de *timeout* en segundos. Existe un evento llamado *Timeout*, el cual va a ser planificado a ser ejecutado igual al valor actual de tiempo, más el valor de entrada definido para *timeout*. Creando así un **temporizador** que indicará el reenvío de los paquetes no verificados. este evento solo será tomado en cuenta si se detecta una posibilidad de pérdida de paquete o la transmisión se encuentra en recuperación de una pérdida.

En la figura 5.36 de la sección 5.4 se puede observar un ejemplo gráfico donde existen *Timeouts*.

4.5.6. Guía para extender $\mu\mathcal{SIM}$

En esta sección daremos pasos ordenados para extender $\mu\mathcal{SIM}$, específicamente para añadir una nueva capa de modelo OSI al simulador.

Flujo de trabajo para extender $\mu\mathcal{SIM}$:

1. Crear un nuevo tipo de nodo que sera usado por la clase *Node*. Por ejemplo si queremos implementar la capa red de OSI, debemos crear el nodo tipo Red.

2. Extender en la definición de la clase *Event*, todos los eventos asociados al nuevo nodo (hay que tener en cuenta los nodos conectados directamente al nuevo nodo). Por ejemplo:

   a) Los eventos tipo *arrival*.

   b) Los eventos tipo *departure*.

   c) Los eventos de propagación (puesta en medio si los nodos están separados por un medio físico de transmisión. Por ejemplo un cable o el aire)

   d) Los eventos de tipo temporizador.

3. Instanciar el nuevo nodo correspondiente a la capa que se está desarrollando, y enlazar con las capas adyacentes.

4. Manejo de los eventos dentro del ciclo principal de ejecución del simulador. Recordar que existen eventos que tienen asociados un nodo FROM y un nodo TO (solo si no es el caso de los eventos temporizadores).

   a) (Opcional) Implementar manipulación del PDU cuando entra o sale del nuevo nodo, por ejemplo si queremos implementar la capa red, se debería asociar al PDU una dirección IP destino y una dirección IP origen.

Un ejemplo detallado que sigue la guía para extender $\mu\mathcal{SIM}$ se puede observar en el apéndice A.

4.6. La Interfaz Gráfica de Usuarios

Adicionalmente, se ha desarrollado un entorno gráfico para manipular algunas variables de $\mu\mathcal{SIM}$ y poner observar su comportamiento en vivo. Esta ha sido orientada principalmente para observar como se comporta TCP en $\mu\mathcal{SIM}$. además tiene varias funcionalidades adicionales, como guardar sesiones y generar un archivo con formato NAM con la traza completa de la simulación para correrla en NAM

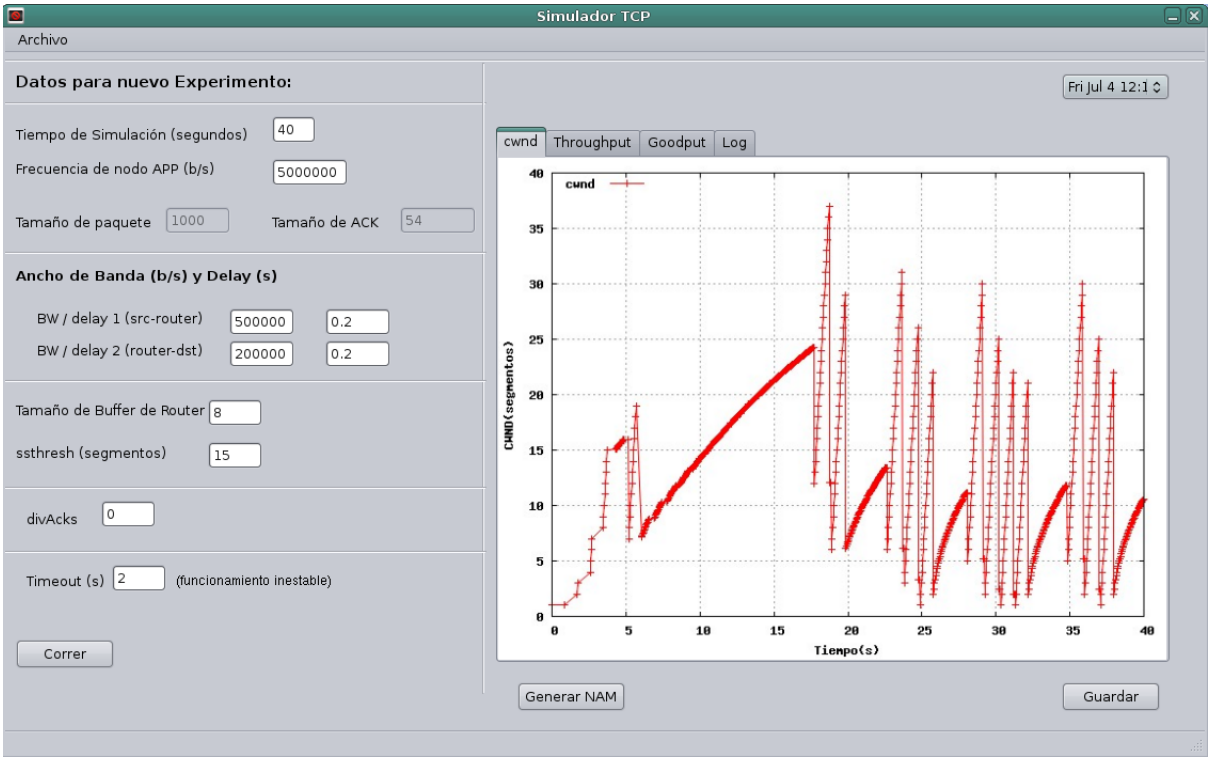

Figura 4.8: Interfaz Gráfica de $\mu\mathcal{SIM}$

4.6.1. Panel de Variables

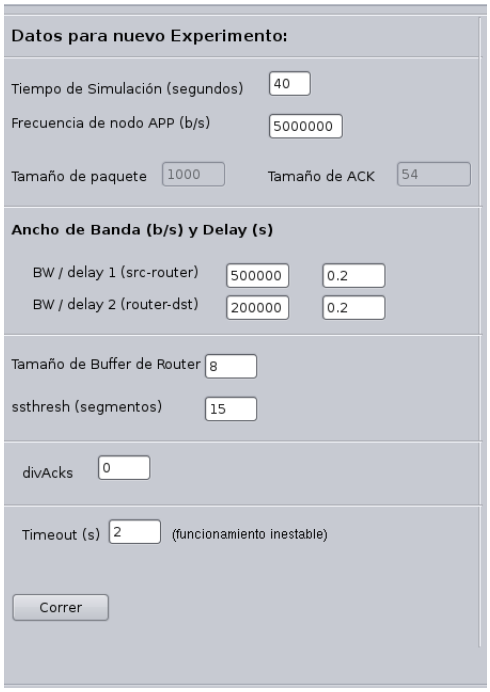

Figura 4.9: Panel de variables

En 4.10 se observa la distribución de los controles donde se puede editar cada uno de los parámetros de la simulación, está ubicado a la izquierda de la ventana principal, y desde ahí, en la parte inferior está el botón donde se corre la simulación.

4.6.2. Panel de Resultados

Está ubicado a la derecha de la ventana principal y es la sección donde se observará todas las gráficas que corresponden a la simulación que ha terminado de correr, el recuadro central contiene 4 tabs que separan cada una de las respuestas, *cwnd*, *throughput*, *goodput* y el log. En la parte inferior hay dos botones, a la izquierda el botón que genera el archivo NAM, el cual, al hacer clic saldrá una ventana emergente que pregunta por la ruta donde va a ser guardada la traza NAM. El segundo botón en la parte inferior derecha, es el botón para guardar el experimento, al presionar se genera un archivo XML con toda la información necesaria para recuperar el experimento,

siempre que se guarde un experimento nuevo, se actualizará la caja de selección de experimento ubicada en la parte superior derecha. Al precionar en un experimento, el simulador recuperará la información del archivo XML y se podrá visualizar en el cuadro central de la interfaz.

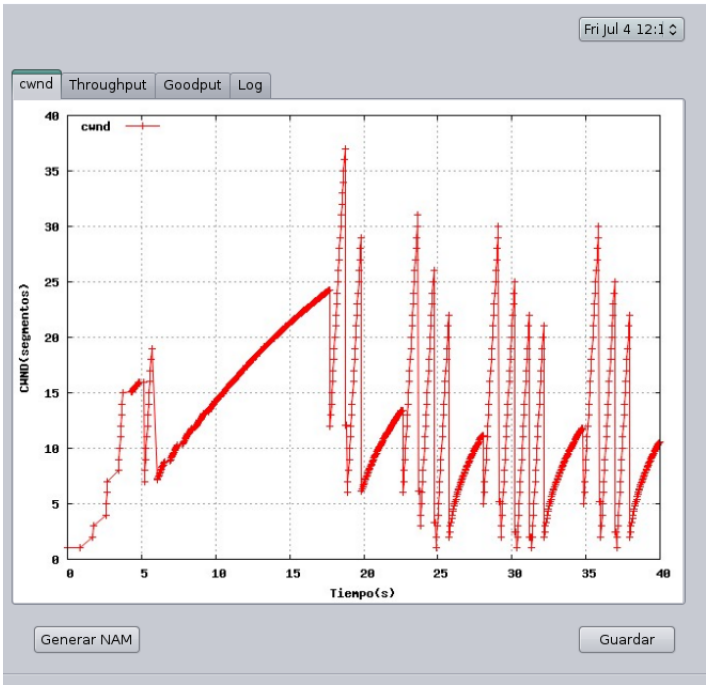

Figura 4.10: Sección de resultados

Para cambiar la ubicación donde se guarda el archivo XML, hay que ir a la opción en la parte superior izquierda de la ventana principal Archivo ⇒ Abrir, aparecerá una ventana emergente donde permitirá elegir el nombre y ubicación de la sesión, si en cambio se elige la ubicación donde hay un archivo de sesión antiguo, la interfaz añadirá todos los experimentos a la caja de selección de experimento y permitirá visualizar cada uno de ellos.

Capítulo 5

Pruebas de $\mu\mathcal{SIM}$

Se realizara una serie de pruebas a $\mu\mathcal{SIM}$ para observar su funcionamiento en diversas configuraciones y características de red, con el objeto de estudiar su respuesta durante las distintas fases implementadas: respuesta a cambio de fases, respuesta a pérdidas de paquetes y la implementación de *divacks* y como afecta al sistema. Las dinámicas a observar serán: el crecimiento de *cwnd* y *ssthreshold* y el rendimiento de la red o *Throughput* y el *Goodput*.

Recordemos que la configuración de red establecida es Nodo emisor $<->$ Router $<->$ Nodo receptor, como se observa en la figura 5.1

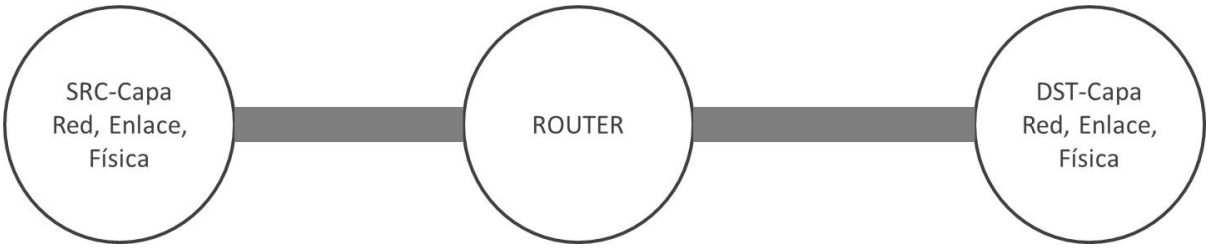

Figura 5.1: Configuración de red

Por cada prueba se mostrará gráficas correspondientes a *cwnd* y *ssthreshold*, rendimiento de la red o *Throughput* y *Goodput*. Recordamos que existen 6 variables que afectaran el resultado de la simulación.

Ancho de Banda 1	bits por segundo entre emisor y Router full-duplex
Ancho de Banda 2	bits por segundo entre Router y receptor full-duplex
One way Delay 1	segundos, retardo entre emisor y Router
One way Delay 2	segundos, retardo entre Router y receptor
Tamaño de cola	segmentos, cola de salida de Router
SSThreshold	segmentos, Umbral de Slow Start

5.1. Slow Start

Se irá cambiando gradualmente los valores de Ancho de Banda y delay por cada prueba realizada.

