Universidad de Valladolid

E.T.S Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Técnicas de control de congestión

Autor:
D. David Madrigal Reoyo

Tutor:
Dña. Teresa Álvarez Álvarez

D. César Vaca Rodríguez

Agradecimientos

Gracias a mi tutora, Teresa Álvarez,
por la dedicación que presta a sus alumnos.
A mi familia, porque siempre está ahí
y a Leyre, porque sin ella esto no podría haber sido posible.

Tabla de contenido

Parte I

CAPITULO 1

1. INTRODUCCIÓN

1.1 Motivación

Actualmente Internet se ha convertido en un importante elemento de comunicación a nivel global, teniendo una gran influencia sobre la sociedad en general. A medida que ha aumentado su popularidad, también ha crecido en tamaño y se ha incrementado el tráfico generado en su seno.

Debido a todo esto, han ido surgiendo muchos trabajos de investigación orientados a optimizar el rendimiento de las comunicaciones que se realizan a través de Internet. Una de estas líneas de investigación consiste en procurar reducir las situaciones de congestión del tráfico que fluye a través de las redes de las que está conformado Internet.

Entre otras, una de las clasificaciones en las que se pueden dividir los métodos de gestión de la congestión es la siguiente:

- *Algoritmos extremo a extremo*: Estos algoritmos están implementados en los *hosts*, los extremos de la comunicación. El algoritmo de control de congestión que predomina en Internet es el que prevé el protocolo de transporte TCP. Este algoritmo, en sus diversas variantes, básicamente consiste en aumentar o disminuir la tasa de envío en función de los paquetes que ha detectado como perdidos.
- *Algoritmos de red*: Implementados en los nodos, intermediarios en la comunicación. Nos centraremos en los algoritmos de manejo de colas en los *routers*. Los nodos disponen de *buffers* de entrada y salida para cada enlace en los que se almacenan los paquetes pendientes y estos algoritmos tratan de mantener el número de paquetes en el buffer a un nivel determinado. El algoritmo de manejo de colas más extendido actualmente, no es realmente un algoritmo de control de congestión, es el llamado *Drop tail* (Descartar la cola), y simplemente consiste en aceptar todos los paquetes que llegan a la cola hasta que ésta se llena, momento a partir del cual se descartarán todos los paquetes que lleguen nuevos.

Sería un error ver ambos métodos de control de manera independiente ya que la forma en la que se gestiona la cola influye en la forma en la que un emisor TCP envía sus paquetes y viceversa. Cuando un algoritmo de gestión de cola decide descartar un paquete, está notificando indirectamente a los agentes TCP de la existencia de la congestión, con lo que estos reducirán su tasa de envío de paquetes reduciendo a su vez la congestión en la cola de los *routers*. Al usar conjuntamente TCP con *Drop tail* se producen efectos no deseados, hablando en términos de congestión, como son los producidos por los descartes de paquetes masivos al llenarse la cola, que provocan inestabilidad en el tamaño, variando el tiempo de respuesta y provocando en ocasiones el monopolio de los enlaces por unas pocas conexiones.

Tras observar estos efectos se propuso la creación de los llamados algoritmos de manejo activo de colas (AQM – *Active Queue Management*), que tratan de descartar algunos paquetes antes de que se produzca la congestión, aprovechando la dinámica de TCP, notificándole la congestión con suficiente antelación como para que responda correctamente, e incluso antes de que se produzca.

La técnica AQM más extendida y la única recomendada por la IETF es la denominada como RED (*Random Early Detection*), pero su efectividad ha sido muy discutida. Por esto, desde diversos campos de la ingeniería, como es el caso de la teoría de control, han surgido multitud de alternativas en forma de algoritmo. Debido a la proliferación de nuevas técnicas, se hace necesario un estudio pormenorizado de cada una de ellas ante una variedad de escenarios lo suficientemente amplia, como para comprobar su verdadera efectividad.

1.2 Objetivos

El propósito principal de este trabajo es el análisis y estudio comparativo de varios de los algoritmos de control de congestión AQM. Entre estos algoritmos se encuentran las técnicas tradicionales en el mundo de las comunicaciones, así como técnicas basadas en la teoría del control automático de procesos proveniente de la industria. Para llevar a cabo este proceso, se eligió la realización de pruebas con un simulador de redes de eventos discretos como es NS surgiendo así, como nuevas necesidades, el entender y documentar la estructura y funcionamiento del simulador, además de ampliar su funcionalidad extendiéndolo para añadir los controladores AQM no incluidos de forma predeterminada en el simulador.

En resumen, los objetivos del Trabajo de Fin de Grado son:

- Estudiar el problema de control de congestión TCP.
- Estudiar la interacción entre el control de congestión en TCP y el manejo de las colas en el *router* como origen de los algoritmos de control AQM.
- Realizar un estudio comparativo de los diferentes sistemas AQM mediante la realización de simulaciones en diversos escenarios representativos.
- Elaboración de las métricas necesarias y realización de una representación adecuada de los datos en bruto obtenidos del simulador.
- Entender el funcionamiento y estructura del simulador NS, y documentarlo.
- Incorporar nuevos módulos al NS para posibilitar la simulación de controles AQM no presentes de forma predeterminada en el simulador.

1.3 Estructura del documento

Esta memoria está estructurada en capítulos, que a su vez se dividen en cuatro partes, más una serie de anexos con información complementaria.

Parte I: Introducción

En esta parte se encuentran los capítulos relacionados con las bases teóricas y los detalles preliminares y de presentación del propio proyecto. En concreto, consta de dos capítulos:

· *Capítulo 1: Introducción*. Este capítulo se centra en la presentación de la motivación y objetivos del proyecto.

· *Capítulo 2: Control de congestión*. Este capítulo desarrolla los conceptos teóricos necesarios para abordar el proyecto.

· *Capítulo 3*: *Plan de desarrollo* En este capítulo se enumeran las técnicas y herramientas que se han utilizado para la elaboración de este proyecto, así como la planificación del mismo.

Parte II: Extensión del simulador NS

Esta segunda parte se dedica al simulador NS, y es donde se desarrolla tanto un análisis de sus características, como un estudio de su funcionamiento interno y de su estructura. Se realizará un pequeño trabajo de ingeniería de software en la parte concerniente al manejo de colas de este simulador. También se proporcionará una

explicación de las extensiones necesarias para añadir nuevos controladores. Todo esto acompañado de los diagramas UML necesarios.

· *Capítulo 4: Introducción y análisis.* En este capítulo se realiza un ligero acercamiento a la estructura del simulador NS y su arquitectura, haciendo un especial hincapié en su soporte del manejo de colas.

· *Capítulo 5: Extensión del software.* Aquí se detalla el diseño de las clases que se han creado para satisfacer las necesidades especificadas en los requisitos.

Parte III: Experimentos

En esta parte se explica todo lo relacionado con los experimentos destinados a realizar la comparativa de los algoritmos AQM, así como los resultados obtenidos de estos experimentos.

· *Capítulo 6: Diseño de los experimentos*. En este capítulo se muestran los tipos, cargas de tráfico, métricas usadas, cambios de parámetro a lo largo de ellos, etc.

· *Capítulo 7: Resultados*. Capítulo en el que se presentan explícitamente los resultados de cada uno de los experimentos, elaborando una comparativa de cada uno de los algoritmos usados en ellos.

Parte IV: Conclusiones

Esta parte se compone de un único capítulo en el que se exponen las conclusiones obtenidas tras la realización del estudio.

· *Capítulo 8: Conclusiones*.

Anexos

En los anexos se encuentran todos los temas relevantes que no han tenido cabida en el resto de la memoria, ya sea porque no se incluían dentro de los objetivos principales de este proyecto, o por ser detalles de instalación y no de carácter genérico.

CAPITULO 2

2 Fundamentos teóricos

Este capítulo comienza con una pequeña introducción a las redes de ordenadores, en la que se describe la conexión y el envío de datos entre un emisor y un receptor, así como el funcionamiento y arquitectura general de un *router*. Seguidamente, se plantea el problema de la congestión en la red, esquematizando escenarios en los que se genera, cuáles son los costes de la misma además de reseñar los retardos que sufre un paquete antes de alcanzar su extremo receptor. Mientras, en el siguiente punto, se habla de TCP y sus mecanismos de control de congestión, viendo cómo ha ido evolucionando, las investigaciones que se han realizado con el propósito de mejorar su reacción ante la congestión, en las que se han modificado ó añadido algoritmos. Se realizan a su vez simulaciones con NS-2 que permiten observar con claridad su comportamiento y evaluar los resultados.

Seguidamente se comentan los mecanismos de manejo de colas en los *routers*, empezando primeramente con el algoritmo *Drop Tail*. Se describe detalladamente y se explican sus flaquezas, las cuales conducen a plantear el desarrollo de AQM/RED. Se dedica un apartado para exponer los resultados de algunas simulaciones realizadas, que permiten observar gráficamente el comportamiento de la cola de un *router* al ejecutar *Drop Tail* o RED y, en el último caso, la necesidad de un buen ajuste teniendo en cuenta ciertos parámetros de la red, como el nivel de carga. AQM/RED puede notificar la congestión de dos formas, implícita (descartando el paquete) ó explícita (fijando un bit en la cabecera del paquete); para este último caso es necesaria la técnica de notificación de congestión explícita o ECN, que se detalla en este punto.

2.1 Introducción a las redes de ordenadores

Internet consiste en un conjunto de ordenadores de usuarios finales conectados por una infraestructura que transporta datos entre estos ordenadores. Esta infraestructura es, básicamente, un conjunto de *routers* conectados por enlaces. Cuando un sistema final tiene datos que enviar, fragmenta los datos dentro de unidades llamadas paquetes que llevan información de control dentro de sus cabeceras, como las direcciones IP fuente y destino del paquete. Cuando el paquete llega al *router*, éste usa la dirección IP destino de paquete como un registro clave y busca en su base de datos de información de

asignación de ruta para la decisión de reenvío. Si el *router* y el receptor están directamente conectados, el *router* envía el paquete al receptor, de lo contrario reenviará el paquete a otro *router* que está más próximo al receptor.

Observemos la **figura 2.1** que describe una conexión entre un emisor y un receptor a través de una interconexión de *routers*.

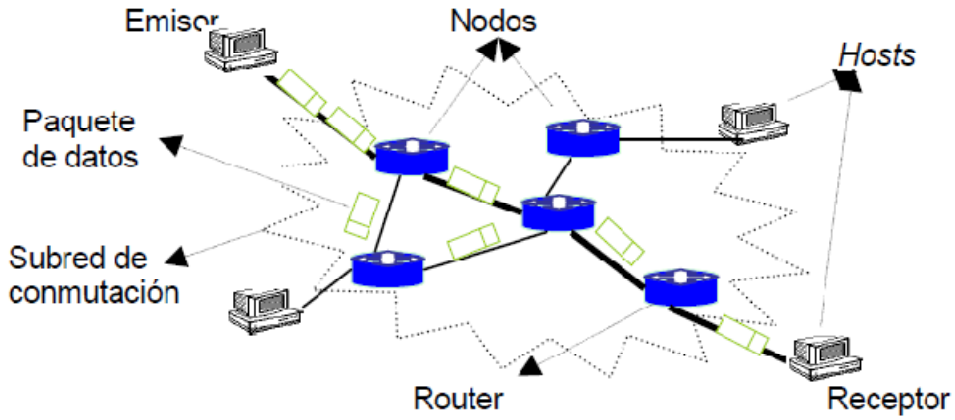


Figura 2.1. Conexión emisor-receptor a través de una interconexión de *routers*

2.1.1 Descripción general del router

Un *router* usualmente tiene múltiples interfaces de red asociadas a diferentes enlaces que conectan al *router* a otros *routers* ó subredes (una subred normalmente se compone de uno o múltiples sistemas finales). Si múltiples paquetes cuyo destino es el mismo enlace de salida llegan simultáneamente al *router* a través de diferentes enlaces de entrada, si estos no pueden ser reenviados en ese momento, tendrán que ser almacenados temporalmente en una cola del enlace de salida. La **figura 2.2** muestra la arquitectura básica de un *router*.