Prueba 1

Ancho de Banda	3 Mb/s
One way Delay	400 ms
Tiempo de simulación	6 s
Delta de medición	400 ms

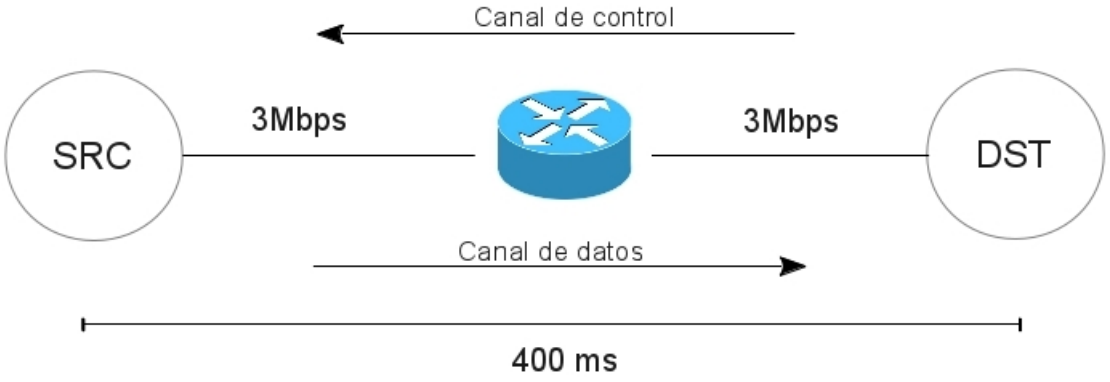

Figura 5.2: Modelo de la red, Prueba 1 de Slow Start

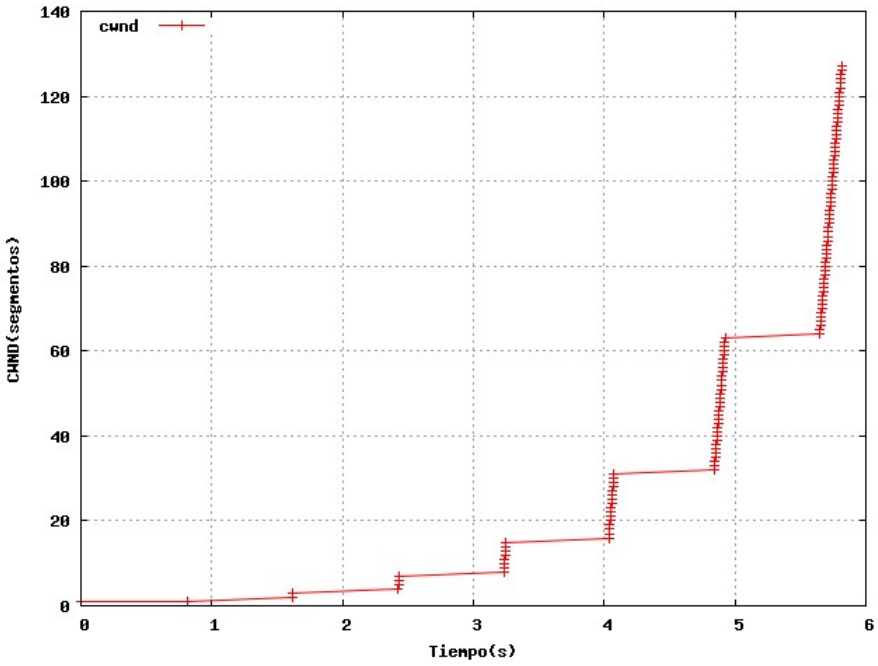

Figura 5.3: *cwnd*, Prueba 1 de Slow Start

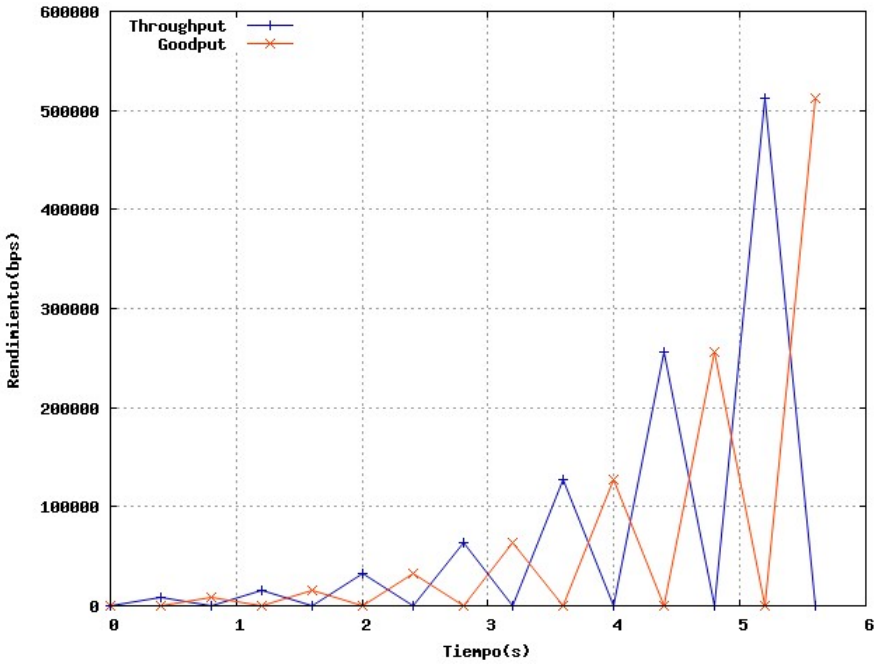

Figura 5.4: Rendimiento, Prueba 1 de Slow Start

Prueba 1	
Cantidad neta de paquetes recibidos Nodo final	127
Cantidad de paquetes recibidos por capa Aplicación	127
Cantidad de paquetes perdidos	0
Cantidad de paquetes reenviados	0
Cantidad de Timeouts	0

Prueba 2

Ancho de Banda	3 Mbps
One way Delay	2 s
Tiempo de simulación	30 s
Delta de medición	2 ms

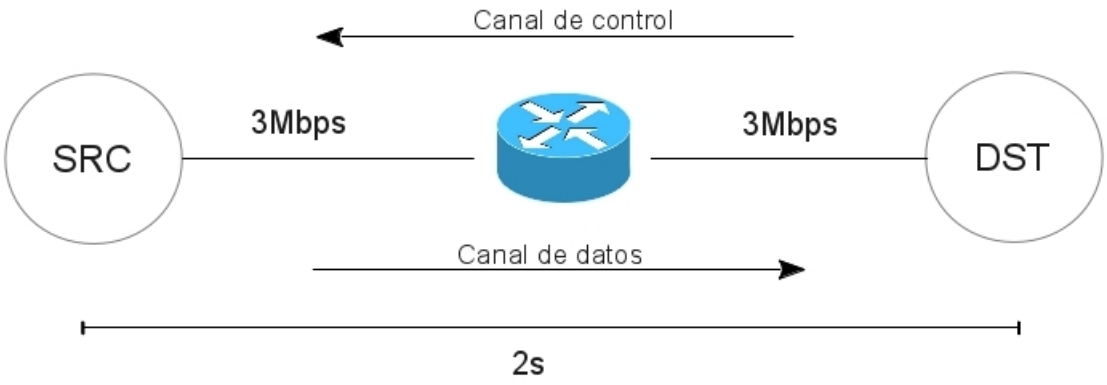

Figura 5.5: Modelo de la red, Prueba 2 de Slow Start

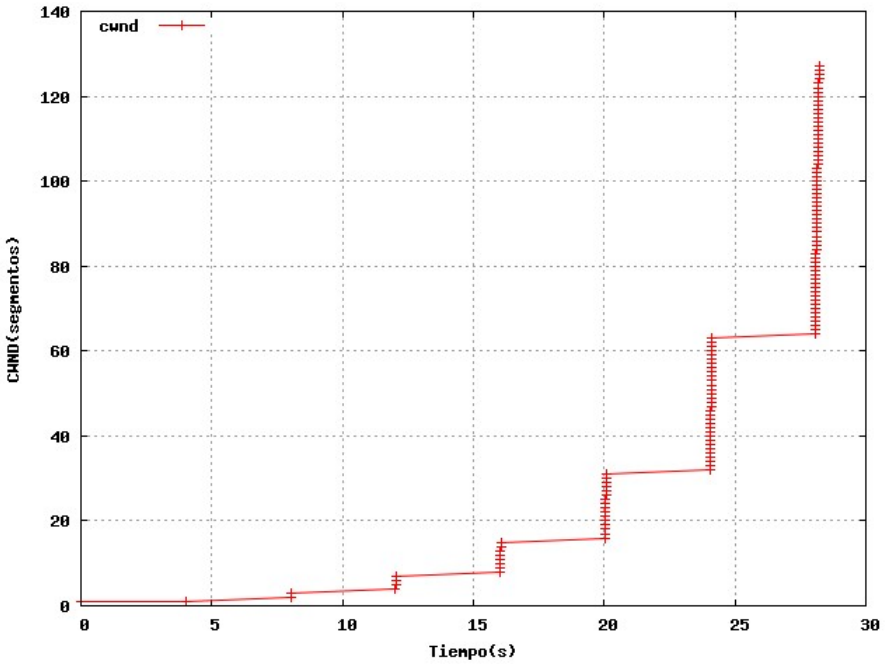

Figura 5.6: *cwnd*, Prueba 2 de Slow Start

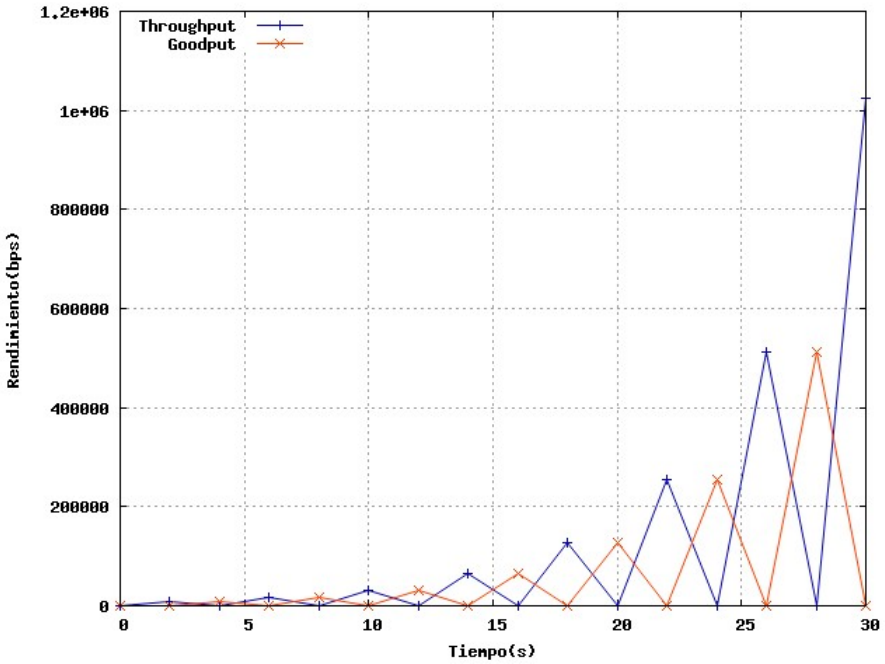

Figura 5.7: Rendimiento, Prueba 2 de Slow Start

Prueba 2	
Cantidad neta de paquetes recibidos Nodo final	127
Cantidad de paquetes recibidos por capa Aplicación	127
Cantidad de paquetes perdidos	0
Cantidad de paquetes reenviados	0
Cantidad de Timeouts	0

Para las pruebas 1 y 2 de esta sección observamos el comportamiento que existe en el simulador durante la fase Slow Start de una transmisión. Para las gráficas de crecimiento de ventana de congestión (figuras 5.3 y 5.6) confirmamos que hay un comportamiento correcto correspondiente a los resultados supuestos teóricos. En las gráficas de rendimiento (figuras 5.4 y 5.7) observamos que la respuesta es adecuada y corresponde con el crecimiento de la ventana de congestión. Como se habla en la sección 4.5.4, el Throughput es medidos desde el emisor de datos, y el Goodput es medido desde el receptor, por eso se ve un desfase en ambas gráficas, ya que existe un tiempo de propagación para un paquete termine de llegar a capa aplicación del nodo receptor. Ademas observamos que la gráfica de Goodput es un espejo del Throughput ya que al no existir perdidas de paquetes, entonces todo paquete que sale del emisor llega ordenadamente al receptor.

Hay que acotar que en las las gráficas de rendimiento existen puntos donde su valor es igual a cero, esto ocurre porque en ese instante no existe flujo en dicho extremo, es decir que si el valor de Goodput en un instante de tiempo es cero, significa que en el nodo receptor no existe flujo de datos. Si el valor de Throughput es igual a cero en un instante, significa que en el nodo emisor no existe flujo de datos.

5.2. Congestion Avoidance

Se irá cambiando gradualmente los valores de Ancho de Banda, delay y *ssthresh* por cada prueba realizada.

Prueba 1

Ancho de Banda	300 Kbps
One way Delay	400 ms
ssthresh	5 segmentos
Tiempo de simulación	10 s
Delta de medición	400 ms

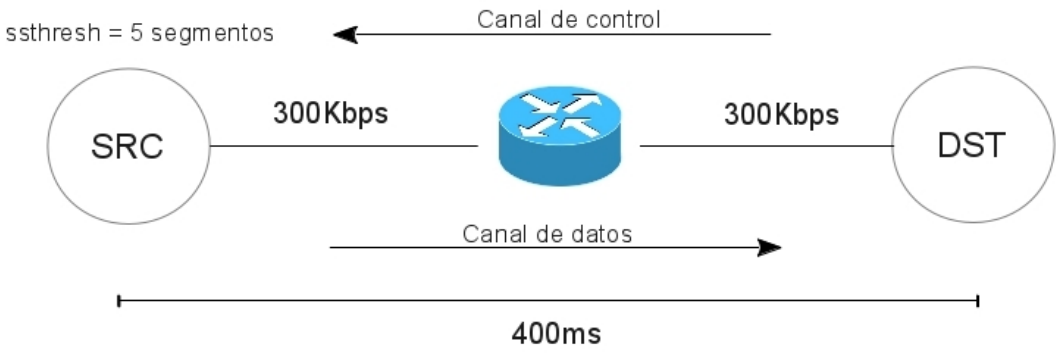

Figura 5.8: Modelo de la red, Prueba 1 de Congestion Avoidance

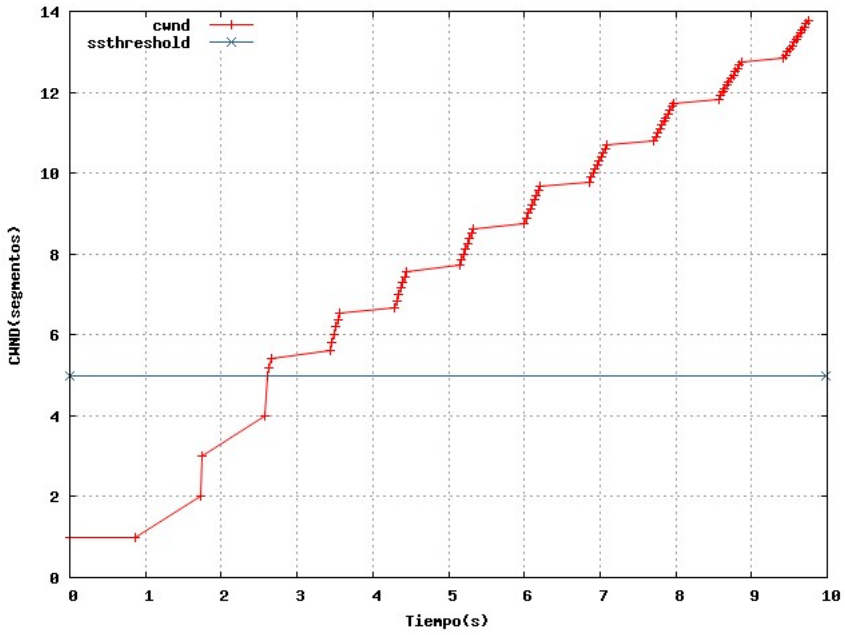

Figura 5.9: *cwnd*, Prueba 1 de Congestion Avoidance

En la figura 5.9 cuando el tamaño de la ventana de congestión supera el valor del umbral de Slow Start se puede observar el cambio a Congestion Avoidance.

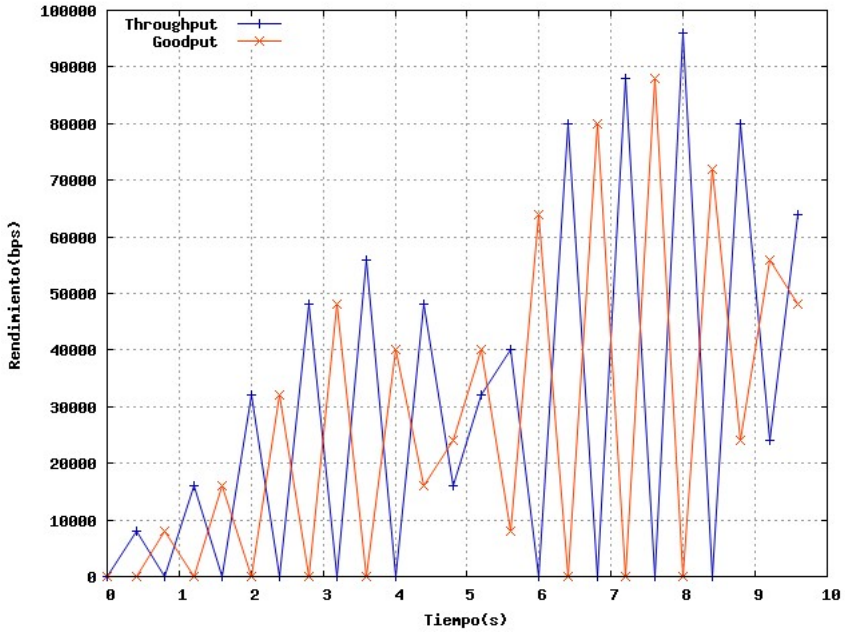

Figura 5.10: Rendimiento, Prueba 1 de Congestion Avoidance

Cantidad neta de paquetes recibidos Nodo final	88
Cantidad de paquetes recibidos por capa Aplicación	88
Cantidad de paquetes perdidos	0
Cantidad de paquetes reenviados	0
Cantidad de Timeouts	0

Prueba 2, corrida a largo plazo

Ancho de Banda	300 Kbps
One way Delay	400 ms
ssthresh	5 segmentos
Tiempo de simulación	35 s
Delta de medición	400 ms

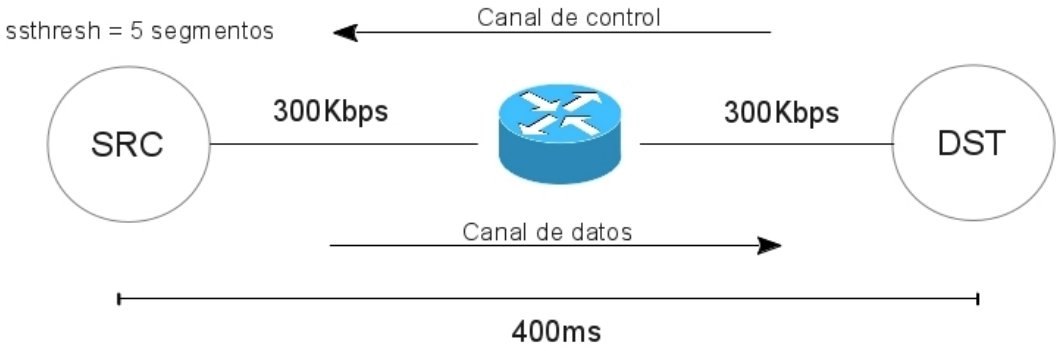

Figura 5.11: Modelo de la red, Prueba 2 de Congestion Avoidance

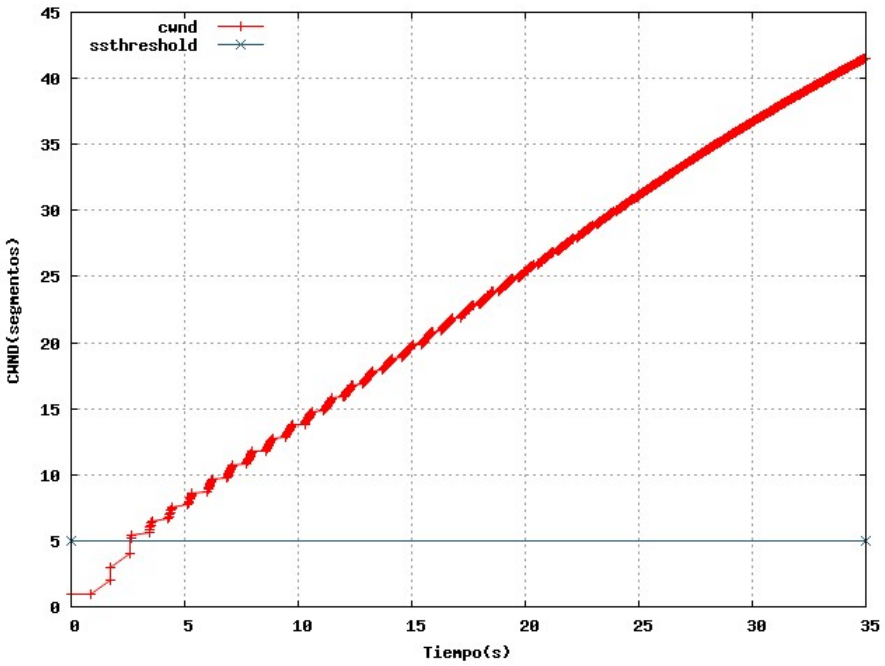

Figura 5.12: *cwnd*, Prueba 2 de Congestion Avoidance

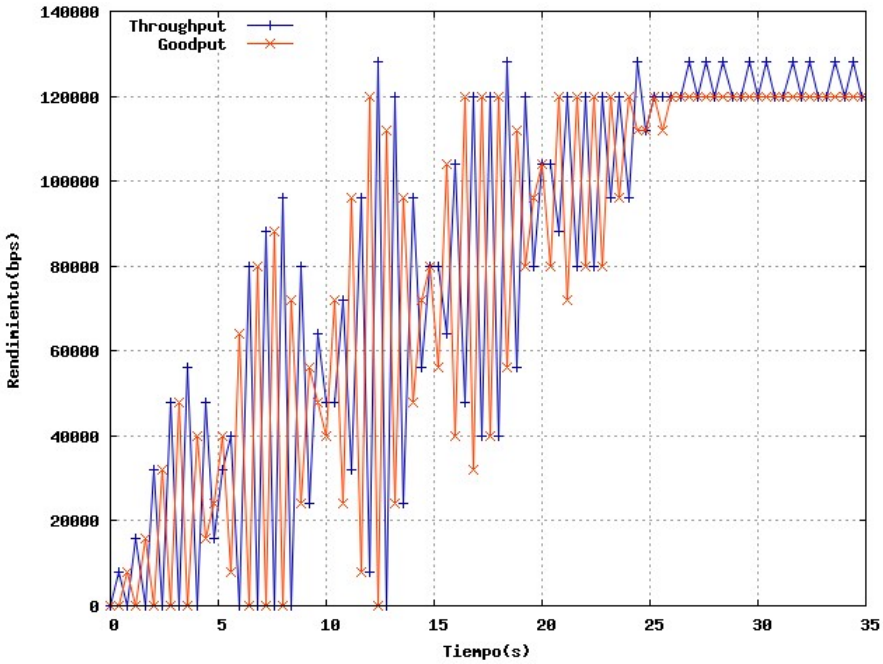

Figura 5.13: Rendimiento, Prueba 2 de Congestion Avoidance

Cantidad neta de paquetes recibidos Nodo final	1255
Cantidad de paquetes recibidos por capa Aplicación	1255
Cantidad de paquetes perdidos	0
Cantidad de paquetes reenviados	0
Cantidad de Timeouts	0

La prueba 2 de Congestion Avoidance se realiza para mostrar el comportamiento de una transmisión sin perdidas que ocurre en un tiempo prolongado, observamos en las imágenes 5.12 y 5.13 aproximadamente en el punto T = 25 segundos, se estabiliza la transmisión alcanzando el valor máximo de datos que viajaran por el sistema.

5.3. Pérdida de Paquetes y Fast Recovery

En las siguientes pruebas, es necesario forzar un cuello de botella, para lograr esto, el ancho de banda entre SRC y Router debe ser mayor al de Router y DST, además, hay que agregar la variable tamaño de buffer de Router para que exista desborde de paquetes.

Se cambiará gradualmente los valores de Ancho de Banda 1(SRC-Router), Ancho de Banda(Router-DST), One way Delay 1 , One way Delay 2, *ssthresh* y tamaño de buffer, por cada prueba realizada.

Prueba 1

Ancho de Banda 1	200 Kbps
Ancho de Banda 2	100 Kbps
One way Delay 1	100 ms
One way Delay 2	200 ms
ssthresh	10 segmentos
buffer	8 segmentos
Tiempo de simulación	40 s
Delta de medición	300 ms

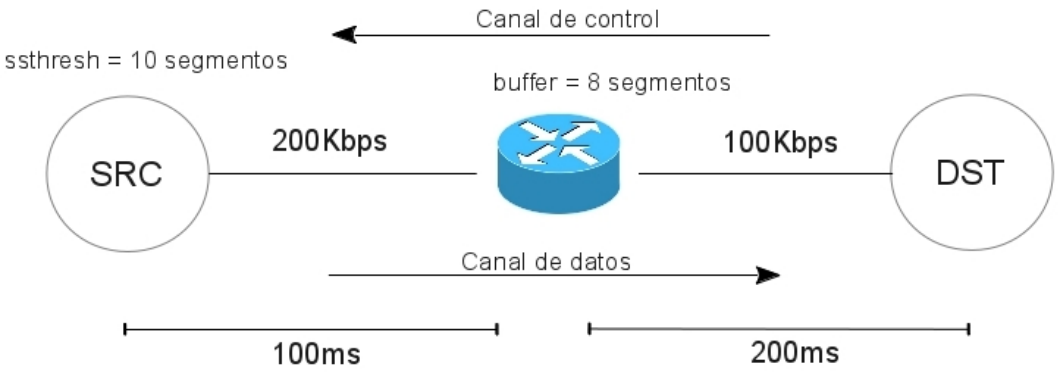

Figura 5.14: Modelo de la red, Prueba 1 Fast Recovery

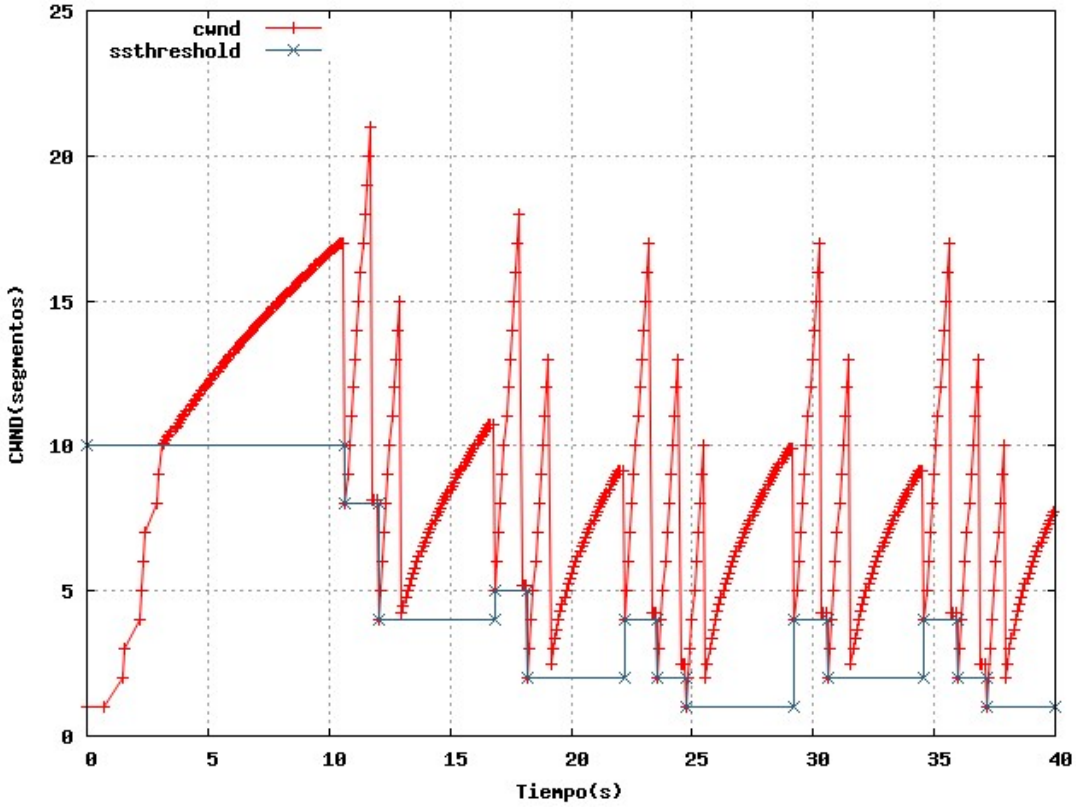

Figura 5.15: *cwnd*, Prueba 1 Fast Recovery

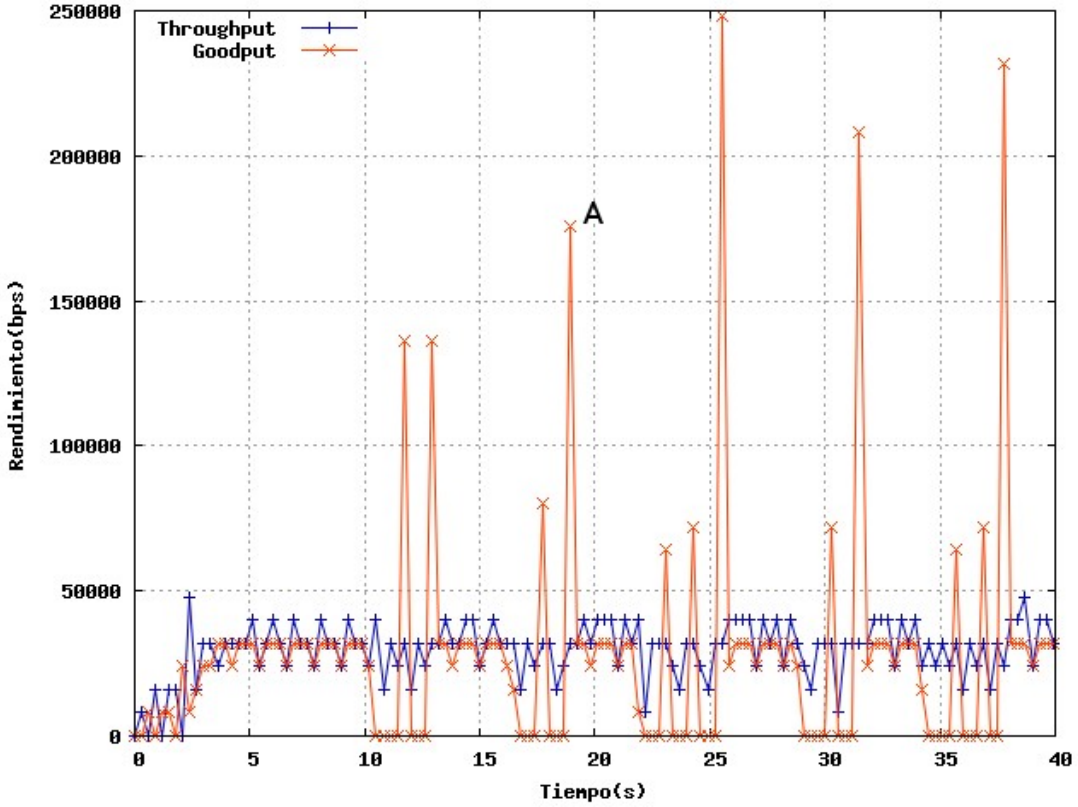

Figura 5.16: Rendimiento, Prueba 1 Fast Recovery

Cantidad neta de paquetes recibidos Nodo final	474
Cantidad de paquetes recibidos por capa Aplicación	474
Cantidad de paquetes perdidos	12
Cantidad de paquetes reenviados	12
Cantidad de Timeouts	0

Podemos observar en la figura 5.16 una serie de picos en la gráfica del *Goodput* (por ejemplo el punto *A* en dicha figura), estos picos corresponden al punto en el que se vacía el buffer de paquetes fuera de orden que llegan al nodo receptor, es causado por la llegada del paquete esperado por el nodo receptor.