Si la tasa de envío del *router* es más grande que la tasa de llegada de los paquetes (el tiempo de transmisión promedio es más pequeño que el tiempo entre llegadas de los paquetes), la cola disminuirá llegando a estar vacía. Por otro lado, si la tasa de envío del *router* es menor que la tasa de llegada de los paquetes (el tiempo de transmisión promedio es más grande que el tiempo entre llegadas de los paquetes), la cola crecerá en un intervalo de tiempo hasta que la cola llegue a llenarse, si siguen llegando más paquetes estos se descartan, por consiguiente tenemos desbordamiento de cola.

Es útil distinguir entre dos clases de algoritmos implementados en el *router* relacionados con el control de congestión, estos son, la gestión de la cola y la planificación de la misma. Los algoritmos de gestión de la cola son los que manejan la longitud de la cola descartando paquetes cuando es necesario ó apropiado, es así como el *router* es capaz de notificar la congestión (forma implícita) a los emisores TCP. El algoritmo de gestión por defecto implementado en los *routers* es *Drop Tail,* que descarta todos los paquetes que llegan, si la cola ha superado su longitud máxima. Mientras los algoritmos de planificación determinan el siguiente paquete a ser transmitido por el enlace, son usados principalmente para manejar la asignación de ancho de banda entre los flujos. La cola es normalmente planificada de forma que el primer paquete que entra es el primero que sale, comúnmente conocido como FIFO (*first-in, first-out*) que traducido al castellano significa "primero en entrar, primero en salir". Con una cola FIFO los paquetes que llegan son encolados al final de la cola. Cuando el *router* realiza la transmisión de un paquete, éste desencola el paquete de la cabecera de la cola y lo transmite por el enlace.

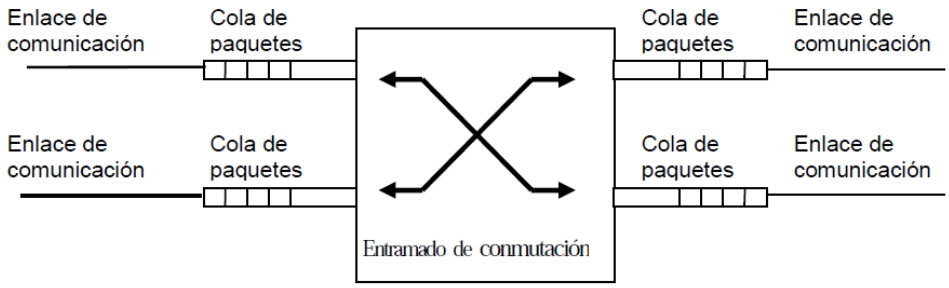


Figura 2.2. Arquitectura básica del *router*

2.1.2 Parámetros de las redes de ordenadores

La efectividad de los procesos que se ejecutan en los extremos de la red (host) depende en gran medida de la eficiencia con la cual la red entrega dichos datos. Es importante entender cuáles son los factores que impactan en el rendimiento de una red.

El rendimiento de una red se mide mediante los parámetros: ancho de banda, y latencia ó retardo.

Ancho de Banda

Es la medida de cuanta información puede fluir desde un lugar a otro en una cantidad de tiempo definido. Generalmente se mide en bits por segundo. Por ejemplo, una red podría tener un ancho de banda de 10 millones de bits por segundo (10Mbps), significando que es capaz de entregar 10 millones de bits cada segundo.

Mientras, se puede hablar del ancho de banda de la red como un todo y, algunas veces, se podría desear más precisión enfocándose, por ejemplo, en el ancho de banda de un solo enlace físico (en la capa física, el ancho de banda está mejorando constantemente). Es comparable al ancho de una tubería, ¿cuánto líquido puede pasar por la tubería?

Latencia o retardo

Es el tiempo empleado por un paquete para viajar de un extremo de la red al otro. Un paquete que comienza en un host (origen), puede pasar a través de un número determinado de *routers* y finalizar su viaje en otro host (destino).

Mientras un paquete va desde un nodo (host ó *router*) hasta el nodo subsiguiente (host ó *router*) a lo largo de su recorrido, los distintos retardos que sufre un paquete son: el retardo de proceso nodal, el retardo de cola, el retardo de transmisión y el retardo de propagación. Se van a explorar estos retardos en el contexto de la **figura 2.3** como una parte de su ruta terminal-a-terminal entre la fuente y el destino. Un paquete se envía desde el nodo anterior, a través del *router* A, al *router* B. El objetivo es categorizar el retardo nodal en el *router* A que tiene un enlace hacia el *router* B. Este enlace está precedido por una cola, donde el paquete puede experimentar un retardo antes de ser transmitido por el enlace, pero en el caso de que no esté transmitiendo en ese momento otro paquete y si no hay otros precedentes en la cola, podrá ser atendido.

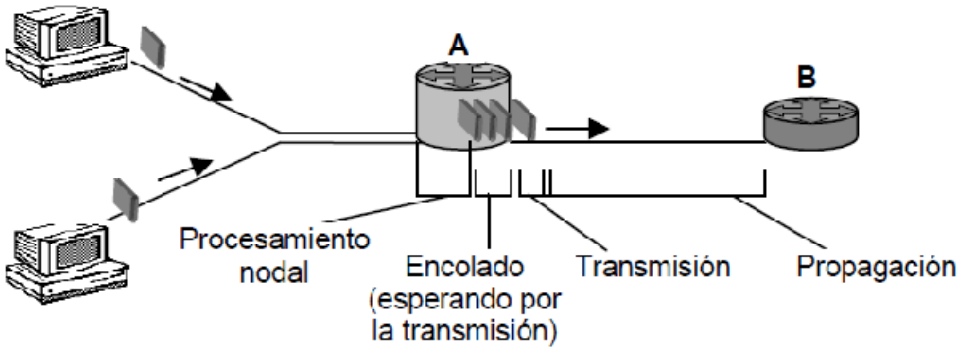


Figura 2.3. El retardo nodal en el *router* A.

Retardo de procesamiento

Al llegar un paquete al *router* A, éste examina la cabecera del paquete para determinar el enlace de salida correspondiente y lo dirige hacia dicho enlace. En este ejemplo, el enlace de salida para el paquete es uno que lo dirige hacia el *router* B. El tiempo que tarda en ello es el retardo de procesamiento nodal, que típicamente son pocos microsegundos.

Retardo de cola

El retardo de cola de un paquete específico dependerá del número de paquetes que hayan llegado anteriormente y que están encolados y esperando para la transmisión sobre el enlace. El retardo de un paquete dado puede variar significativamente de un paquete a otro. Si la cola está vacía y ningún otro paquete se está transmitiendo actualmente, entonces el retardo de cola del paquete en cuestión es cero. Por otro lado, si el tráfico es pesado y otros muchos paquetes están esperando para ser transmitidos, el retardo de cola será largo. Los retardos de cola pueden ser en la práctica del orden de microsegundos a milisegundos. ¿Cuándo es grande y cuando es insignificante el retardo de cola? La respuesta a esta pregunta depende en gran medida de la tasa a la que llegue el tráfico a la cola, de la velocidad de transmisión del enlace y de la naturaleza del tráfico que llega (es decir, de si el tráfico llega periódicamente o llega en ráfagas).

Retardo de transmisión

Es el tiempo que se requiere para transmitir todos los bits de un paquete en el enlace. Si el tamaño del paquete es MSS bits y, la tasa de transmisión del enlace desde el "*router*

A" al "*router* B" es C bits/seg. (La tasa C se determina por la tasa de transmisión del enlace, por ejemplo, para un enlace Ethernet de 10 Mbps, la tasa es C = 10 Mbps).

Entonces, el retardo de transmisión es MSS/C seg. Por lo tanto, el retardo de transmisión depende del tamaño del paquete y de la velocidad del enlace.

Retardo de propagación

Una vez que un bit es empujado al enlace, necesita propagarse al "*router* B". El tiempo necesario para propagarse desde el comienzo del enlace hasta el *router* B es el retardo de propagación. El retardo de propagación puede ser despreciable para un enlace que conecta dos *routers* en el mismo campus universitario (ejemplo, un par de microsegundos); pero si la distancia es mayor, éste puede ser de cientos de milisegundos. El retardo de propagación depende exclusivamente de la distancia. Cuanto más rápida es la red, más importancia tiene el retardo de propagación.

El retardo de propagación y de transmisión son totalmente independientes uno de otro. Conocer uno no implica saber algo del otro.

Producto de ancho de banda por latencia

También es importante hablar del producto de estas dos métricas. Intuitivamente, si se imaginara el canal entre una pareja de procesos como una tubería, el ancho de banda se considera como el ancho de la tubería, y la latencia como la longitud. Por lo tanto el producto será el volumen de la tubería, es decir, la cantidad de bits que puede contener; este valor informa de cuántos bits el emisor puede transmitir antes de que el primer bit llegue al receptor.

Se supone que la fuente desea confirmación de la recepción de información. Para ello se cuenta con otro canal con iguales características. Lo que interesa es el tiempo de ida y vuelta (RTT) antes que la latencia en una sola vía. Entonces, el emisor puede enviar hasta dos veces el producto en bits antes de recibir una señal del receptor, informando de que todo está bien. Los bits en el tubo están "en proceso", lo que significa que si el receptor dice al emisor que pare de transmitir, éste podría recibir hasta una cantidad igual al doble del producto de ancho de banda por el retardo antes de que el emisor esté listo para atender la solicitud de detener la transmisión. Pero hay otra circunstancia que se debe tener en cuenta, si el emisor no llena el tubo, no se utilizará completamente la red.

Por ejemplo, si se consideran dos enlaces de 1Mbps y 1Gbps, ambos con un RTT de 100ms. Se quiere transmitir un archivo de 1MB. En el enlace de 1Mpbs el archivo emplea

100 RTTs para ser transmitido, pero en el caso de 1Gbps el archivo tarda 1 RTT en ser transmitido.

Ahora si se quiere transmitir un archivo de 1KB, considerando los mismos enlaces, ambos con un RTT de 100ms, el archivo tarda el mismo tiempo en los dos enlaces (1 RTT).

Jitter

Se denomina *jitter* a la variabilidad del tiempo de ejecución de los paquetes. Esta media es de suma importancia para las aplicaciones multimedia que han llegado a hacerse increíblemente populares sobre la red IP. En particular, la voz sobre IP ó VoIP y la videoconferencia en tiempo real. Sin embargo, debido a la naturaleza de "mejor esfuerzo" de las redes IP, el tráfico en tiempo real puede experimentar retardos y variaciones del retardo, consiguiendo resultados totalmente insatisfactorios, particularmente cuando la red está congestionada. Por consiguiente, sería necesario proporcionar mecanismos de calidad de servicio (QoS) que permitan mejorar la transmisión de dicha información.

Rendimiento de las aplicaciones

La aplicación es quien marca la pauta en la decisión de cuál de las métricas es más importante. Por ejemplo, una aplicación donde un cliente envía un mensaje de un byte a un servidor y recibe, como respuesta, otro mensaje de un byte es una aplicación restringida por la latencia. La aplicación se comportará diferente sobre un canal transcontinental con un tiempo de ida y vuelta de 100 milisegundos que sobre una LAN con un tiempo de ida y vuelta de un milisegundo; si el canal es de 1 Mbps ó de 100 Mbps es poco importante. Los requerimientos de latencia de una aplicación pueden ser más complicados que decir "deme la latencia más pequeña que pueda". Algunas veces no importa demasiado si el retardo en una vía es de 100 ms ó de 500 ms, es más importante saber si la latencia varía de paquete a paquete.

Considérese la situación en que una fuente envía un paquete cada 33 milisegundos, como puede ser el caso de una aplicación de video transmitiendo *frames* 30 veces por segundo. Si los paquetes llegan a su destino espaciando exactamente los 33 milisegundos, entonces se puede deducir que el retardo experimentado por cada paquete en la red fue exactamente el mismo. Si el tiempo entre paquetes que llegan al destino es variable, entonces el retardo experimentado por la secuencia de paquetes fue variable y se dice que la red ha introducido *jitter* al flujo de paquetes. Esto sucede con

mayor probabilidad cuando los paquetes experimentan diferentes retardos en colas de espera en una red de conmutación de paquetes con múltiples saltos.

Para comprender la importancia del *jitter*, se puede suponer que los paquetes que están siendo transmitidos sobre la red transportan *frames* de video y para mostrar estos *frames* en la pantalla del receptor deben llegar cada 33 milisegundos. Si un *frame* llega antes, entonces éste puede ser guardado por el receptor hasta que sea el momento de mostrarlo, pero si el *frame* llega tarde, entonces el receptor no tendrá el *frame* que necesita para mostrar en la pantalla, la calidad del video sufrirá y no será fluida. Obsérvese que no es necesario eliminar el *jitter*, sólo saber que tan mal puede llegar a estar. La razón para hacer la anterior afirmación es que si el receptor conoce los límites inferior y superior de la latencia que un paquete puede experimentar, éste puede retardar el tiempo en el cual comience a reproducir el video (es decir, establecer cuándo es el momento adecuado para mostrar el primer *frame*) de tal forma que asegure que cualquier *frame* que se requiera desplegar ya haya llegado al receptor.

En contraste, considérese una aplicación de una biblioteca digital a la cual se le solicita una imagen de 25MB, cuanto más grande sea el ancho de banda, más rápido podrá recuperar la imagen el usuario. En este caso, el ancho de banda es más importante que la latencia. Supóngase que el canal tiene un ancho de banda de 10 Mbps, le tomará 20 segundos transmitir la imagen, sin importar si la latencia es de 1 milisegundo ó de 100 milisegundos. La diferencia entre un tiempo de respuesta de 20.001 segundos y 20.1 segundos es despreciable.

Las aplicaciones pueden generar tráfico a ráfagas. Si esta tasa de transmisión pico es más alta que la capacidad del canal disponible, entonces el exceso de datos debe ser almacenado temporalmente en alguna parte para ser transmitido después. Si se sabe qué tan grande puede ser la ráfaga generada y transmitida por la aplicación el diseñador de la red podrá asignar suficiente capacidad de almacenamiento temporal para almacenar dicha ráfaga.

2.2 El problema de la congestión

La congestión puede ser definida como un estado de la red en la cual la demanda total de recursos excede la capacidad disponible.

Se describen a continuación un par de escenarios donde se produce congestión, se verá la causa y el efecto de ésta. La **figura 2.4** muestra dos emisores, dos receptores y un

router con tamaño de búfer infinito, los emisores envían a una tasa media de λ_e bytes/seg. Los paquetes del *host* A y los del *host* B pasan a través del *router* y sobre un enlace compartido de salida de capacidad R.

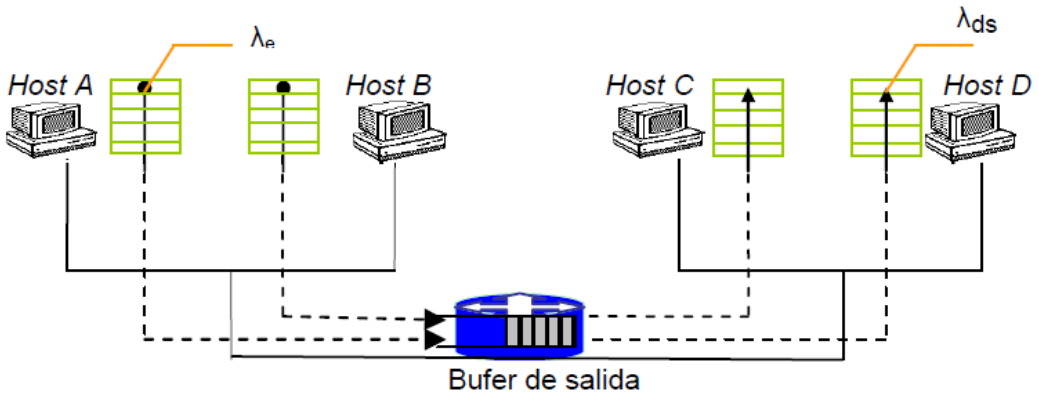


Figura 2.4. Escenario de congestión: dos conexiones compartiendo un único salto

La **figura 2.5** esquematiza las prestaciones de la conexión del *host* A. La gráfica de la izquierda representa la productividad por conexión (el número de bytes por segundo en el receptor) como una función de la tasa de envío de la conexión. Para una tasa de envío entre 0 y R/2, la productividad en el receptor es igual a la tasa de emisión del receptor. Cuando la tasa de emisión crece por encima de R/2, la productividad es, sin embargo, de sólo R/2. Este límite superior de la productividad es una consecuencia de la compartición de la capacidad del enlace entre dos conexiones.

Conseguir una productividad por conexión de R/2, es óptimo dado que el enlace está completamente ocupado entregando paquetes a sus destinos. A medida que crece la tasa de emisión, el retardo tiende a infinito. Véase la gráfica de la derecha de la **figura 2.5.** "Como se puede observar, se experimenta mayor retardo en la cola según la tasa de llegada de paquetes se acerca a la capacidad del enlace."

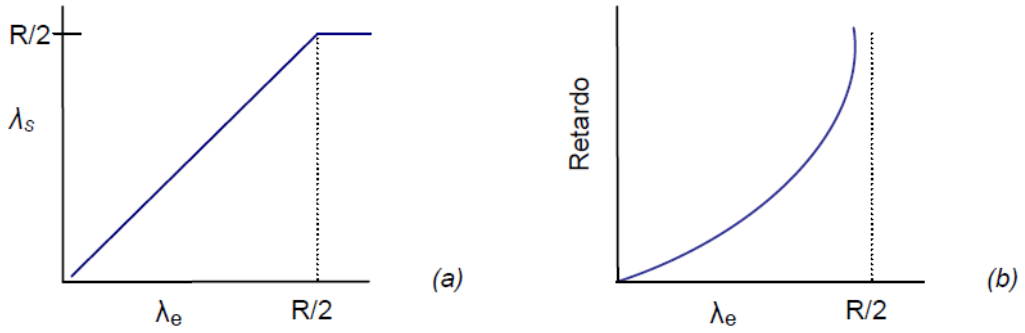


Figura 2.5. Productividad (a) y retardo (b) como función de la tasa de emisión del *host*.

Se limita ahora la capacidad del búfer. Los paquetes serán descartados por el *router* cuando supere su capacidad. Suponiendo que el emisor retransmite un paquete cuando tiene la certeza de que éste se ha perdido, se tendría ahora que $\lambda'_e = \lambda_e$ + retransmisiones. Por lo que los costes de la congestión serían las "retransmisiones necesarias para poder compensar los paquetes descartados ó dejados caer debido al desbordamiento del búfer realizando un mayor trabajo y utilización de recursos que pudo ser innecesario".

También es posible realizar retransmisiones innecesarias en presencia de grandes retardos, en caso de que el emisor supone que el paquete se ha perdido, pero éste puede estar en camino para alcanzar su destino, por lo tanto hacen que el ancho de banda del enlace se utilice para reencaminar copias innecesarias de paquetes. Se van a esquematizar los efectos de la congestión en la **figura 2.6.**

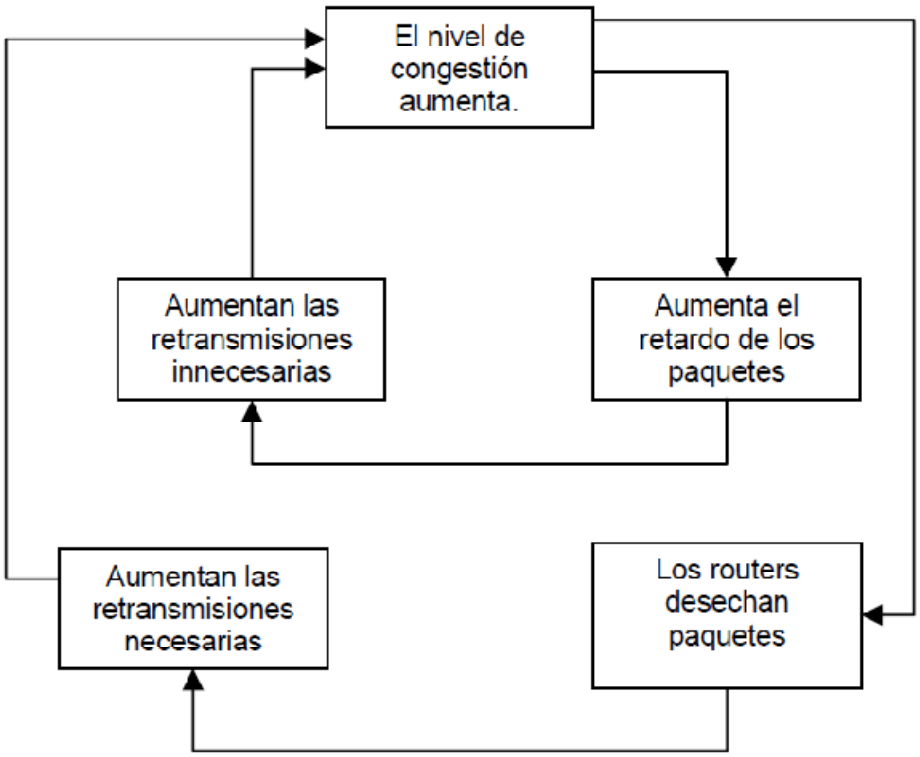


Figura 2.6. Efectos de la congestión.

2.3 Control extremo a extremo: TCP

2.3.1 Descripción de TCP

El protocolo TCP (*Transmission Control Protocol*) forma, junto a IP, el corazón del modelo de referencia seguido en Internet. IP permite el enrutamiento de paquetes entre redes. Sin embargo, no tiene garantías sobre la distribución. TCP, es el protocolo de transporte que ofrece un servicio fiable y la regulación del flujo de datos del origen al destino. La entidad TCP en cada extremo de una conexión asegura que los datos se entreguen a la aplicación local de forma:

- Precisa
- En secuencia
- Completa
- Libre de duplicados y errores

Esta fiabilidad la consigue usando ventanas deslizantes, números de secuencia y un proceso de sincronización que asegura que cada *host* está listo y dispuesto a comunicar, como se muestra en la **figura 2.7.**

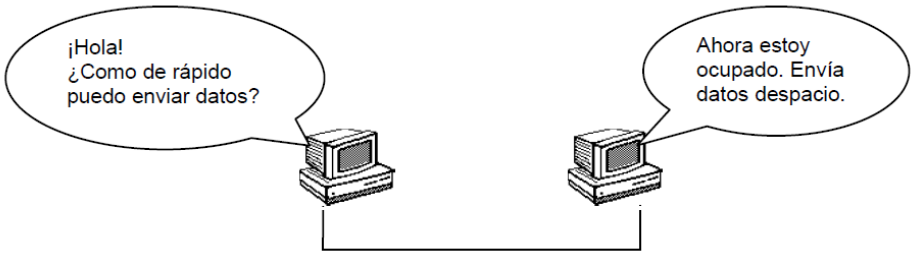


Figura 2.7. Funcionamiento TCP

¿Cómo trabaja TCP? TCP divide los datos de la aplicación, antes de transmitirlos, en unidades de información llamadas segmentos (en cada capa de la arquitectura TCP/IP, las unidades de datos (PDU) tiene un nombre específico, es decir, en la capa de transporte la PDU se denomina segmento, en la capa de red paquete ó datagrama, en la de enlace trama. Utilizamos la definición de paquete de forma general). A cada unidad de información se le asigna un número de secuencia con el fin de llevar un control de la llegada de los segmentos al destino. Y la forma en la que el emisor sabe que los segmentos han llegado a su destino es a través de los "reconocimientos (ACK, siglas en ingles)". Los que tienen como objetivos principales regularizar el ritmo de transmisión de TCP, asegurando que los segmentos pueden ser transmitidos solamente cuando otros segmentos han salido de la red, informando a la fuente para que ella sepa si tiene que reenviar algún segmento.

Como TCP es un protocolo orientado a conexión. Antes de la transmisión de datos, los *hosts* atraviesan un proceso de sincronización para establecer una conexión virtual. Este proceso de sincronización garantiza que ambos lados están listos para la transmisión de datos. Este proceso se conoce como establecimiento de conexión de tres vías.