Prueba 2

Ancho de Banda 1	500 Kbps
Ancho de Banda 2	400 Kbps
One way Delay 1	300 ms
One way Delay 2	300 ms
ssthresh	50 segmentos
buffer	10 segmentos
Tiempo de simulación	50 s
Delta de medición	600 ms

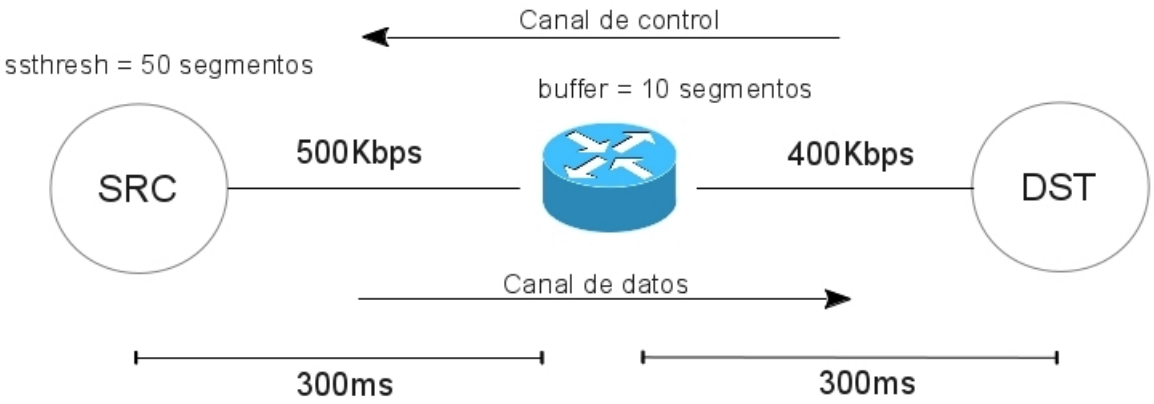

Figura 5.17: Modelo de la red, Prueba 2 Fast Recovery

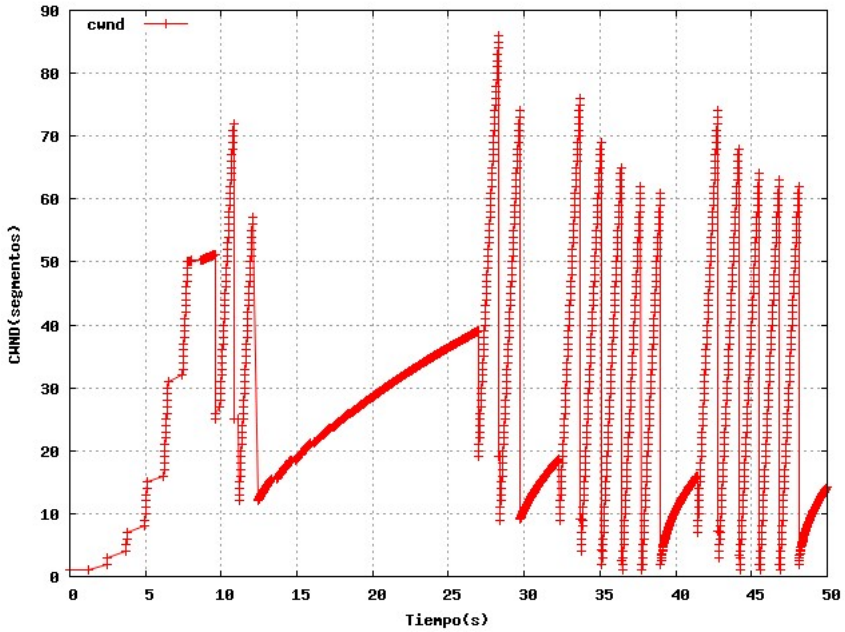

Figura 5.18: *cwnd*, Prueba 2 Fast Recovery

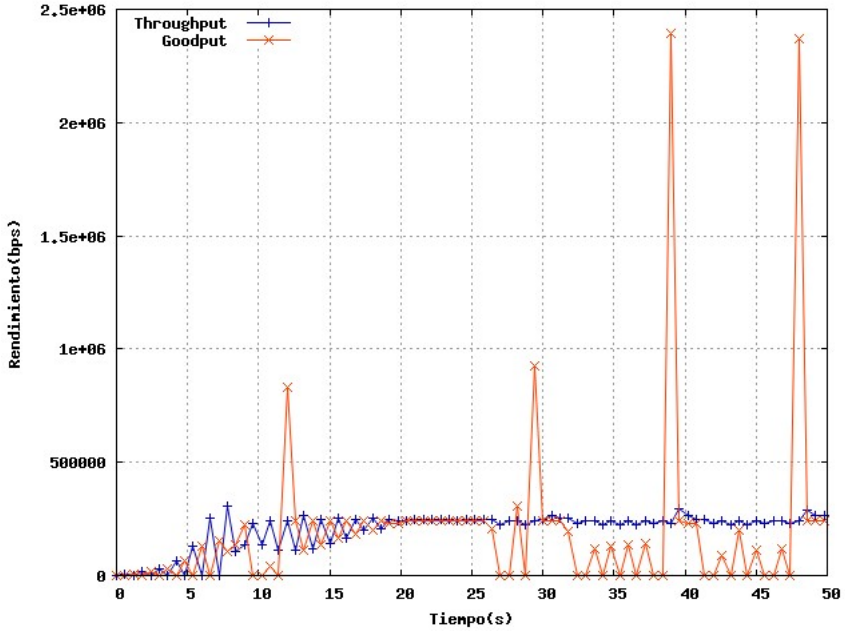

Figura 5.19: Rendimiento, Prueba 2 Fast Recovery

Cantidad neta de paquetes recibidos Nodo final	2063
Cantidad de paquetes recibidos por capa Aplicación	2058
Cantidad de paquetes perdidos	17
Cantidad de paquetes reenviados	14
Cantidad de Timeouts	0

En la prueba 2 realizamos un experimento similar al mostrado en prueba 1 de Fast Recovery, para efectos de comparación. Observamos que existen los mismos fenómenos explicados en la prueba 1. Ademas debemos recordar que mientras el Throughput puede estabilizarse en un punto, el Goodput va a depender de la cantidad de perdidas de paquetes que exista.

Prueba 3

En la prueba 3 comparamos la respuesta gráfica al cambio del delta (δ) para la medición del rendimiento de la red, en las pruebas pasadas el cálculo del delta fue siempre igual a la suma entre One way Delay 1 y One Way Delay 2.

Ancho de Banda 1	200000 b/s
Ancho de Banda 2	100000 b/s
One Way Delay 1	0.05 s
One Way Delay 2	0.05 s
ssthresh	10 segmentos
buffer	8 segmentos
Tiempo de simulación	20 s