La sincronización se consigue intercambiando segmentos de la siguiente manera:

1. El proceso iniciador o proceso activo, envía un mensaje de sincronización (*flag* SYN activo, del encabezado del segmento) al proceso con quien quiere comunicarse, especificando en el campo de número de secuencia, un número de secuencia inicial (ISN); y algunas opciones más.

2. Si el receptor de este mensaje desea la comunicación, responde con un acuse de recibo (*flag* ACK activo, indicando que el campo de acuse de recibo es válido) y con un mensaje de sincronización (*flag* SYN activo) especificando su propio ISN, su ventana inicial y las probables opciones negociadas.
3. La fase de conexión termina cuando el iniciador envía acuse de recibo de este último mensaje de sincronización.

Una vez superada esta fase, el emisor estará listo para enviar datos. Uno de los servicios proporcionados por TCP es el **control de flujo,** que regula cuantos datos se envían durante un periodo de transmisión dado. Se hablará de esto detalladamente más adelante.

Por último, una vez finaliza la transmisión, la siguiente fase será la desconexión.

2.3.2 Detección de pérdida en TCP

Dado que los paquetes pueden perderse en el tránsito entre el emisor y receptor debido a los desbordamientos de las colas de los *routers* en una red IP. Un servicio de red para el intercambio fiable de datos entre las aplicaciones requiere que los sistemas finales (*host*) implementen mecanismos para detectar la pérdida de paquetes y retransmitir el paquete perdido. TCP detecta la pérdida de paquetes asociando a cada byte de dato un número identificador de secuencia.

El receptor TCP reconoce la recepción de un paquete de datos enviando al emisor un paquete de reconocimiento (ACK). El paquete ACK lleva un número de secuencia indicando el número de secuencia en orden del siguiente byte que espera el receptor. Así, un paquete ACK da a conocer al emisor que todos los bytes de datos que tienen un número de secuencia más pequeño que el paquete ACK han sido recibidos por el receptor. Por cada paquete de datos fuera de orden que llega al receptor, éste retransmite un ACK que contiene el número de secuencia del último byte de dato que ha sido recibido en orden.

En el caso que el emisor pueda enviar múltiples paquetes en secuencia, una manera rápida de detectar pérdida de paquetes de datos es con un uso directo de duplicados ACK. Como se menciona arriba, el receptor envía duplicados ACK para informar al emisor qué paquetes de datos llegan al receptor fuera de orden.

Los duplicados ACK pueden ser generados por la pérdida de paquetes de datos. En ese caso, el receptor genera un duplicado ACK por cada paquete de datos recibido en el receptor después del paquete perdido. Otro caso es, por reordenamiento de paquetes

de datos en la red, porque cada paquete de datos que llega al receptor fuera de orden ocasiona que el receptor genere un duplicado ACK, de estos casos con diferencia la pérdida es la más común. Los duplicados ACK ayudan al emisor a detectar y retransmitir los paquetes perdidos. Sin embargo, los duplicados ACK se generan únicamente cuando un paquete de datos perdido es seguido por otros paquetes de datos que llegan al receptor. Si el emisor transmite una secuencia de paquetes de datos y el último paquete de datos de la secuencia se pierde, el receptor no puede detectar la pérdida del paquete de datos. En ese caso, el receptor no genera duplicados ACK para informar al emisor acerca de la pérdida del paquete.

Además, los duplicados ACK transmitidos por el receptor pueden perderse en el camino de vuelta al emisor. Debido a esos problemas, el emisor necesita contar con otra técnica local para detectar la pérdida de sus paquetes de datos. El emisor arranca un temporizador para cada paquete de datos que éste transmite. Si el temporizador expira y el reconocimiento de ese paquete no ha sido recibido, asume que el paquete de datos se ha perdido y lo retransmite.

2.3.3 Estimación del tiempo de ida y vuelta y el límite de espera

El valor del temporizador de retransmisión, usado por una entidad TCP emisora para determinar cuándo se debe retransmitir un segmento, generalmente será considerablemente más grande que el RTT. Para ello, se calcula una estimación de RTT.

La estimación de RTT juega un rol importante en el comportamiento de TCP. Los tiempos de ida y vuelta experimentados por una conexión normalmente consisten del retardo de propagación y del retardo de encolado.

Los retardos de propagación normalmente son constantes, pero los retardos de encolado pueden variar porque las colas de los *routers* a lo largo del camino de una conexión pueden crecer ó disminuir en el tiempo. Debido a la variabilidad de los retardos de encolado, no es fácil para los sistemas finales estimar el tiempo de ida y vuelta con exactitud. Sin embargo, una estimación exacta del tiempo de ida y vuelta permite a TCP fijar su temporizador eficientemente. Una subestimación de los tiempos de ida y vuelta pueden conducir a intervalos de espera demasiado cortos; si se usa para detectar perdida de paquetes de datos en el emisor podrían causar retransmisiones innecesarias de los paquetes y un derroche de ancho de banda. Por otro lado, una sobre estimación de los tiempo de ida y vuelta pueden causar que el emisor espere demasiado para detectar una pérdida de paquete y incremente el retardo de las aplicaciones que se ejecutan sobre TCP.

Se calcula la estimación de RTT de la siguiente manera:

EstimaciónRTT = (1-a)*EstimaciónRTT + (a)* MuestraRTT

Además de tener una estimación del RTT, también es importante la variabilidad del RTT que se mide en TCP mediante la expresión:

DevRTT = (1-b)*DevRTT + b*|MuestraRTT – EstimaciónRTT|

Según la RFC 2988 los valores recomendados para a= 1/8, b=1/4.

A partir de las anteriores estimaciones, se calcula el tiempo de espera del temporizador:

IntervaloEspera = EstimaciónRTT + 4 DevRTT

2.3.4 Control de congestión en TCP

El mecanismo de control de congestión de TCP permite que cada lado de la conexión mantenga una variable denominada ventana de congestión (*cwnd*), la cual impone una restricción sobre la tasa a la que el emisor TCP puede enviar tráfico en la red. Concretamente, la cantidad de datos pendientes de reconocimiento en el emisor no puede exceder del mínimo entre la ventana de congestión y la ventana de recepción (*rwnd*), por tanto *current_window = min (cwnd, rwnd)*. Donde *current_window*, es la ventana actual ó ventana en uso que representa la cantidad de información que envía el emisor cada RTT. La ventana de recepción es la restricción que impone el receptor al emisor informándole de la capacidad disponible en su búfer. Básicamente se trata de que un emisor rápido no ahogue a un receptor lento. De esta forma se realiza el control de flujo entre los extremos de la red.

En la cabecera del segmento TCP, se define un campo de 16 bits para el tamaño de ventana del receptor, es decir, el tamaño de ventana máximo está limitado a 65.535 bytes o 65KB. Pero para una mayor eficiencia en redes de gran ancho de banda, debe usarse un tamaño de ventana mayor. Como el campo de ventana no puede expandirse se usa un factor de escalado. La escala de ventana es una opción usada para incrementar el máximo tamaño de ventana a 1Gigabyte.

Para poder centrarnos en el control de congestión (como opuesto al control de flujo), se asume que el búfer de recepción es tan grande que la restricción asociada a la ventana de recepción puede ser ignorada; por ello, la cantidad de datos no reconocidos en el emisor estaría limitada únicamente por la ventana de congestión.

Durante un intervalo concreto de ida y vuelta RTT, la tasa de envío del emisor dependerá de la ventana actual o ventana en uso, y del tiempo de ida y vuelta actual, por lo tanto la tasa de envío será aproximadamente *current_window/RTT* paquetes/segundo.

La modificación de la ventana de congestión se realiza en función del RTT, lo que puede considerarse perjudicial en casos que este valor sea alto, haciendo más lenta la reacción del protocolo.

TCP intenta conseguir una alta utilización del ancho de banda, controlando su tasa de envío, realizando una utilización efectiva de los recursos de la red. Véase cómo lo hace:

Arranque lento y prevención de congestión

Cuando una nueva conexión TCP se establece, el emisor no sabe el estado real de la red, por lo tanto para no inundarlo con muchos segmentos, el tamaño de la ventana inicial de envío es menor ó igual a 2 paquetes (en la fase de establecimiento de la conexión, se define la cantidad máxima de datos que puede llevar un segmento, esto es el MSS (*Maximun Segment Size*, Tamaño Máximo de Segmento)). Pero sería ineficiente que en cada tiempo de ida y vuelta simplemente se envíe un solo paquete, dado que el ancho de banda para la conexión puede ser mucho mayor. Por consiguiente, por cada ACK recibido correctamente, el emisor incrementa su ventana de congestión en un paquete. En cada tiempo de ida y vuelta, su tamaño se duplica. La ventana de congestión va aumentando exponencialmente si no hay congestión. Este comportamiento se denomina “arranque lento” o “comienzo lento”.

El aumento del tamaño de la ventana de congestión depende de los ACK’s que se vayan recibiendo desde el receptor, por lo que igualmente se está dependiendo de la congestión que pudiera existir o no en la red. Se puede decir que a mayor congestión, menos ACK’s recibidos y, por tanto, la ventana de congestión será más pequeña y enviará menos segmentos TCP en ese instante de tiempo.

El mecanismo de “prevención de congestión” frena al emisor tratando de evitar situaciones de congestión severa. La idea es la siguiente: fija un umbral (puede tomar inicialmente, bien el valor anunciado por *rwnd*, ó la cota superior de *rwnd*) de tal manera que:

- Si *cwnd* < umbral, entonces se encuentra en situación de “arranque lento” donde: *cwnd* = *cwnd* + 1.

- si *cwnd* > umbral, entonces estamos en situación de "prevención de congestión" donde por cada ack recibido la ventana de congestión crece aproximadamente en 1/*cwnd*, esto es: *cwnd* = *cwnd* + 1/*cwnd*.

  *En caso de que ambas sean iguales se puede utilizar cualquiera de los dos algoritmos.

Imagínese que *cwnd* sea 8, entonces se pueden enviar 8 paquetes seguidos. Si no hay congestión, se recibirán los correspondientes 8 ACK's y, si se está en situación de "no congestión" entonces *cwnd* pasará a ser aproximadamente 9. Obsérvese estos dos comportamientos en la **figura 2.8.**

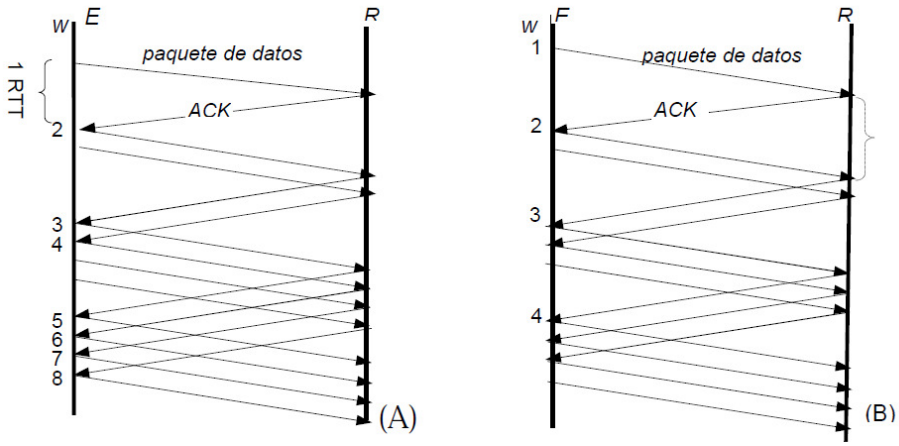


Figura 2.8. Emisor-(A) Arranque lento – (B) Prevención de congestión

La **figura 2.8 (A)**, muestra el crecimiento en el número de paquetes en tránsito durante el arranque lento. Si es un incremento exponencial, entonces ¿por qué se denomina "arranque lento"?. La respuesta está en la historia en los orígenes de TCP. ¿Qué pasa cuándo se establece una conexión, no hay paquetes en tránsito y la fuente comienza a enviar paquetes?. Si el extremo fuente de la conexión envía tantos paquetes como lo permite la ventana del receptor -exactamente lo que se hacía en la antigüedad- los *routers* no son capaces de consumir la avalancha de paquetes que llegan y todo va a depender de la capacidad de los búferes disponibles en los *routers*. En otras palabras, este mecanismo es más lento que enviar un número de paquetes dado por la ventana de recepción.

Pérdida de paquete

La pérdida de un paquete indica que hay congestión (téngase en cuenta que en las redes cableadas de hoy en día es difícil que se descarte un segmento por errores en la transmisión). Ante ese evento el emisor TCP reducir su tasa de envío para evitar la inestabilidad en la red. Como se menciona líneas arriba, la pérdida de paquetes puede ser detectada por duplicados ACK ó por *timeout*.

Cuando ocurre este evento tanto la ventana de congestión (*cwnd*) como el umbral se modifican. El valor asignado depende de si se tienen activados los mecanismos de retransmisión rápida y recuperación rápida".

El objetivo de la "**retransmisión rápida**" es acelerar la retransmisión del segmento de datos perdidos, si se reciben tres ACK's duplicados sin esperar a que venza su temporizador. Con este algoritmo, ante un evento de pérdida (por *timeout* ó por tres ACK's repetidos) el umbral se reduce a la mitad del valor de la ventana de transmisión pero no por debajo de dos es decir: *umbral = max (2,0.5*current_window)*, donde *current_window=min (cwnd,rwnd)* y la ventana de congestión (*cwnd*) toma el valor de uno, comenzando la fase de "arranque lento".

El algoritmo de "**recuperación rápida**" permite mejorar el rendimiento. La forma en que lo hace es aplicando únicamente "arranque lento" cuando se retransmite por *timeout* y no por tres ACK's duplicados. El algoritmo entiende que cuando llega el tercer ACK duplicado de un mismo segmento (supone paquete perdido) se establece el umbral a la mitad del valor de la ventana de transmisión *umbral=max (2,0.5*current_window)*. Sin embargo, no entra en escena el algoritmo de "arranque lento", sino que se retransmite el segmento perdido y se establece *cwnd = umbral + 3*.

Este tres entra en escena puesto que si han llegado tres ACK's del segmento en cuestión es porque los segmentos consecutivos a éste han llegado a su destino. Igualmente, podría darse el caso de que siguiesen llegando ACK's relacionados a la retransmisión de este segmento; es decir, mientras que se realiza la operación de retransmisión podría haber segmentos en la red pendientes de llegar a su destino y ante esto, se incrementa la ventana de congestión por cada uno de estos ACK, es decir, *cwnd = cwnd +1*. Esto ocurre hasta que el emisor recibe el ACK correspondiente del segmento perdido en cuestión y, retransmitido posteriormente, igualando justo en este momento el tamaño de la ventana de congestión *cwnd* al tamaño del umbral comentado anteriormente.

Con esto se puede ver que se ha retransmitido el segmento perdido mientras que se sigue en la fase de "prevención de congestión", sin tener que entrar en la fase de "arranque lento". El umbral se reduce a la mitad del tamaño de la ventana de envío en el momento en que el emisor es consciente de la pérdida del segmento. El mecanismo de arranque lento, es solo usado al comienzo de una conexión y toda vez que ocurre *timeout*.

Los mecanismos de retransmisión rápida y recuperación rápida lo podemos resumir como sigue:

- **Sin "retransmisión rápida y recuperación rápida"** Si hay que retransmitir un segmento (por vencimiento de temporizador o por recepción de ACK's duplicados): umbral = *max(2, 0.5*cwnd) cwnd* = 1 (siempre se empieza con "arranque lento").
- **Con "retransmisión rápida y recuperación rápida"** Si hay que reenviar un paquete por vencimiento del temporizador: (congestión seria)

  umbral = *max (2, 0.5*cwnd)*;
  cwnd = 1 (se empieza con "arranque lento");

  Si se reciben más de tres ack duplicados seguidos

  - umbral = *max (2, 0.5*cwnd)*
  - retransmitir el segmento perdido y fijar *cwnd* = umbral + 3 Lo que se conoce como inflado artificial de *cwnd*, hace crecer *cwnd* en la cantidad de segmentos (ACK en este caso) que abandonaron la red
  - por cada ACK duplicado adicional recibido, incrementar *cwnd* en un paquete, es decir *cwnd* = *cwnd* + 1, se transmite un segmento si es permitido por el nuevo valor de la ventana.
  - si el siguiente ACK que llegue reconoce nuevos datos, entonces *cwnd* = umbral (valor fijado al principio), ejecutar "prevención de congestión" (Este ACK debe reconocer todos los segmentos intermedios enviados entre el segmento perdido y la recepción del tercer ACK duplicado, si ninguno de estos se ha perdido).

Generalmente estos mecanismos son nombrados en Ingles: *"Slow Start", "Congestion Avoidance"," Fast Retransmit" y "Fast Recovery".*

La **figura 2.9** muestra el comportamiento de la ventana de congestión en el tiempo.

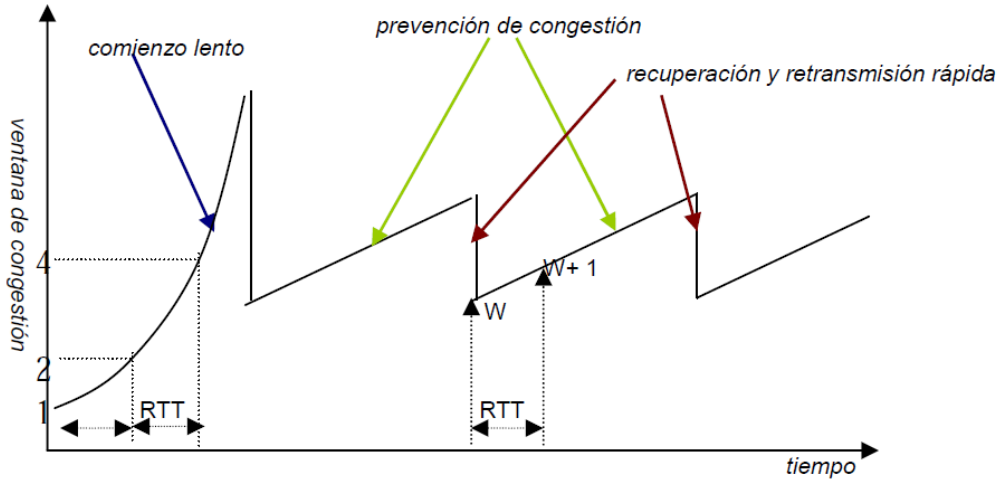


Figura 2.9. Comportamiento de la ventana de congestión en el tiempo.

2.3.5 Variantes TCP

A principio de los años 80, el equipo de Investigación en Sistemas de Cómputo de la Universidad de California, recibió un contrato para implementar la pila de protocolos TCP/IP en el núcleo de sus versiones del sistema operativo UNIX. Tradicionalmente Berkeley ha distribuido el código fuente de sus sistemas de manera gratuita y éste ha sido el estándar de los protocolos de redes de Internet. Un porcentaje muy importante de las implementaciones actuales, se deriva directamente de las versiones de UNIX 4.3BSD Tahoe y 4.3BSD Reno.

TCP es un protocolo desarrollado y complejo. Mejoras significativas han sido propuestas y llevadas a cabo a lo largo los años, algunas de las mejoras son la gestión de los temporizadores de retransmisión y la gestión de la ventana.

Se fueron creando diferentes versiones de TCP, algunas de ellas se mencionan a continuación.

2.3.5.1 TCP TAHOE

TCP Tahoe (1988), incluye los algoritmos de comienzo lento, prevención de congestión y retransmisión rápida, explicadas líneas arriba. Las mejoras también incluyen una modificación al estimador del tiempo de ida y vuelta, utilizados para fijar los valores de *timeout* de retransmisión.

2.3.5.2 TCP RENO

TCP Reno (1990), mantiene las mejoras incorporadas a Tahoe, pero modifica la operación de retransmisión rápida para incluir la recuperación rápida. Este último evita que se vacíe el canal después de la retransmisión rápida, recuperándose más rápidamente de la congestión. En lugar de empezar con arranque lento (como lo hace el emisor TCP Tahoe), el emisor Reno utiliza duplicados ACK's adicionales para agenciar la salida de paquetes.

Reno está optimizado para el caso en que hay pérdida de un solo paquete de una ventana de datos. El emisor Reno transmite a lo sumo un paquete perdido por RTT. Reno puede sufrir problemas de rendimiento cuando múltiples paquetes se pierden de una ventana de datos.

2.3.5.3 TCP NEW RENO

TCP New Reno, evita algunos problemas de rendimiento de Reno. La modificación está en el comportamiento del emisor durante la fase de recuperación rápida, cuando éste recibe un ACK parcial notificando la entrega de algunos, pero no todos los paquetes que estaban a la espera de salir al inicio de la fase de recuperación rápida. En Reno, los ACK's parcial sacan al TCP fuera de la fase de la recuperación rápida reduciendo el tamaño de la ventana de envío al tamaño de la ventana de congestión. En New Reno, los ACK's parcial no sacan al TCP fuera de la fase de recuperación rápida. En cambio, éstos son vistos como un indicador de que el paquete inmediato al paquete notificado en el espacio de los números de secuencia ha sido perdido, y debe ser retransmitido. Por lo tanto, cuando múltiples paquetes son perdidos de una ventana de datos, New Reno se puede recomponer sin un *timeout* de retransmisión, retransmitiendo un paquete perdido por RTT hasta que todos los paquetes perdidos de esa ventana de datos han sido retransmitidos.

New Reno permanece en recuperación rápida hasta que todos los datos pendientes (cuando la fase de recuperación rápida se inició) hayan sido notificados.

2.3.5.4 TCP STACK

TCP Sack (1996), fue propuesto para mejorar la reacción de TCP cuando hay múltiples pérdidas de paquetes de una ventana de envío. Dado que la información de los reconocimientos es limitada, un emisor TCP puede enterarse de solo un paquete perdido por tiempo de ida y vuelta. Si es un emisor agresivo, podría retransmitir paquetes tempranamente, cuando estos ya pueden haber sido exitosamente recibidos. Esta

limitación puede ser sobrellevada utilizando un mecanismo de reconocimiento selectivo y una política de retransmisión selectiva (Sack) implementada en el emisor.

Los algoritmos implementados en TCP Sack son una extensión del control de congestión de Reno, ambos usan el mismo algoritmo para incrementar y decrementar la ventana de congestión. Agregar Sack a TCP no cambia el algoritmo de control de congestión base. La implementación de TCP Sack mantiene las propiedades de Reno de ser robusto ante la presencia de paquetes fuera de orden y retransmite por *timeouts* como último recurso.

La implementación de TCP Sack entra en la fase de recuperación rápida igual que TCP Reno. El emisor retransmite un paquete y reduce la ventana de congestión a la mitad. Durante la recuperación rápida, Sack mantiene una variable llamada *pipe* que representa el número estimado de paquetes pendientes en el camino, esto difiere del mecanismo implementado en Reno. La variable *pipe* es incrementada en uno cuando el emisor transmite un nuevo paquete o retransmite un paquete viejo. *Pipe* es decrementado en uno cuando el emisor recibe un paquete duplicado-ACK con una opción SACK, informando que un nuevo dato ha sido recibido por el receptor.

El emisor mantiene una estructura de datos, que recuerda los reconocimientos de previas opciones SACK. Cuando al emisor se le permite enviar un paquete, este retransmite el siguiente paquete de la lista de paquetes que él infiere que están faltando en el receptor. Si no hay tales paquetes y se informa que la ventana del receptor es suficientemente grande, entonces el emisor transmite un nuevo paquete. Cuando un paquete retransmitido se pierde, la implementación de Sack detecta la pérdida por *timeout*, retransmitiendo el paquete perdido y después empieza la fase de comienzo lento.

El emisor sale de la fase de recuperación rápida cuando un reconocimiento de recuperación es recibido, notificando todos los datos que estaban pendientes cuando se inició la recuperación rápida.

2.3.6 Imparcialidad

Considere N conexiones TCP, cada una con un camino diferente entre extremos, pero todas pasan a través de un enlace cuello de botella con una tasa de transmisión de C bps. (Por enlace cuello de botella entendemos que para cada conexión, todos los demás enlaces a lo largo del camino no están congestionados y tienen suficiente capacidad de transmisión comparados con la capacidad de transmisión del enlace cuello de botella).