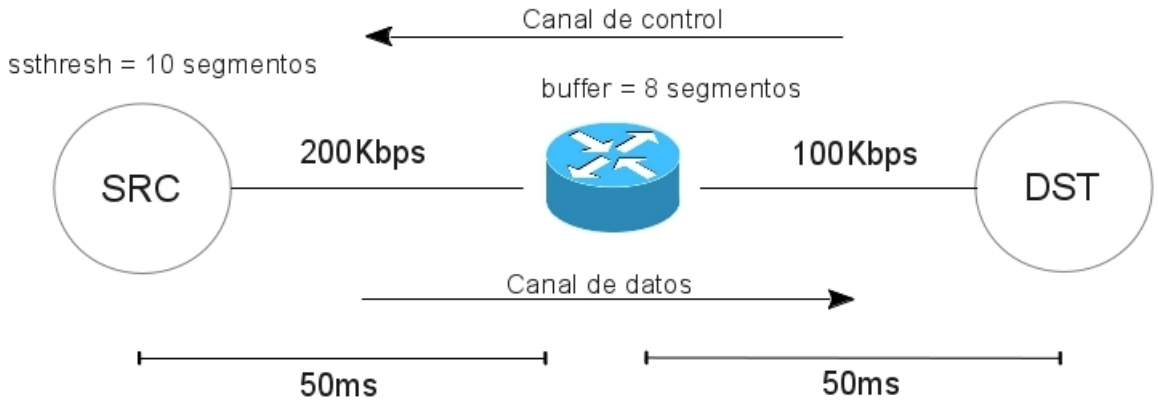

Figura 5.20: Modelo de la red, Prueba 3 Fast Recovery

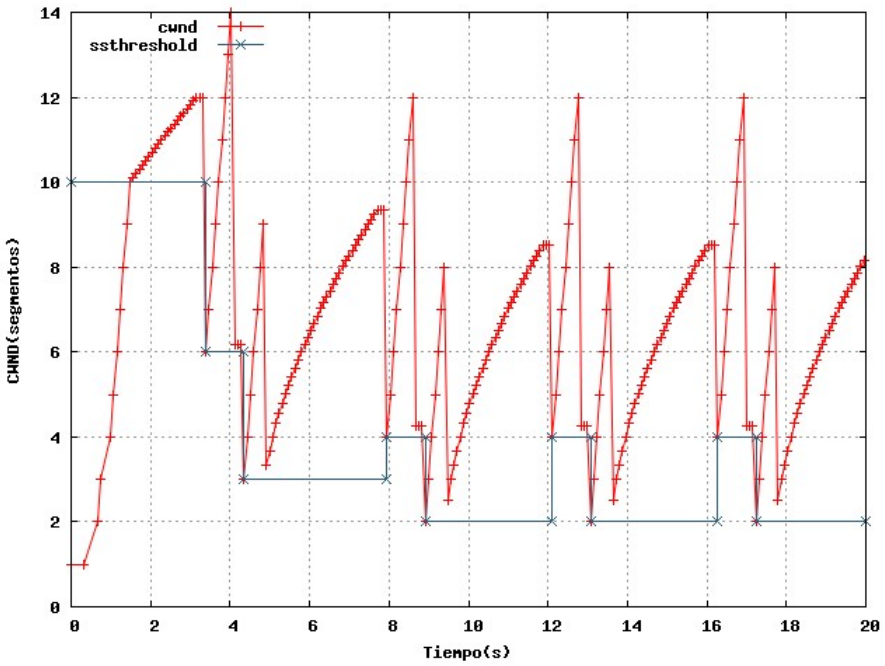

Figura 5.21: *cwnd*, Prueba 3 Fast Recovery

Para $\delta = 100$ms

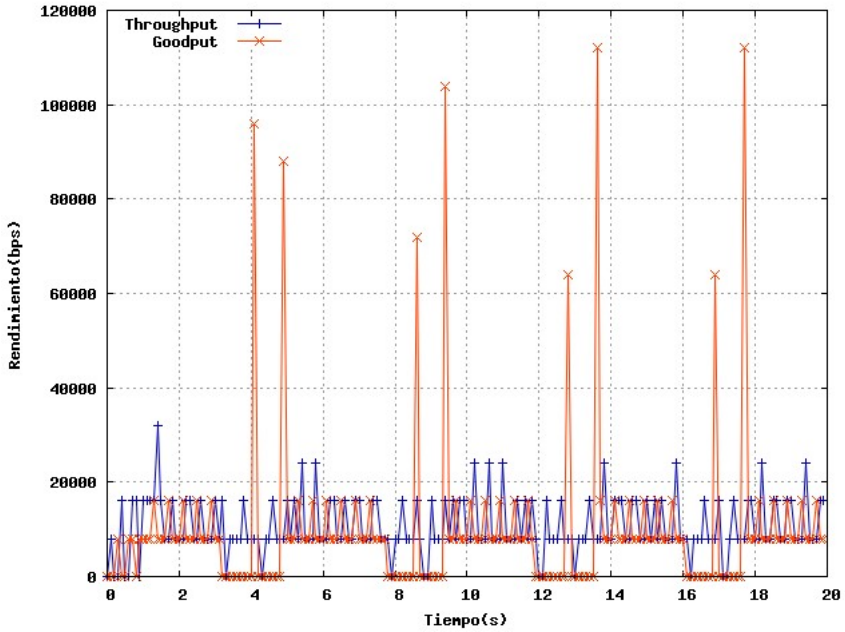

Figura 5.22: Rendimiento 1, Prueba 3 Fast Recovery

Para $\delta = 300$ms

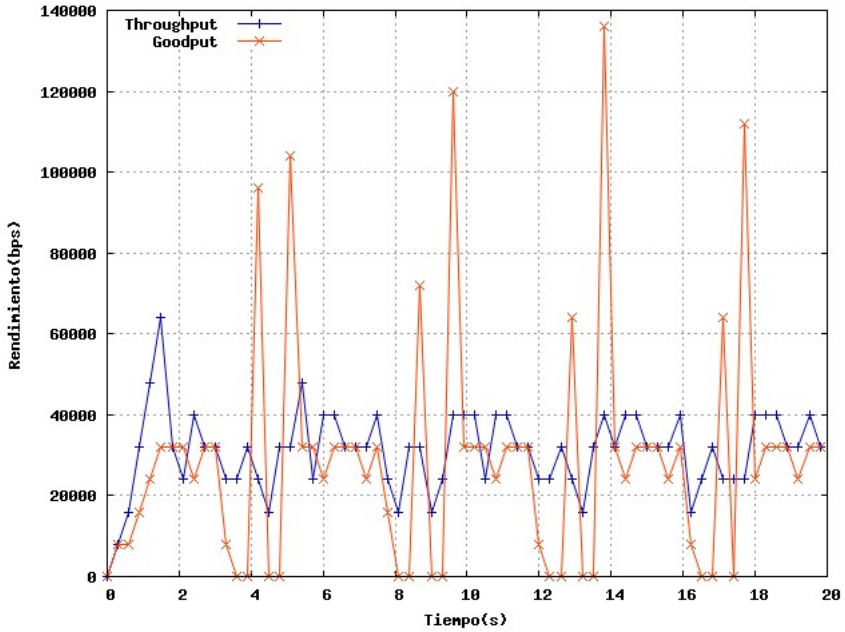

Figura 5.23: Rendimiento 2, Prueba 3 Fast Recovery

Para $\delta = 500$ms

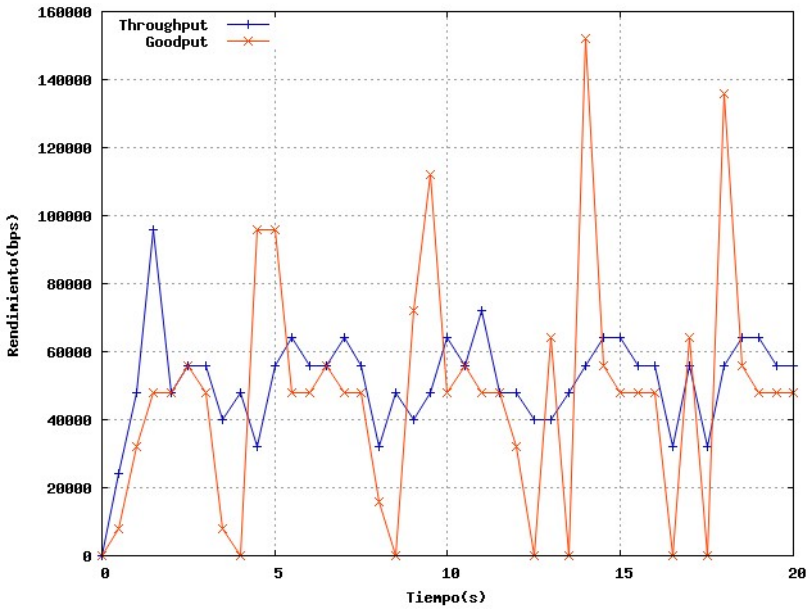

Figura 5.24: Rendimiento 3, Prueba 3 Fast Recovery

Cantidad neta de paquetes recibidos Nodo final	242
Cantidad de paquetes recibidos por capa Aplicación	242
Cantidad de paquetes perdidos	17
Cantidad de paquetes reenviados	14
Cantidad de Timeouts	0

Hay que aclarar que el valor de δ no va a afectar la respuesta del sistema, si no la frecuencia en la que se va a mostrar la información correspondiente al Throughput y el Goodput, en la sección 4.5.4 se aclara un poco más sobre δ.

5.4. Aceleración de la Transmisión

Se debe realizar pruebas a los *divacks* en escenarios con congestión y sin congestión.

Prueba 1, comparando Slow Start

Ancho de Banda 1	200 Kbps
Ancho de Banda 2	200 Kbps
One Way Delay 1	500 ms
One Way Delay 2	500 ms
Tiempo de simulación	10 s

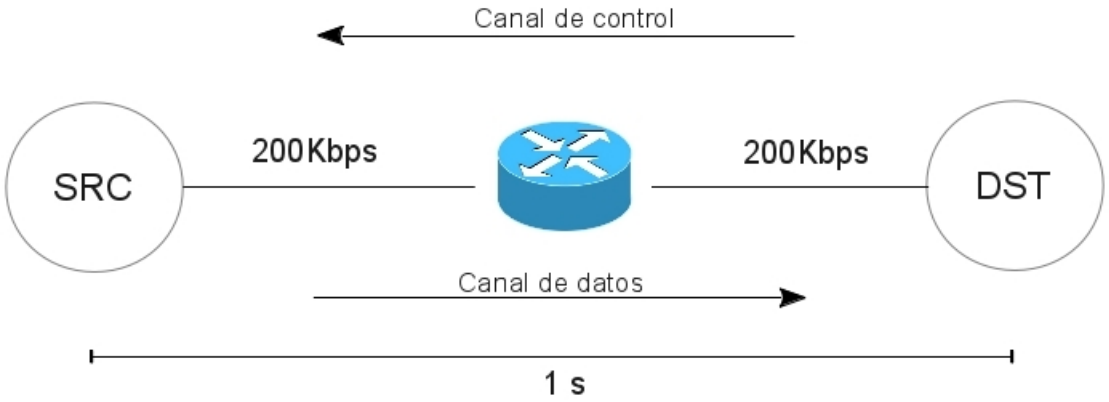

Figura 5.25: Modelo de la red, Prueba 1 de *divacks*

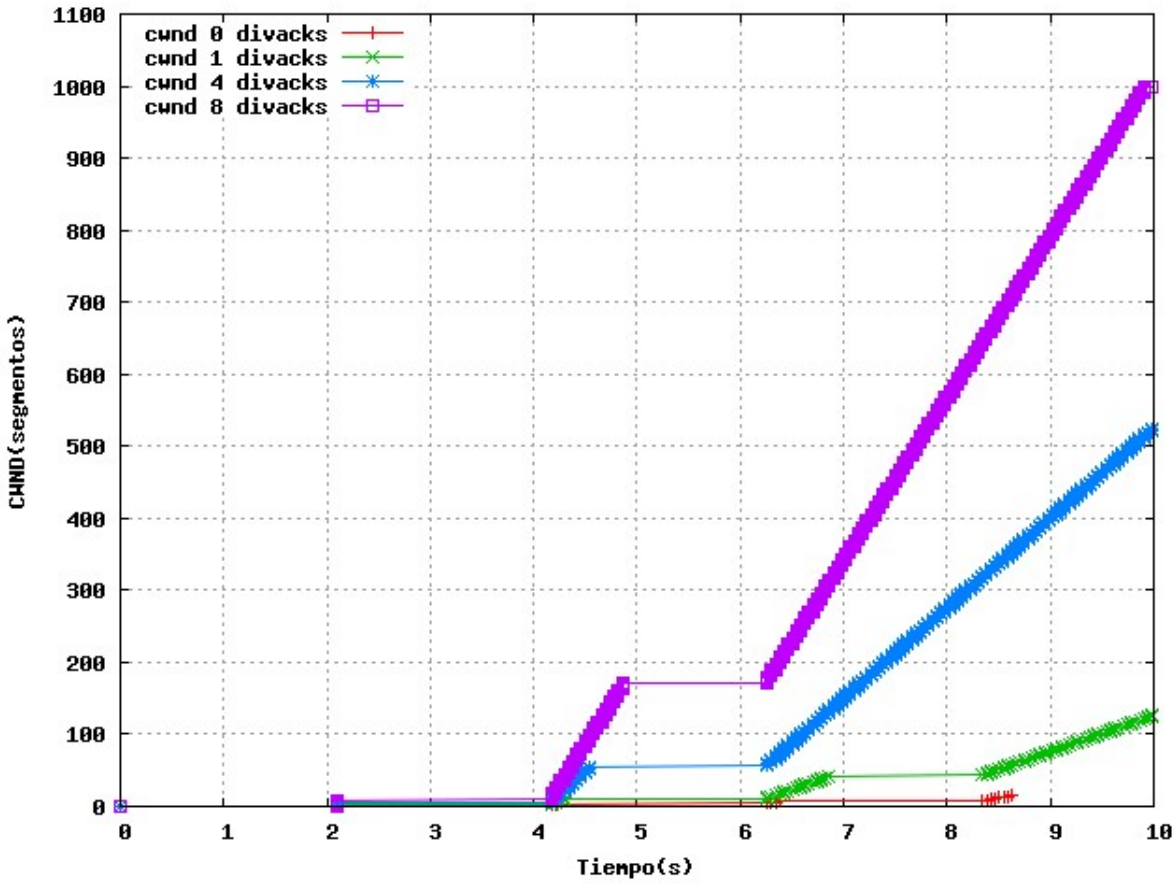

Figura 5.26: *cwnd*, Prueba 1 de *divacks*

Se nota el incremento en la aceleración de *cwnd*

divacks	0	1	4	8
Cantidad de paquetes recibidos Nodo final	30	88	130	138
Cantidad de paquetes recibidos por Aplicación	30	88	130	138
Cantidad de paquetes perdidos	0	0	0	0
Cantidad de paquetes reenviados	0	0	0	0
Cantidad de Timeouts	0	0	0	0

Prueba 2, comparando Congestion Avoidance

Ancho de Banda 1	2 Mbps
Ancho de Banda 2	2 Mbps
One Way Delay 1	200 ms
One Way Delay 2	200 ms
Tiempo de simulación	5 s
ssthresh	1 segmento

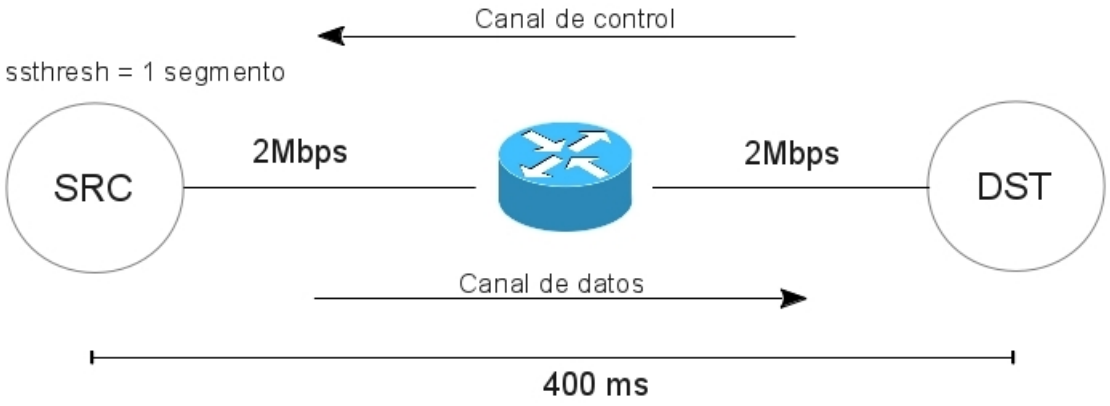

Figura 5.27: Modelo de la red, Prueba 2 de *divacks*

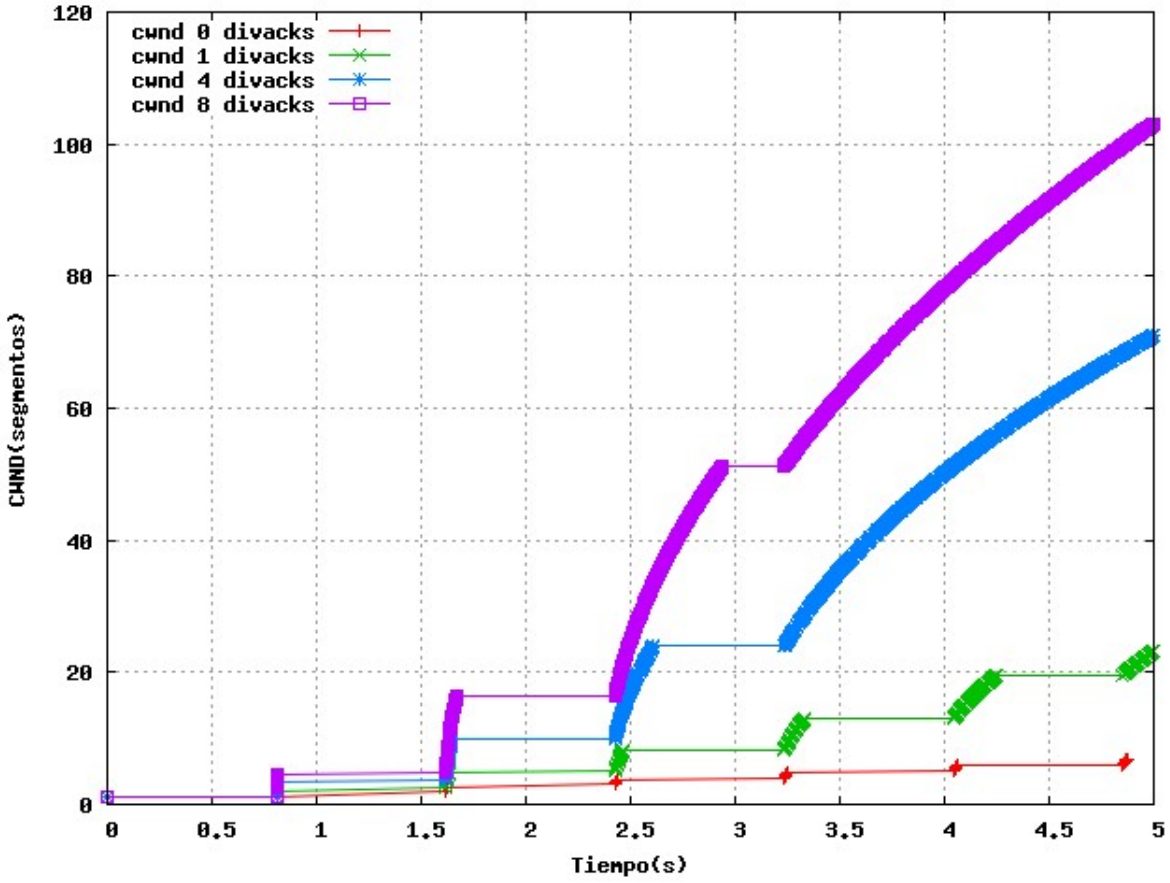

Figura 5.28: *cwnd*, Prueba 2 de *divacks*

El incremento en la aceleración del tamaño de la *cwnd* se sigue notando fácilmente estando en la fase Congestion Avoidance.

divacks	0	1	4	8
Cantidad de paquetes recibidos Nodo final	21	202	597	685
Cantidad de paquetes recibidos por Aplicación	21	202	597	685
Cantidad de paquetes perdidos	0	0	0	0
Cantidad de paquetes reenviados	0	0	0	0
Cantidad de Timeouts	0	0	0	0