Supóngase que sólo atraviesan el enlace cuello de botella conexiones TCP, y que cada conexión está transfiriendo un archivo grande.

Se dice que un mecanismo de control de congestión de TCP es imparcial si la tasa de transmisión media de cada conexión es aproximadamente de N/C; esto es, si cada conexión consigue una porción igual de ancho de banda del enlace.

¿Es TCP imparcial, un algoritmo de incremento aditivo y decremento multiplicativo, para conexiones TCP diferentes dadas que comienzan en diferentes instantes y, por lo tanto, pueden tener diferentes tamaños de ventana en un momento dado?. El incremento aditivo y el decremento multiplicativo del control de congestión de TCP son condiciones necesarias y suficientes para la convergencia a un estado eficiente y justo a pesar del estado inicial de la red. Considérese un caso simple, dos conexiones TCP que comparten un enlace único con una tasa de transmisión C, según se muestra en la **figura 2.10.**

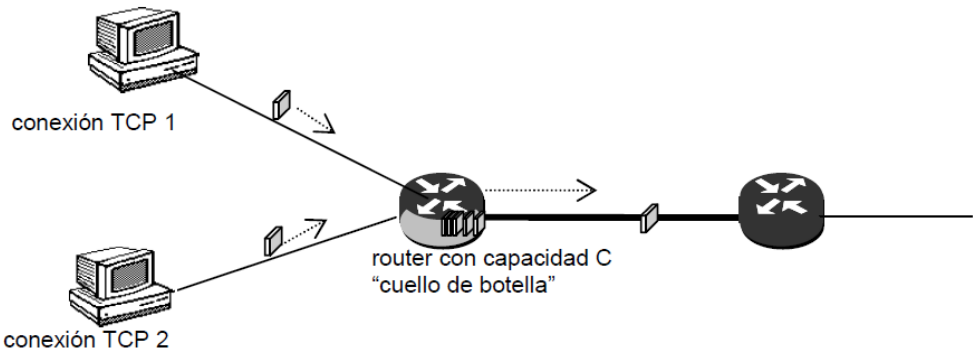


Figura 2.10. Dos conexiones TCP comparten un enlace cuello de botella

Supóngase que las dos conexiones tienen el mismo tamaño de paquete, y RTT (de forma que tienen el mismo tamaño de ventana de congestión y por tanto la misma productividad), que tienen la misma cantidad de datos para enviar, y que ninguna otra conexión atraviesa el enlace compartido. También, se ignora la fase de arranque lento de TCP, y se supone que las conexiones TCP están funcionando continuamente en modo de evitación de la congestión.

En la **figura 2.11** se observa la gráfica de productividad para las dos conexiones TCP. Si TCP comparte el ancho de banda del enlace equitativamente entre las dos conexiones, entonces la productividad debería caer a lo largo de una recta de 45 grados (*reparto*

equitativo del ancho de banda) que arranca en el origen. Idealmente, la suma de las dos productividades debería ser igual a C. Lo bueno sería conseguir que las productividades cayeran hasta algún lugar cerca de la intersección de la línea de reparto equitativo del ancho de banda con la línea de utilización completa del ancho de banda de la **figura 2.11.**

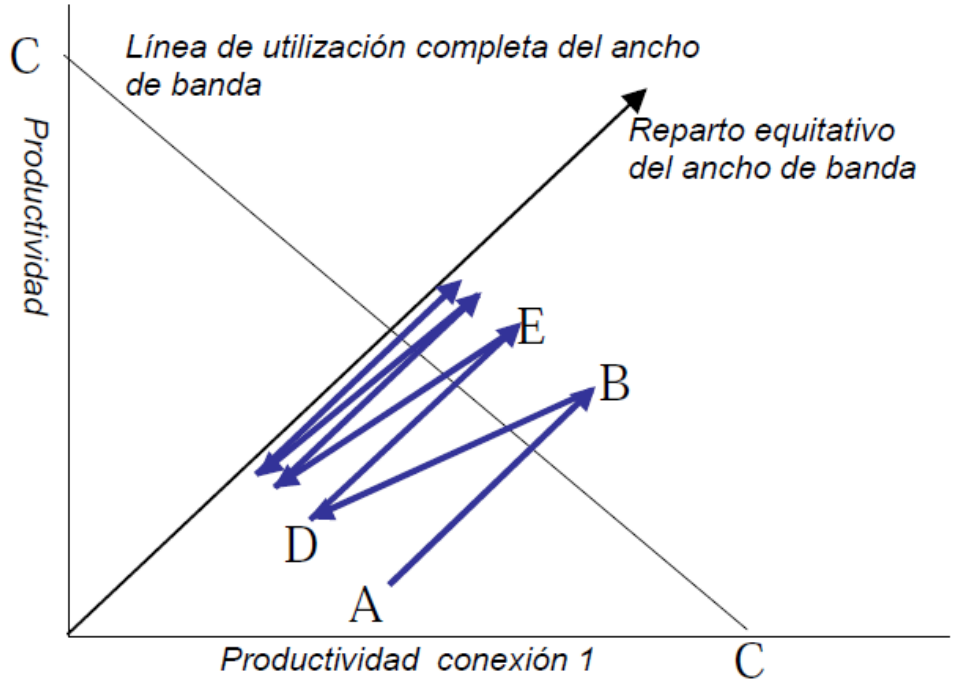


Figura 2.11. Productividad producida por las conexiones TCP 1 y 2

Supóngase que los tamaños de las ventanas TCP son tales que, en un instante dado, las productividades conseguidas por las conexiones 1 y 2 son las indicadas por el punto A, de la **figura 2.11,** como la cantidad de ancho de banda del enlace consumida conjuntamente por las dos conexiones siendo menor que C, no se producirán pérdidas, y ambas conexiones incrementarán su ventana en 1 paquete por RTT. Por tanto, la productividad conjunta de las dos conexiones se desplaza en una línea de 45 grados (incremento igual para ambas conexiones) a partir del punto A.

Eventualmente, el ancho de banda conjuntamente consumido por las dos conexiones será mayor que C, y eventualmente se producirán pérdidas. Supóngase que las conexiones 1 y 2 experimentan pérdidas de paquetes cuando alcanzan productividades marcadas por el punto B. Entonces, las conexiones 1 y 2 decrementan sus ventanas en un factor de dos. Las productividades resultantes obtenidas son las marcadas por el punto

D, a mitad de camino del vector que comienza en el punto B y finaliza en el origen. Como el ancho de banda utilizado conjuntamente es menor que C en el punto D, las dos conexiones incrementan de nuevo sus productividades a lo largo de una línea de 45 grados, comenzando en el punto D. Eventualmente, se producirán pérdidas de nuevo (por ejemplo, en el punto E), y las dos conexiones volverán a decrementar sus tamaños de ventana en un factor de dos, y así sucesivamente. El ancho de banda obtenido por las dos conexiones fluctúa eventualmente a lo largo de la línea de reparto equitativo del ancho de banda.

Estas dos conexiones convergerán en este comportamiento independientemente de dónde se encuentren en el espacio bidimensional. Aunque son varias las suposiciones tras este escenario, aún proporciona una idea intuitiva de por qué TCP produce un reparto equitativo del ancho de banda entre las conexiones.

En este escenario idealizado, se ha supuesto que sólo atraviesan el enlace conexiones TCP, con el mismo tiempo de ida y vuelta, y que sólo una conexión TCP está asociada con un par de *host* origen-destino. En la práctica, normalmente no se cumplen estas condiciones, y las aplicaciones pueden por tanto obtener porciones muy desiguales del ancho de banda del enlace. Se ha demostrado que cuando múltiples conexiones comparten un enlace, aquellas sesiones con menor RTT son capaces de conseguir ancho de banda del enlace más rápidamente según va quedando disponible (abren sus ventanas de congestión más rápidamente), y de esta forma disfrutarán de una mayor productividad que aquellas conexiones con mayores RTT.

2.4 Gestión de las colas de tráfico

El control de congestión de TCP se divide básicamente en dos partes principales. La primera parte controla el tamaño de la ventana de congestión mientras el sistema final intenta encontrar un estado de equilibrio de la red; lo que se conoce como “comienzo lento”. La segunda parte, controla el tamaño de la ventana después de que el sistema final ha alcanzado tal estado, permitiendo mantenerse en él. Se denomina “prevención de congestión”.

El control de TCP puede ser visto como un sistema de control con realimentación. Una complejidad es que la señal de control llega con retraso, esto es, hay un retardo finito entre la detección de la congestión y la recepción de la señal por la fuente de tráfico. El *router* es quién genera dicha señal, que depende del mecanismo de gestión de cola que maneja la longitud, descartando o marcando paquetes cuando es necesario o apropiado.

Drop Tail, mecanismo ejecutado por defecto en Internet, tiene algunas desventajas que se verán a continuación, razones fundamentales de la necesidad de mecanismos activos en el manejo de las colas de tráfico en los *routers* IP.

2.4.1 Drop Tail

La técnica para manejar la longitud de la cola del *router* es fijar una longitud máxima (en términos de paquetes) para cada cola. Acepta paquetes hasta que la longitud máxima sea alcanzada, todos los paquetes que lleguen después serán descartados hasta que disminuya la cola, debido a que un paquete ha dejado la cola para ser transmitido por el enlace. Es sencillo de implementar.

Cierre

En algunas situaciones *Drop Tail* permite a una única conexión ó unas pocas conexiones monopolizar el espacio de la cola, impidiendo que otras conexiones consigan alojarse en ella. Este fenómeno es a menudo el resultado de la sincronización de flujos TCP, en el que varios emisores detectan pérdidas en el mismo periodo de tiempo. Se genera un círculo vicioso que comienza con periodos de utilización relativamente baja de los recursos de la red, seguida por una intensa congestión.

Colas llenas

Drop Tail permite mantener colas llenas (ó casi llenas) durante largos periodos de tiempo. Es importante reducir el tamaño de la cola en estado-estable, y éste es quizá el objetivo más importante del manejo de la cola.

Se puede suponer que la latencia es más importante que el rendimiento. Sin embargo es crucial tener en cuenta que a menudo los paquetes llegan a los *routers* en ráfagas y al encontrarse la cola llena, la ráfaga de llegada de paquetes será motivo de múltiples descartes. Esto puede traer como consecuencia una sincronización de flujos, reduciendo el rendimiento global.

2.4.2 Active Queue Management

La finalidad del almacenamiento temporal en la red es absorber la ráfaga de datos y transmitirlos durante la subsiguiente (esperada) ráfaga de silencio, y esto no sería posible si la cola está llena. Por lo tanto, es importante tener normalmente colas pequeñas.

Los límites de las colas no deberían reflejar el estado estable de la cola que se quiere mantener en la red; en cambio, deberían reflejar el tamaño de ráfaga que se necesita

absorber. Es sabido que el descarte de paquetes es un medio de notificación de la congestión en la red. La solución al problema de "colas llenas" sería descartar paquetes antes que la cola llegue a llenarse, a fin de que los sistemas finales puedan responder a la congestión antes del desbordamiento de los *buffers*. Este enfoque proactivo lo denominan "manejo activo de la cola". El manejo activo de la cola permite a los *routers* controlar cuándo y cuántos paquetes descartar.

AQM es un área activa de investigación y un gran número de algoritmos de manejo activo de las colas del *router*, han sido propuestos para diferentes propósitos. Se describirán fundamentalmente los mecanismos que fueron diseñados con el propósito de estabilizar las colas de tráfico para limitar el retardo de los paquetes. La motivación es proporcionar alguna noción de calidad de servicio con descarte inteligente. Un reto para los algoritmos AQM es que la estabilización de la cola del *router* no debería ser alcanzada a expensas de una disminución del rendimiento del enlace ó incrementando las tasas de pérdida. Seguidamente se verá cómo los algoritmos de manejo activo de las colas de tráfico intentan conseguirlo.

Estos algoritmos pueden notificar a los sistemas finales de la congestión incipiente, realizando un descarte temprano de los paquetes que llegan como una indicación implícita de la congestión ó fijando un bit específico en la cabecera de los paquetes como una indicación explícita de la congestión. Este mecanismo explícito se denomina ECN (*Explicit Congestion Notification,* Notificación de Congestión Explícita), que trabaja en conjunto con AQM, evitando el descarte del paquete como señal de congestión temprana.

A medida que se vaya redactando sobre los mecanismos AQM, si se utiliza la palabra marcado, se hace referencia a descartar el paquete o fijar un bit en la cabecera del mismo; este último caso será posible con ECN. Se explicará al finalizar la descripción de los algoritmos AQM.

A su vez, se describirán varios algoritmos basándose en la clasificación del articulo *"A Survey On Active Queue Management Mechanisms"*, como se muestra en la **Figura 2.12.**

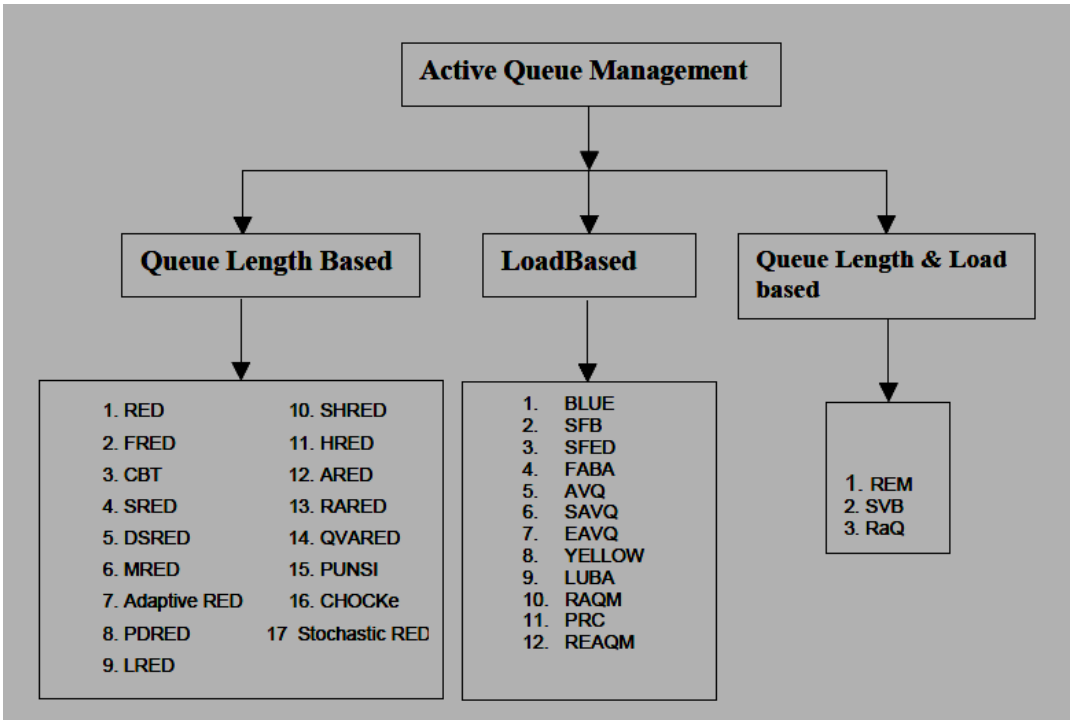


Figura 2.12. Clasificación de los Algoritmos AQM.

2.4.2.1 Algoritmos Basados en la ocupación de cola

Esta clase de algoritmos AQM basan la probabilidad de descarte en observar la ocupación media o instantánea de la cola, con el objetivo principal de estabilizar el tamaño de la cola. RED y sus modificaciones pertenecen a esta categoría. Algunos de ellos sólo tienen una modificación básica del diseño original de RED, como es el caso del algoritmo *Adaptive* RED (ARED). Otros algoritmos de esta categoría como *Flow Random Early Drop* (FRED) y *Stabilized* RED (SRED), intentan ser más equitativos con los distintos flujos al distribuir el ancho de banda disponible.

El principal inconveniente de estos AQM es que crean innecesarios retardos en la cola y *jitter* de encolamiento así como bajo rendimiento de salida.

2.4.2.1.1 RED

RED (*Random Early Detection,* Detección temprana aleatoria); fue presentado en 1993, siendo el primer mecanismo de gestión activa de cola. Su propósito es **evitar la congestión y mantener la red en una región de alto rendimiento y bajo retardo**. RED fue diseñado para trabajar con algoritmos de control de congestión de los sistemas

finales como los de TCP (basados en la ventana de envío). RED controla el tamaño promedio de la cola eligiendo conexiones de forma aleatoria para notificar la congestión.

Las conexiones que usan mayor ancho de banda serán las que tengan una probabilidad más alta de que se descarten sus paquetes. RED intenta mantener el tamaño promedio de la cola pequeño, con el fin de absorber ráfagas de tráfico, permitiendo algunas fluctuaciones en el tamaño actual de la cola. También intenta evitar la sincronización global con el descarte aleatorio de paquetes. La principal desventaja del algoritmo RED es que su rendimiento es muy sensible al ajuste de sus parámetros y una mala configuración degradaría su rendimiento, tal vez con resultados mucho más pobres que con *Drop Tail*.

Algoritmo de RED

RED calcula el tamaño promedio de la cola, *avg*, cada vez que llega un paquete mediante el algoritmo EWMA (*Exponentially Weighted Moving Average,* Media Móvil con Ponderación Exponencial).

Definido por la expresión:

$$avg \leftarrow (1 - \omega_q).avg + \omega_q .q$$

Donde q, avg y ω_q, son el tamaño instantáneo de la cola, el tamaño promedio de la cola y el peso de la función de la media móvil. La motivación de usar EWMA en vez del tamaño instantáneo de la cola es detectar la congestión persistente en el *router*; tolerar la congestión transitoria, permitiendo que el tamaño instantáneo de la cola aumente temporalmente.

El comportamiento del EWMA es el de un filtro paso-bajo, para el que ω_q define la frecuencia de corte. Después de calcular el tamaño promedio de la cola decide si descartar o no el paquete entrante comparándolo con dos umbrales, un umbral mínimo y uno máximo de la siguiente manera:

- Si el tamaño promedio de la cola está por debajo de un umbral mínimo, min_{th}, los paquetes son encolados normalmente (la probabilidad de descarte es nula).
- Por encima del umbral mínimo min_{th}, la probabilidad de descarte de paquete, p_a, aumenta linealmente hasta llegar a la probabilidad max_p, que se alcanza cuando se llega a un umbral de ocupación máxima max_{th}.

- Por encima de max_{th}, se descartan todos los paquetes.

Las tres fases anteriormente comentadas se suelen denominar fase normal, fase de prevención de congestión y fase de control de congestión. La **figura 2.13** muestra la función de probabilidad de descarte de RED.

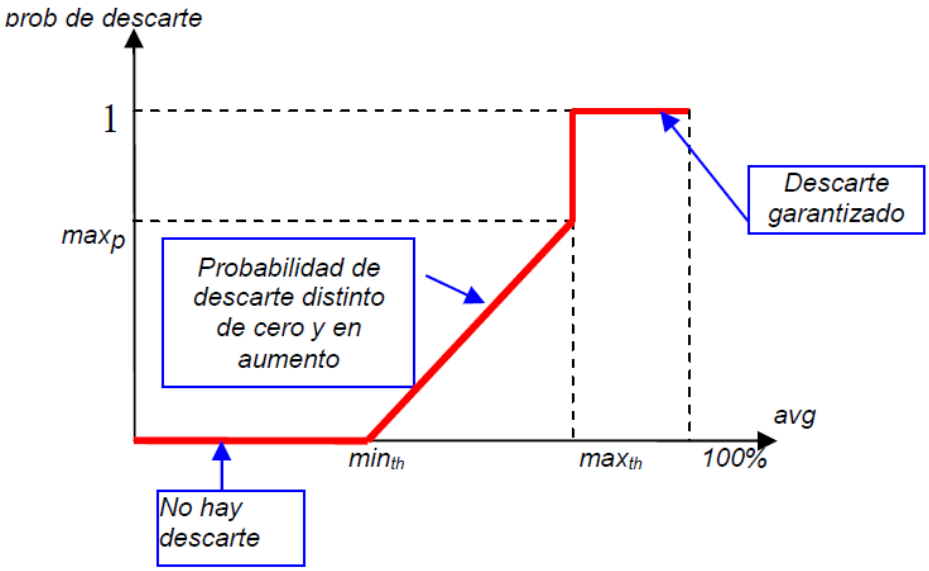


Figura 2.13. Probabilidad de descarte de RED

El algoritmo general de RED se describe en la **tabla 2.1.**

Algoritmo general RED
Para cada paquete que llega Calcular el tamaño promedio de la cola avg if ($min_{th} \leq avg < max_{th}$) calcular la probabilidad p_a con probabilidad p_a: marcar el paquete; else if ($max_{th} \leq avg$) marcar el paquete;

Tabla 2.1 Algoritmo general RED.

Parámetros de RED

El rendimiento de AQM basado en RED depende del ajuste de sus cuatro parámetros que son: el umbral mínimo min_{th}, el umbral máximo max_{th}, la probabilidad máxima max_p y el peso de la cola ω_q. Estos parámetros se definen en la **tabla 2.2.**

El análisis de RED ha generado varios documentos interesantes. El ajuste de sus parámetros ha sido una ciencia inexacta hasta ahora, de tal modo que algunos investigadores han abogado en contra de su uso como M. May et al, que llevaron a cabo un estudio de RED y los principales resultados que obtuvieron fueron: primero, RED con *buffers* pequeños no mejora significativamente el rendimiento de la red, en particular el rendimiento global es menor que con *Drop Tail* y la diferencia en el retardo no es significativa. Segundo, la dificultad de su ajuste. Sin embargo, también observaron que RED sí permite controlar el tamaño de la cola con grandes *buffers*.

Aunque supera las flaquezas de *Drop Tail*, sintonizar RED para que trabaje adecuadamente bajo diferentes escenarios de congestión, es difícil.

Para una operación adecuada bajo condiciones de la red, se requiere un ajuste de los cuatro parámetros coordinado con los parámetros del escenario de la red, el número de conexiones de TCP (N), el tiempo de ida y vuelta (R) y la capacidad del enlace "cuello de botella" (C). Se verá esta relación más adelante.

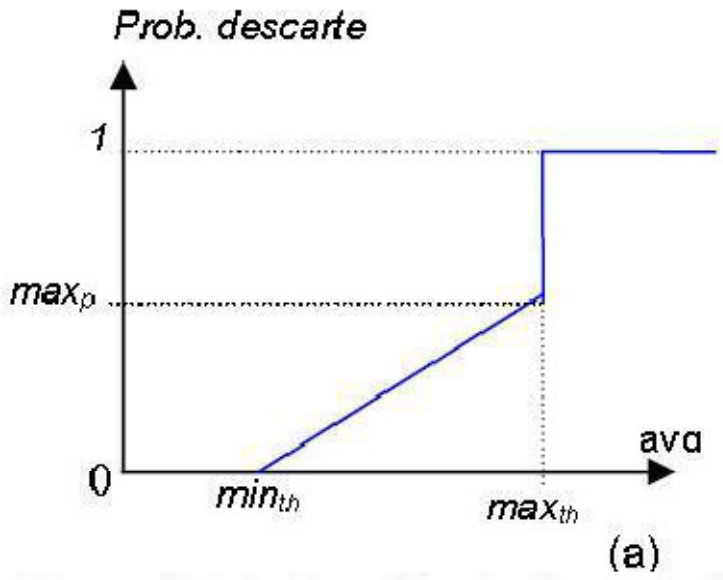
Parámetro	Símbolo	Rol que juega en el algoritmo RED
Peso de la cola	ω_q	Coeficiente del filtro paso-bajo para evaluar el tamaño promedio de la cola. Relaciona la ocupación promedio de la cola con la ocupación instantánea.
Probabilidad máxima de descarte	max_p	Determina la agresividad de marcado de paquete.
Umbral mínimo	min_{th}	Umbral hasta donde el tamaño promedio de la cola puede incrementar antes de comenzar a marcar paquetes.
Umbral máximo	max_{th}	Umbral que el tamaño promedio de la cola puede alcanzar antes de marcar todos los paquetes entrantes a la cola.