Prueba 3, probando divacks en una red congestionada

Ancho de Banda 1	500 Kbps
Ancho de Banda 2	200 Kbps
One Way Delay 1	200 ms
One Way Delay 2	200 ms
ssthresh	15 segmentos
buffer	8 segmentos
Tiempo de simulación	40 s
Delta de medición	400 ms

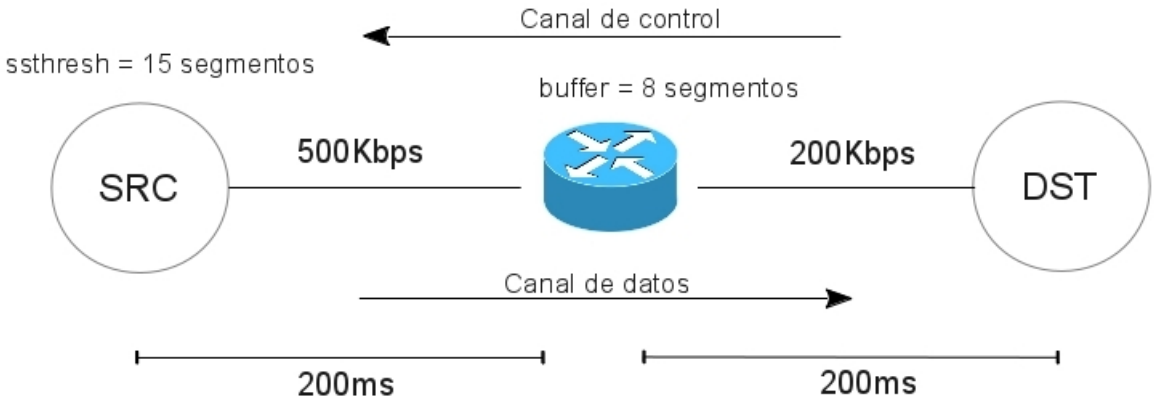

Figura 5.29: Modelo de la red, Prueba 3 de *divacks*

Prueba con 0 *divacks*

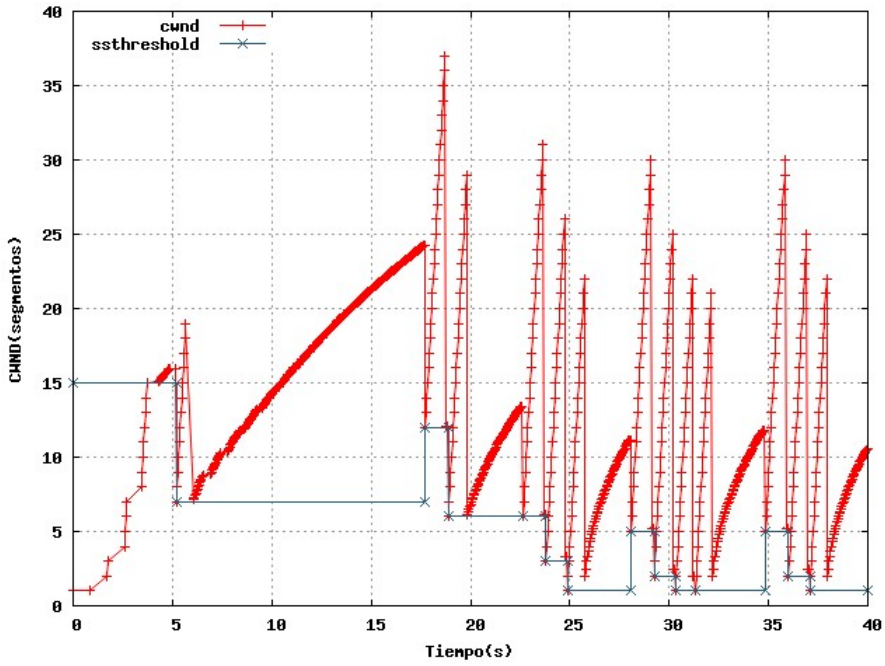

Figura 5.30: *cwnd*, Prueba 3, 0 *divacks*

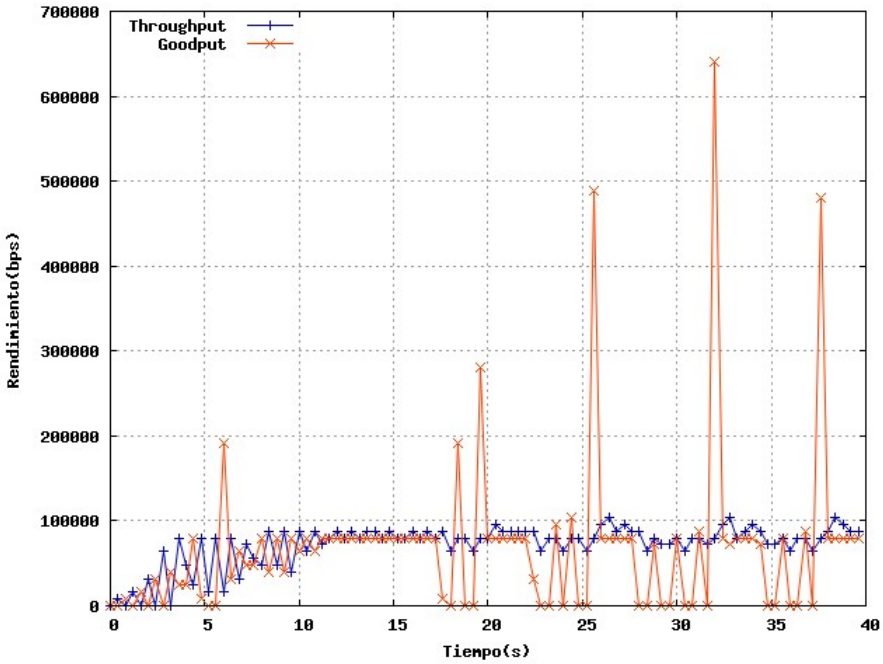

Figura 5.31: Rendimiento, Prueba 3, 0 *divacks*

Prueba con 1 *divack*

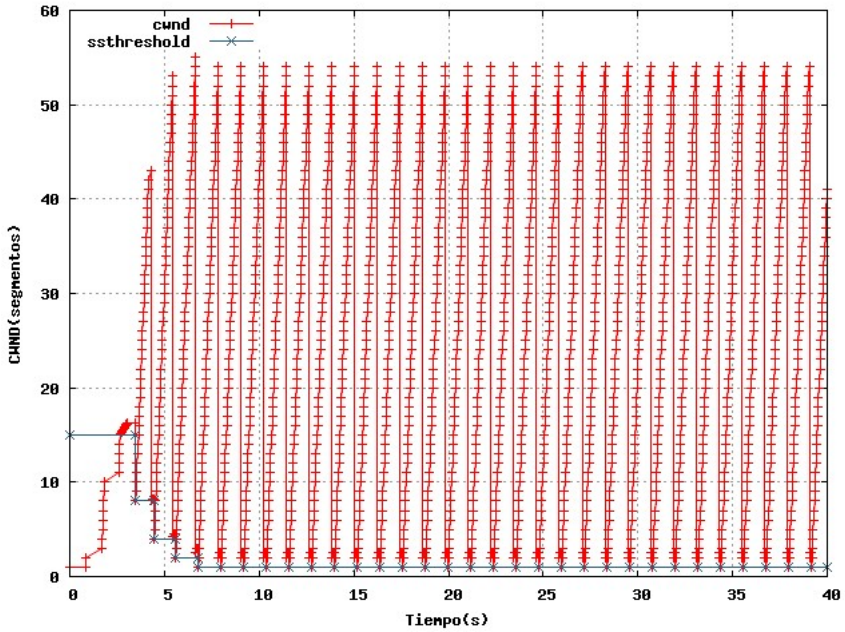

Figura 5.32: *cwnd*, Prueba 3, 1 *divack*

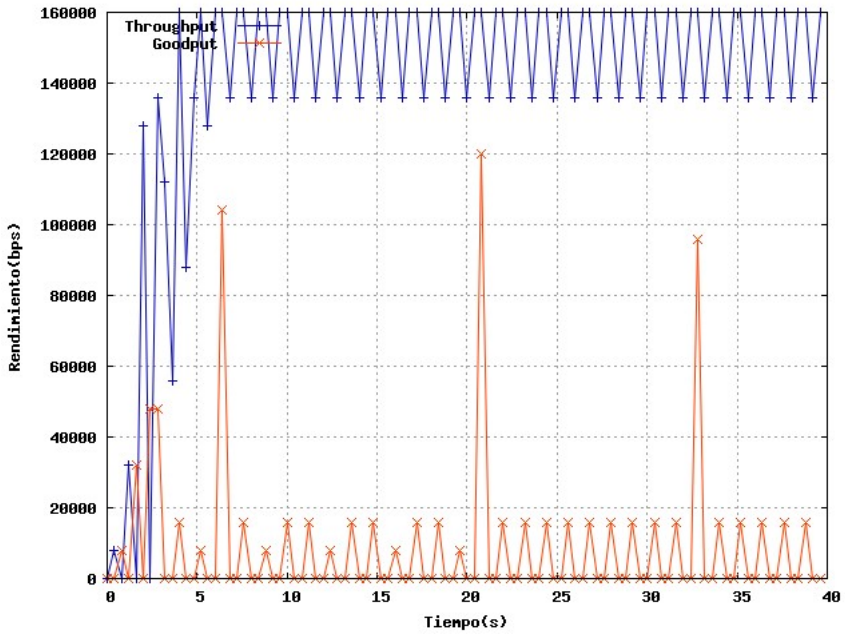

Figura 5.33: Rendimiento, Prueba 3, 1 *divack*

Se puede ver en la figura 5.32 que existe múltiples pérdidas de paquetes en la transmisión, y en la figura 5.33 se observa que la cantidad de paquetes recibidos por capa Aplicación (*Goodput*) es menor que la cantidad de paquetes enviados (*Throughput*).

Prueba con 2 *divacks*:

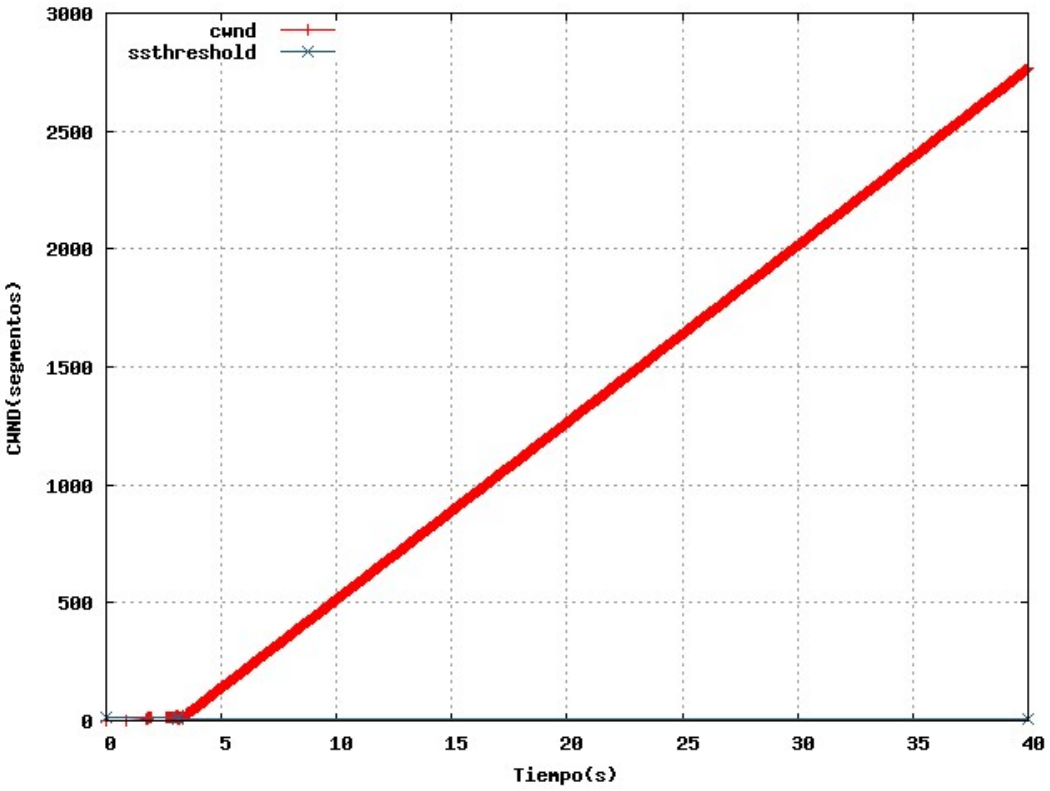

Figura 5.34: *cwnd*, Prueba 3, 2 *divacks* sin timeout

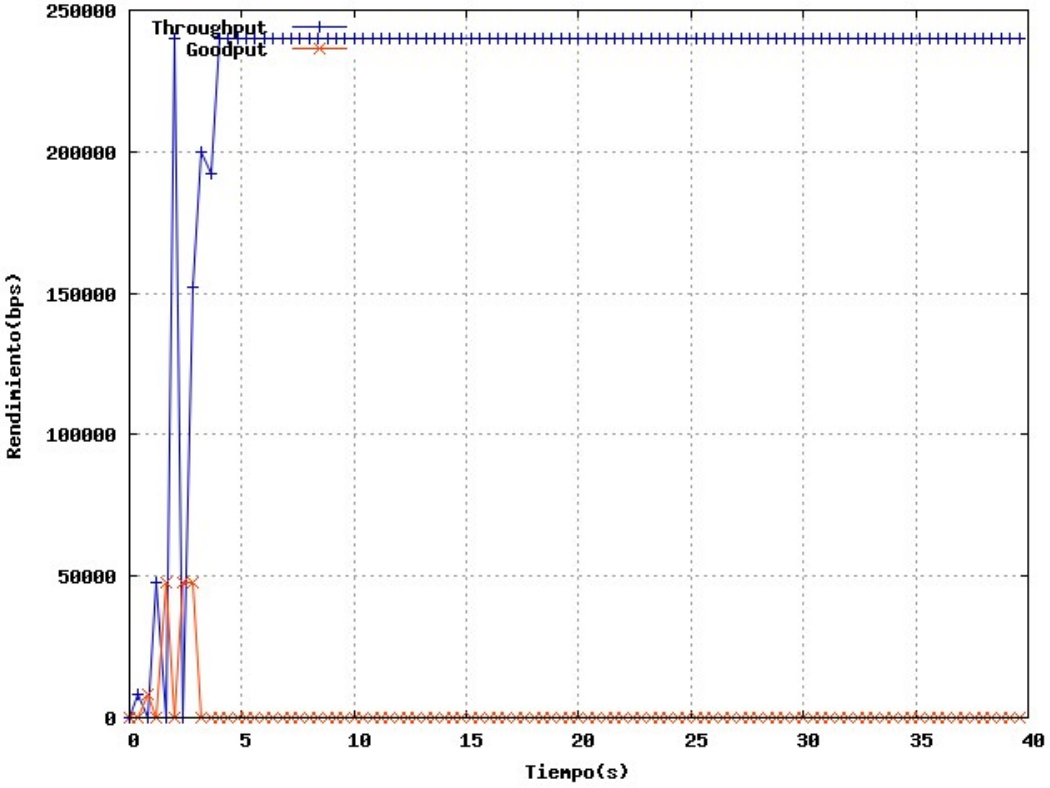

Figura 5.35: Rendimiento, Prueba 3, 2 *divacks* sin timeout

Observamos en la figura 5.35, que mientras existe una cantidad grande de datos generados por el nodo emisor (Throughput), son pocos los paquetes que llegan de manera ordenada a la capa Aplicación (Goodput).

Se puede inferir que la respuesta que se muestra en la figura 5.34 se obtiene, porque existe una estadía indefinida en la fase *Fast Recovery*, esto ocurre porque se pierde el paquete retransmitido esta fase (*Fast Recovery*). Para dar una mejor respuesta al comportamiento de la transmisión debemos usar un valor de ***timeout*** para poder salir de la estadía indefinida de *Fast Recovery*.

Prueba con 2 *divacks* y *timeout* igual a 2 segundos:

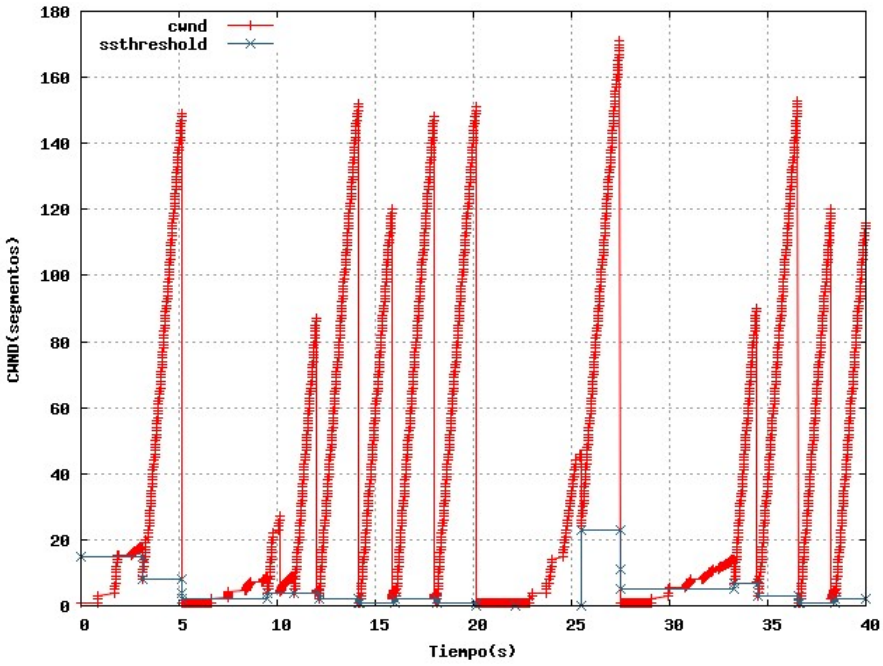

Figura 5.36: *cwnd*, Prueba 3, 2 *divacks*

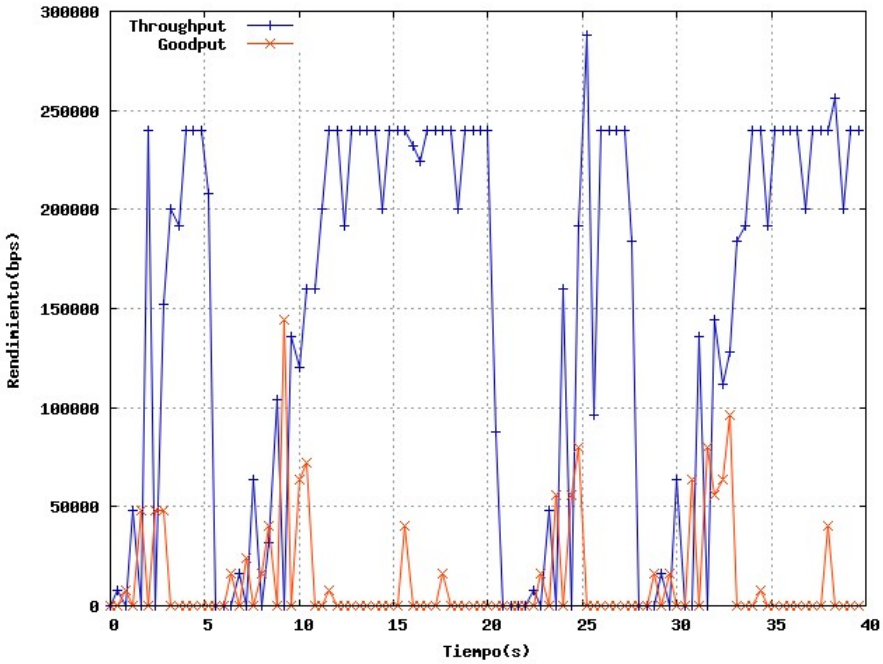

Figura 5.37: Rendimiento, Prueba 3, 2 *divacks*

En tiempo aproximadamente igual a 5 segundos de las figuras 5.36 y 5.37 se puede observar la ocurrencia de un ***timeout***, existe una retransmisión completa desde el último paquete reconocido.

divacks	0 divacks	1 divacks	2 divacks
Cantidad neta de paquetes recibidos Nodo final	872	939	791
Cantidad de paquetes recibidos por capa Aplicación	867	108	154
Cantidad de paquetes perdidos	16	816	939
Cantidad de paquetes reenviados	13	31	21
Cantidad de Timeouts	0	0	9

Se puede observar que la cantidad de paquetes generados por los divacks es tan grande, que tiene la capacidad de inundar fácilmente una red predispuesta a congestionarse, los resultados de los divacks son totalmente negativos, según la respuesta que da $\mu\mathcal{SIM}$ para estos casos. Vemos que la capacidad de recuperación de pérdidas es cada ves más difícil a medida que se aumenta la frecuencia de envío de *divacks*. En el ultimo caso donde *divacks* es igual a 2, visualizamos que ocurren 9 timeouts, considerando que no existe ningún tipo de paralización o pérdida de ACKs, el único motivo que se puede inferir para explicar este fenómeno, es que después de reenviar un paquete perdido, el mismo paquete se pierde, y entra en un ciclo de espera dentro de la fase *Fast Recovery*, el cual, la única manera de salir es declarándose en *timeout* y reiniciando la red.

Capítulo 6

Conclusiones y Recomendaciones

En este proyecto se estudió el impacto de la variación de los divacks en el tiempo total de transferencia para archivos cortos. Para ello se desarrolló un modelo matemático, y simulador para probar distintas estrategias asociadas a la variación abrupta de divacks. Con el modelo matemático se logra establecer un estudio inicial sobre el impacto que tiene modificar la cantidad de divAcks, en el tiempo que tarda en concluir una transferencia de archivo de tamaño dado.

En el estado actual en el que se encuentra $\mu\mathcal{SIM}$, podemos indicar que hemos cumplido con tres aspectos fundamentales que definen a $\mu\mathcal{SIM}$. Primero, logramos diseñar, implementar y probar un simulador a eventos discretos con un patrón definido basado en la salida de eventos de un nodo y llegada a la cola del nodo siguiente. Estos nos permitió generalizar el procedimiento tanto para la comunicación vertical (intra-nodo) como para la comunicación horizontal (inter-nodo). Este modelo permite hacer énfasis en los diferentes costes durante el desarrollo por etapas de un simulador de entes comunicantes a eventos discretos.

Segundo, basado en el modelo y técnica antes expuestos, implementamos TCP para observar en la red el efecto del control de congestión. Además, se puede observar el efecto de la organización OSI por capas. Los algoritmos de TCP están bien localizados en el modelo propuesto en la capa transporte. Este hecho ayuda a comprender y ejemplificar la orientación por capas. Recalcando así la eficiencia en la organización de la solución.

Por último se añadió una modificación TCP para acelerar las transmisiones basado en la técnica de *divacks*, estudiando su impacto en la velocidad de transferencia. Aunque en la practica esta añadidura tiene complejas implicaciones en la conducta de TCP, pudimos implementarla siguiendo los principios de organización de código aquí expuestos.

En las pruebas expuestas en el capítulo 5, podemos destacar que los resultados obtenidos al introducir divACKs al sistema muestran una respuesta positiva cuando el sistema no tiene perdidas de datos, pero cuando el sistema tiende a congestionarse el funcionamiento resulta ser contrario a lo deseado. Entonces podemos decir que el uso de divACKs pudiera ser destinado a la transferencia de archivos cortos y en redes poco congestionadas para evitar la perdida de datos. Pero también se puede inferir que el uso de los divACKs puede existir en redes con congestión, pero que su duración debe existir por un corto periodo de tiempo, es decir que la transmisión va a ser corta. Comprendiendo por transmisión de datos corta como toda aquella que solo ocurrirá durante la fase Slow Start de una conexión TCP.

Recomendaciones y Propuestas

$\mu\mathcal{SIM}$ desde un inicio fue imaginado como un simulador con fines didácticos, esto implicó que su desarrollo se realizó de la forma más simple posible, y que fuera lo suficientemente versátil, como para que soporte la creación de módulos que extienda su funcionalidad. Recordemos que los escenarios posibles de simulación, son bastante extensos, y si queremos lograr un modelo fiel que responda de manera robusta a cualquier entorno, se deben considerar otras estructuras y un modelado según los requerimientos.

En un futuro, pensamos trabajar los siguientes aspectos para $\mu\mathcal{SIM}$:

- Continuar con el estudio e implementación de diferentes mecanismos de arranque rápido existentes hoy en día.
- $\mu\mathcal{SIM}$ es extendible siguiendo los principios básicos de comunicación aquí expuestos. Podríamos entonces, siguiendo el ejemplo de TCP, desarrollar a detalle cada una de las capas de modelo OSI.

- Existe aun trabajo por hacer en capa transporte de $\mu\mathcal{SIM}$. Por ejemplo, otros protocolos de recuperación de pérdidas pueden ser fácilmente añadidos. Implementar algoritmos alternativos como Reno, Tahoe, SACK, FACK, para realizar comparaciones entre las distintas maneras de recuperación de pérdidas. además UDP podría muy fácilmente ser estudiado y desarrollado. Para ello, sabemos que artificios como *sockets* y *puertos* son necesarios.

- Recordemos que una red puede estar compuesta por más de dos nodos origen o terminales. Entonces, para esto se debe desarrollarse la capa Red del modelo OSI. En esta capa debe atacarse la implementación de conceptos como el de enrutamiento y el de identificación de nodos.

Apéndice A

Guía detallada para extender $\mu\mathcal{SIM}$

Para complementar la sección 4.5.6 del documento, explicaremos a detalle como añadir una capa del modelo OSI a $\mu\mathcal{SIM}$ siguiendo los pasos propuestos. Para el ejemplo se añadirá una version inicial de capa Red de Modelo OSI.

Paso 1: Extender en la definición de la clase *Node* un nuevo tipo de nodo. En el archivo **node.h** existe el enumerado llamado *NodeType* que corresponde a los tipos de nodos existentes.

```
enum NodeType {
    SIMPLE_SRC,     // Nodo Simple que hace forward
    SIMPLE_END,     // Nodo Simple que recibe y consume
    APP,            // Capa Aplicacion
    ROUTER,         // Nodo Router
    TRANSPORT,      // Capa Transporte
    IP // Nueva Capa a añadir, Capa Red
};
```

Recordemos que en la figura 3.3 de la sección 3.1.2, explicamos como se manejan las capas de modelo OSI y cual es la analogía entre nodos y capas. Dicha figura se muestra a continuación.

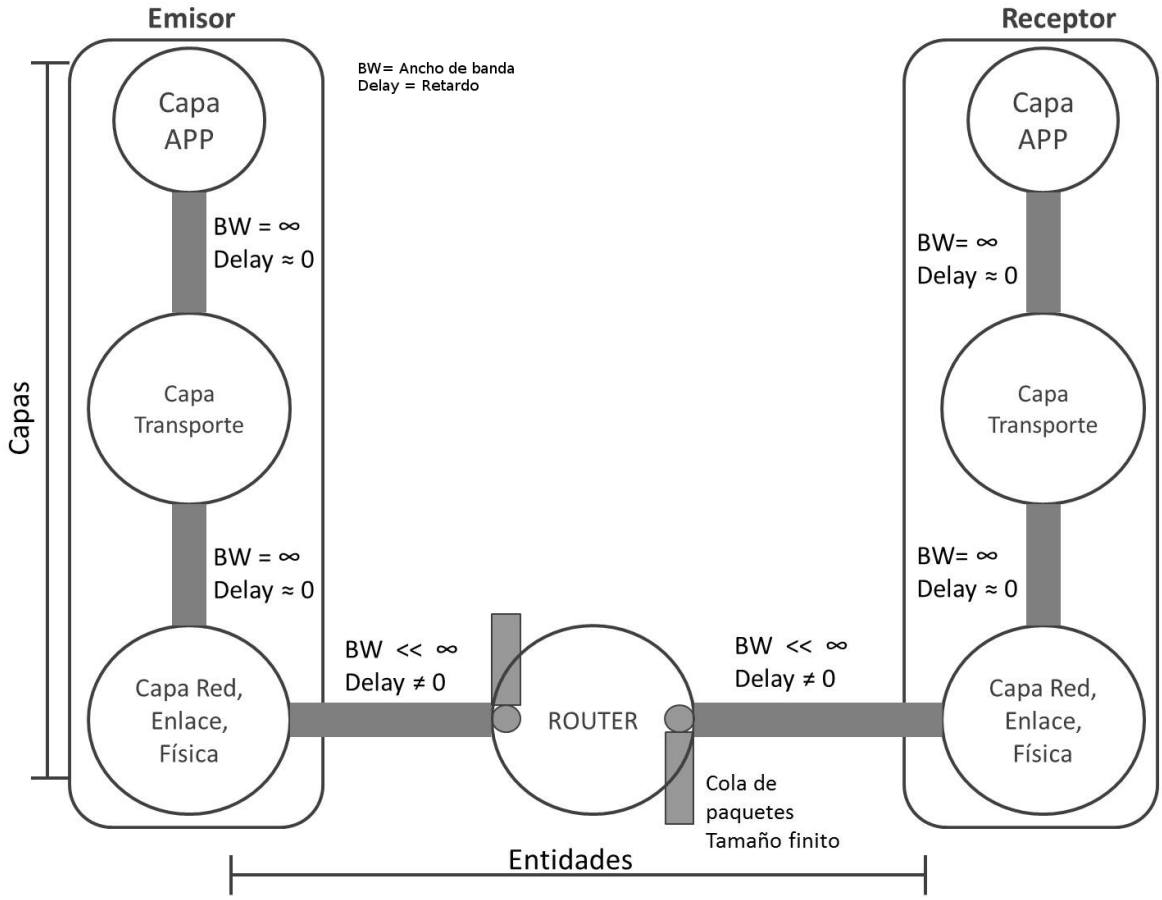

Figura A.1: Capas en $\mu\mathcal{SIM}$

Paso 2: Extender en la definición de la clase *Event*, todos los eventos asociados al nuevo nodo (hay que tener en cuenta los nodos conectados directamente al nuevo nodo). Los tipos de eventos están en el archivo **event.h** en el enumerado *EVENT_TYPE*.

```
enum EVENT_TYPE {
    PKT_ARRIVAL_TO_QUEUE,
    PKT_ARRIVAL_TO_SERVICE,
    PKT_DEPARTURE_FROM_NODE,
    .
    .
    .
    //Nuevos eventos correspondentes a capa RED
```

```
    PKT_ARRIVAL_TO_IP,
    PKT_DEPARTURE_FROM_IP,
    ACK_ARRIVAL_TO_IP,
    ACK_DEPARTURE_FROM_IP,
    LAST_EVENT
};
```

Recordemos los eventos asociados a las capas desarrolladas en el simulador $\mu\mathcal{SIM}$, por ejemplo la figura 4.3 de la sección 4.3 se muestran los eventos asociados al nodo enrutador, podemos ver que al menos ocurrirán 4 eventos: entrada y salida de datos del nodo, y entrada y salida de ACKs del nodo. Dicha figura se muestra a continuación.

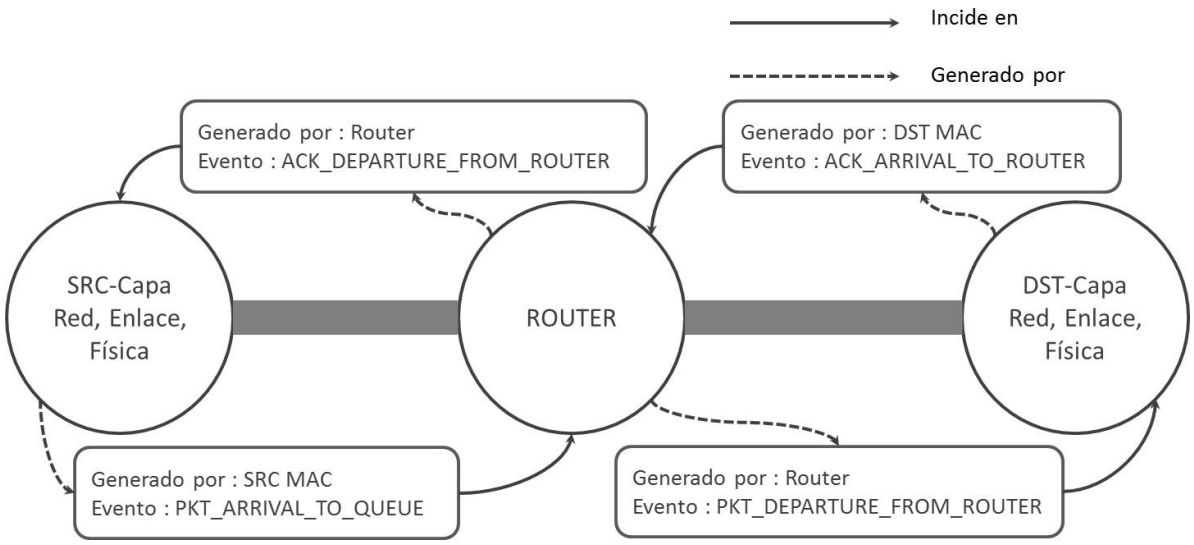

Figura A.2: Eventos asociados a nodo Router

Paso 3: Instanciar el nuevo nodo correspondiente a la capa que se está desarrollando, y enlazar con las capas adyacentes. En el archivo **usim.h** se encuentran todos los objetos que van a formar parte del simulador y es donde se instanciaran el nuevo nodo que corresponde a capa Red ademas se crearan los enlaces entre las capas.

```
Node * app = new Node(0, APP, "app in src", 2147483647);
Node * b = new Node(1, SIMPLE_SRC, "src iface", 2147483647);
```

```
Node * c = new Node(2, SIMPLE_END, "dst iface", 2147483647);
Node * tr_b = new Node(3, TRANSPORT, "transport layer", 2147483647);
Node * tr_c = new Node(4, TRANSPORT, "transport layer", 2147483647);
Node * r = new Node(5, ROUTER, "router", BUFFERSIZE);
//Nuevo nodo
Node * red = new Node(6, IP, "Capa Red", 2147483647);
```

Recordemos que los parámetros de entrada para instanciar una clase *Node* son: id, el cual en el caso del nuevo nodo red es ¨6¨, tipo de nodo el cual es ¨IP¨, etiqueta que lo describe que es ¨Capa Red¨ y el tamaño de cola el cual le damos un valor entero alto ya que los tamaños de colas en las capas tienden a tener un valor infinito.

Luego se enlazan los nodos.

```
Link * tr_b_to_red = new Link(true, -1, 0.001, tr_b, red, LINK_STACK);
tr_b->add_interface(tr_b_to_red);

Link * red_to_b = new Link(true, -1, 0.001,red,b , LINK_STACK);
red->add_interface(red_to_b);

Link * red_to_tr_b= new Link(true, -1, 0.001, red, tr_b, LINK_STACK);
red->add_interface(red_to_tr_b);

Link * b_to_red = new Link(true, -1, 0.001,b, red , LINK_STACK);
b->add_interface(b_to_red);
```

Se crean 4 enlaces nuevos para conectar de manera bidireccional el nuevo nodo con los nodos adyacentes capa transporte y capa física. Recordemos que los parámetros para instanciar un enlace son: bandera activo, el cual esta en verdadero e indica si el enlace está activo, ancho de banda el cual tiene un valor igual a ¨-1¨ [1], el retraso del

[1]Valor ¨-1¨ significa por definición inicial del programa que el ancho de banda entre esos nodos es infinito, esto se hace con el motivo de modelar las conexiones entre capas de un terminal ya que el ancho de banda entre capas es tan alto que su valor no afectará a los resultados obtenidos en el futuro

enlace que vale 0.001, el nodo origen, el nodo destino, y el tipo de enlace el cual en la versión 1 de $\mu\mathcal{SIM}$ solo se usa el tipo ¨LINK_STACK¨, pero se tiene planificado extender el concepto para futuras versiones.

Paso 4: Se implementa la manipulación del PDU dentro del manejador de los nuevos eventos, recordamos que tenemos que definir su comportamiento para cada uno de los eventos. En la version 1 de $\mu\mathcal{SIM}$, los manejadores de eventos se encuentran dentro del archivo **usim.h**

A continuación se indicara como se capturaría un evento y como se implementaría el manejador de dicho evento.

Primero se debe ubicar el ciclo principal que recorre la lista de eventos ordenados:

```
while (usim_sched.get_curr_time() <= TSIM) {
.
.
.
//Aquí se implementaría la captura del evento y su manejador
.
.
delete current_event;
current_event = usim_sched.dispatch();
}
```

Luego se escribe la estructura de decisión que va a capturar el evento, y dentro se implementa el manejador:

```
//Cuando un PDU sale de la capa Transporte y llega a la capa Red
if (current_event->get_from_node()->get_node_type() == TRANSPORT &&
                    current_event->get_type() == PKT_ARRIVAL_TO_IP)
{
//Se obtiene el PDU del evento
Pdu * _pdu = current_event->get_from_node()->pdu_departure();

/**********************
```

```
Aquí se puede manipular la cabecera del PDU para agregarle las
direcciones IP asociados al nodo destino y nodo emisor.
Su implementación se deja a criterio del desarrollador
**********************/

//Se planifica el siguiente paso del simulador.
//método plan_service() de la clase ListScheduler
usim_sched.plan_service(PKT_DEPARTURE_FROM_IP,
_pdu,current_event->get_to_node(),b);

}
```

Como aclaratoria el método plan_service() de la clase *ListScheduler*, sirve para planificar el próximo evento a ocurrir en el simulador, este método toma como parámetros de entrada el evento a planificar, el pdu a transportar, el nodo origen y el nodo destino.

Recordemos que debemos implementar la rutina que manipulará cada uno de los eventos nuevos creados.

Apéndice B

Repositorio del Proyecto

Un repositorio Git abierto alojado en GitHub se encuentra para todo aquel que desee estudiar $\mu\mathcal{SIM}$, la dirección actual del repositorio es el siguiente:

`https://github.com/eduardoGranados/uSim.git`

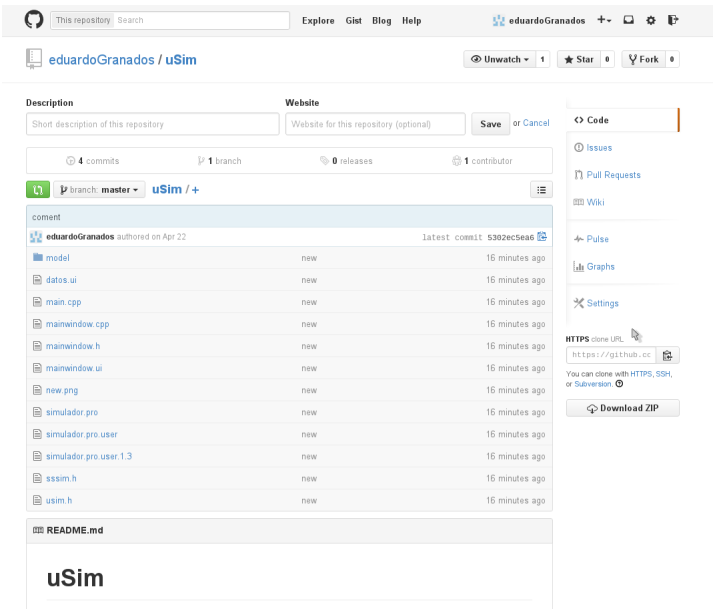

Figura B.1: Repositorio Git de $\mu\mathcal{SIM}$

Nota: La dirección del repositorio puede estar sujeta a cambios, se estará comentando su posible cambio en la misma dirección dada.

Bibliografía

[1] Sripanidkulchai and B. Maggs. An analysis of live streaming workloads on the internet. Proceedings of ACM IMC. ACM Press, 2004.

[2] E. Hossain T. Issariyakul. Introduction to network simulator ns2. New York, NY, USA, 2009.

[3] Jhon Postel. RFC 793: Transmission Control Protocol . Internet Engineering Task Force (IETF), 1981.

[4] A. Arcia-Moret, O. Díaz, and Montavont N. A Tunable Slow Start for TCP. Choroni, Venezuela, December 2012. The 4th Global Information Infraestructure and Networking Symposium GIIS 2012.

[5] A. Arcia-Moret. *Modifications du mécanisme d'acquittements du protocole TCP: évaluation et application aux réseaux filaires et sans fils.* PhD thesis, Ecole National Supériore des Télécomunications, 2009.

[6] Bob Briscoe. Problem Statement: Transport Protocols Don't Have To Do Fairness, draft-briscoe-tsvwg-relax-fairness-01. Internet Engineering Task Force (IETF) DRAFT, 2009.

[7] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. In *SIGCOMM Comput. Commun. Rev.*, volume 4, 2010.

[8] Samantha Gamboa and Andrés Arcia-Moret. *Estudio del Tráfico Transporte en una Red Satelital.* Universidad de Los Andes, 2011.

[9] A. Arcia-Moret, N. Montavont, J.-M. Bonnin, and D. Ros. TCP ACK Division Revisited. In *V CIBELEC*, Mérida, Venezuela, 2010.

[10] Anderson T. Savage S., Wetherall D. TCP congestion control with a misbehaving receiver. ACM SIGCOMM Computer Communications Review., 1999.

[11] Amine Dhraief, Abdelfattah Belghith, Nicolas Montavont, Jean-Marie Bonnin, and Mohamed Kassab. Ns2 based simulation framework to evaluate the performance of wireless distribution systems. In *Proceedings of the 2007 Spring Simulaiton Multiconference - Volume 1*, SpringSim '07, pages 264–271, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[12] Hachana Safaa and Montavont Nicolas. Design and Implementation of a Wireless Communication Networks Simulation Environment. Ecole Nationale des Sciences de l'Informatique Laboratoire Cristal, 2006.

[13] A. Lezama, A. A. Borrero, and Arcia-Moret. Diseño e Implementación de Algoritmos de Enrutamiento en Redes Completamente Opticas: Un estudio comparativo. Universidad de Los Andes, Mérida, 2013.

[14] Boehm B. A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes., 1986.

[15] Sally Floyd, Tom Henderson, Andrei Gurtov, and Y. Nishida. Rfc 6582: The newreno modification to tcp's fast recovery algorithm. Internet Engineering Task Force (IETF), 2012.

[16] ISO/IEC. ISO/IEC 7498-1: Open Systems Interconnection model (OSI), second edition. International Organization for Standardization (ISO), 1994.

[17] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[18] R. Thurlow. Rpc: Remote procedure call protocol specification. Internet Engineering Task Force (IETF), 2009.

[19] Allman M., Paxson V., and E. Blanton. Internet Standard Track RFC 5681: TCP Congestion Control. IETF, 2009.

[20] Stroustrup Bjarne. *The C++ Programming Language, 3rd edition.* Addison-Wesley, 1997.

[21] Summerfield M. Blanchette D. C++ GUI Programming with Qt 4., 2008.

[22] O. Diaz and A. Arcia-Moret. Diseño e implementación de un mecanismo de arranque rápido de flujos transporte: Un estudio en redes 802.11, 2012.

[23] M. Ciminieri, A. Dams, N. Montavont, and A. Arcia-Moret. Providing a seamless access to mobile users, 2013.

[24] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. RFC 3135: Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. Internet Engineering Task Force (IETF), 2001.