Tabla 2.2 Descripción de los parámetros de RED

Gentle RED

Se observó un comportamiento oscilatorio en la cola del *router* RED debido a la discontinuidad en la probabilidad de descarte (el salto de max_p a 1 en max_{th}). Esto es debido a que todos los paquetes serán descartados cuando el tamaño promedio de la cola, *avg*, exceda el umbral máximo, max_{th}.

Para resolver este problema, se propuso Gentle RED, definido como RED con "modo suavizado", es una variante de RED que sencillamente modifica la función de descarte de RED, para el caso en que el tamaño promedio de la cola exceda el umbral máximo, como se muestra en la **figura 2.14.** La probabilidad de descarte aumenta linealmente desde la probabilidad máxima de descarte max_p a uno, cuando el tamaño promedio de la cola varía desde el umbral máximo max_{th}, hasta $2*max_{th}$. Con una pendiente de $(1-max_p)/max_{th}$. Evitando el cambio brusco de max_p a uno. El parámetro añadido es "gentle_", únicamente se ajusta a uno.



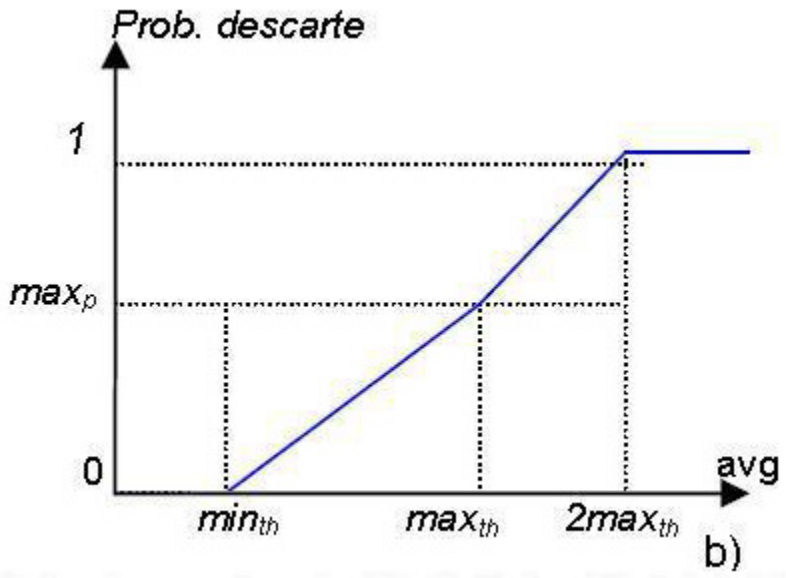


Figura 2.14. Función de probabilidad de descarte de RED (a) y GRED (b).

Adaptive RED

Un punto débil de RED es que no tiene en cuenta el número de flujos que comparten un enlace "cuello de botella". Recuérdese que TCP reduce su tasa de envío a la mitad cuando ha detectado un descarte. Si el ancho de banda de un enlace "cuello de botella" es igualmente compartido por '*n*' flujos TCP (cada uno recibe *1/n* del ancho de banda del enlace), un único paquete descartado causa que un flujo reduzca su tasa de transmisión a $0.5n^{-1}$ y reduzca la carga ofertada por un factor de $(1-0.5n^{-1})$. Por lo tanto si '*n*' aumenta, el impacto de las notificaciones de congestión individual, disminuye. En esta situación, si el algoritmo RED no es configurado para ser más agresivo, la cola RED puede degenerarse a una simple cola *Drop Tail*. Por otro lado, si '*n*' es pequeño el impacto de las notificaciones de congestión individual son grandes. En esta condición, si el algoritmo de RED es agresivo, puede ocurrir una infrautilización del enlace, si muchas conexiones disminuyen su tasa de envío en respuesta a la congestión observada. Una clave del éxito de un mecanismo de manejo activo, es encontrar un equilibrio efectivo entre reducir la tasa de pérdida y evitar una infrautilización del enlace, ajustando apropiadamente la tasa de notificación de congestión. Como también no permitir un gran incremento en la latencia.

Feng *et al.*, propusieron un algoritmo de auto-configuración de RED ajustando max_p; para adaptarse a los cambios de la carga de tráfico, conocido como **Adaptive RED** (*Adaptive Random Early Dectection*). Esta propuesta mantiene la estructura básica de RED y simplemente ajusta el parámetro max_p para mantener el tamaño promedio de la cola entre el umbral mínimo min_{yh} y el max_{th}.

La idea detrás de este algoritmo es deducir si RED debería ser más o menos agresivo examinando las variaciones de la longitud promedio de la cola. La probabilidad máxima de descarte, max_p, se ajusta cada vez que la longitud promedio de la cola cae fuera del rango objetivo entre el min_{th} *y* max_{th}. Cuando la longitud promedio de la cola es más pequeña que el umbral mínimo min_{th}, la probabilidad máxima max_p, se reduce multiplicativamente para disminuir la agresividad de RED; cuando la longitud promedio de la cola es mayor que el umbral máximo, max_{th}, la probabilidad máxima, max_p, se incrementa multiplicativamente. Se describe el algoritmo de auto-configuración de RED en la siguiente **tabla 2.3,** donde max_p es aumentado o disminuido multiplicativamente por factores arbitrarios de $\alpha = 3$ *y* $\beta = 2$, respectivamente.

Algoritmo de auto-configuración de Feng et al

En cada actualización del avg:
if (min_{th} < avg < max_{th})
status ← between;
max_p ← 0;
if (avg < min_{th} && status != Below)
status ← Below;
max_p ← max_p/α;
if (avg > max_{th} && status != Above)
status ← Above;
max_p ← max_p*β

Tabla 2.3. Algoritmo de auto-configuración de Feng et al.

Floyd et al., mejoran el algoritmo original "Adaptive RED". La estructura básica del algoritmo la mantienen intacta, es decir, la auto-configuración de RED ajustando el parámetro max_p. Pero en vez de aplicar un incremento y decremento multiplicativo, utilizan la política de incremento aditivo y decremento multiplicativo del parámetro max_p; para mejorar el control del tamaño promedio de la cola. También proporcionan guías para elegir el umbral mínimo $mint_{th}$, el umbral máximo max_{th} y el coeficiente del filtro paso bajo para calcular el tamaño promedio de la cola teniendo en cuenta la capacidad del enlace. El algoritmo Adaptive RED propuesto por *Floyd et al.,* se describe en la siguiente **tabla 2.4.** Este algoritmo es conocido como ARED, también incluye el

"modo suavizado" que se comentó líneas arriba. Los parámetros del algoritmo ARED se muestran en la **tabla 2.5.**

Algoritmo de auto-configuración de Floyd et al

En cada intervalo (0.5 segundos) actualizar max_p:
if (avg > min_{th} + 0.6 (max_{th} - min_{th}) && $max_p \leq 0.5$)
incrementar max_p:
$max_p \leftarrow max_p + \min(0.01, max_p/4)$
else if (avg < min_{th} + 0.4(max_{th} - min_{th}) && $max_p \geq 0.01$)
decrementar max_p:
$max_p \leftarrow max_p * \beta$; // $\beta = 0.9$

Tabla 2.4. Algoritmo de auto-configuración por Floyd et al.

Símbolo	Descripción
ω_q	Coeficiente de filtro paso-bajo para calcular el promedio de la cola.
β	Factor de decremento para adaptar max_p
min_{th}	Umbral mínimo del tamaño promedio de la cola
max_{th}	Umbral máximo del tamaño promedio de la cola

Tabla 2.5. Parámetros de ARED

Una debilidad principal en el algoritmo original RED es que no controla de forma eficaz y previsible la longitud promedio de la cola, si el parámetro max_p, es elevado o la congestión en el enlace es ligera, RED mantiene el tamaño promedio de la cola cerca del umbral mínimo, min_{th}. Por otro lado, si max_p es bajo o el enlace está altamente congestionado, el tamaño promedio de la cola tiende a crecer y superar el umbral máximo.

2.4.2.1.2 FRED

El algoritmo FRED (*Flow Random Early Drop*, Descarte Aleatorio Temprano de Flujos) es una versión modificada de RED, fue propuesto para paliar el efecto de falta de equidad del esquema RED. Este algoritmo protege flujos frágiles de otros flujos más agresivos (que tienden a monopolizar la cola y consumir todo el ancho de banda disponible) imponiendo una probabilidad de descarte que depende de la ocupación de la cola de cada uno de los flujos activos. El algoritmo intenta proporcionar igual ancho de banda manteniendo sólo información de los flujos que tengan paquetes en la cola.

FRED trabaja de forma similar a RED, sólo difiere en algunos aspectos, introduce cinco nuevos parámetros para cada flujo: Min_f, Max_f, $Qavg_f$, qf_{len}, *strike*. Estos parámetros identifican la ocupación máxima y mínima en la zona de notificación probabilística, la ocupación media de paquetes por flujo, la cantidad instantánea de paquetes del flujo en la cola y el número de ocasiones en que el flujo no responde a la notificación de congestión, respectivamente. FRED calcula *Qavg* por cada paquete recibido si la cola está vacía. En el caso en que esté ocupada, se hace el cálculo hasta que el paquete sea admitido. Además, por cada paquete que sale de la cola se calcula *Qavg* y $Qavg_f$.

Los parámetros del algoritmo FRED se pueden observar en la **tabla 2.6.**

Símbolo	Descripción
Min_f	Ocupación mínima en la zona de notificación probabilística
Max_f	Ocupación máxima en la zona de notificación probabilística
$Qavg_f$	Ocupación media de paquetes por flujo
Qf_{len}	Cantidad instantánea de paquetes del flujo en cola
Strike	Número de ocasiones en que el flujo no responde a la notificación de congestión

Tabla 2.6. Parámetros de FRED

Este algoritmo penaliza un paquete si cree que éste pertenece a un flujo agresivo. Se considera un flujo agresivo si:

$$Qf_{len} \geq Max_f \;||\; (Qavg \geq Max_{th} \;\&\&\; qf_{len} > 2Qavg_f) \;||\; (qf_{len} \geq Qavg_f \;\&\&\; strike > 1)$$

Este esquema acepta preferentemente paquetes de flujos con pocos paquetes en la cola, permitiendo un valor Min_f de paquetes a cada conexión sin pérdidas. De ahí que los flujos considerados frágiles serán descartados de forma probabilística sólo cuando

$$qf_{len} \geq MAX\ (Min_f, Qavg_f)$$

2.4.2.1.3. SRED

El algoritmo SRED (*Stabilized Random Early Drop,* Descarte Aleatorio Temprano Estabilizado) intenta conseguir los objetivos de AQM, tales como alto rendimiento y bajo retardo de encolado, estabilizando la longitud de la cola en torno a un umbral de la

misma. SRED estabiliza la cola del *router* descartando los paquetes que llegan con una probabilidad de descarte dependiente de la carga de tráfico en la red.

La idea básica detrás del algoritmo SRED es reducir las fluctuaciones de la cola, es decir estabilizar la ocupación de la cola a un nivel dado, independientemente del número de flujos activos TCP. Este algoritmo difiere de RED en el sentido en que no computa la ocupación media de la cola, por lo que la probabilidad de descarte depende solamente de la ocupación instantánea y del número estimado de flujos activos. Principalmente, el algoritmo identifica qué flujos TCP están tomando más ancho de banda de la porción que les corresponde y los penaliza para asignar así de forma más equitativa el ancho de banda entre todos los flujos.

Si la carga de la red es pesada y llegan más paquetes de los que el *router* pueda entregar, SRED aumenta la probabilidad de descarte estabilizando la cola del *router*. Por otro lado, SRED reduce la probabilidad de descarte si la carga de la red es ligera. También mide la carga de la red monitorizando la cola instantánea del *router* y estimando el número de conexiones activas sobre el enlace.

Tiene la misma forma de actuar que los algoritmos propuestos por *Feng et al.,* y *Floyd et al.* Sin embargo, a diferencia de estos mecanismos, SRED intenta calcular el número de flujos activos y usa esta estimación para ajustar la probabilidad de descarte.

2.4.2.2 AQM basados en medida de tasas

Alternativamente a utilizar la información de la ocupación media o instantánea de la cola, otros algoritmos AQM detectan o previenen la congestión basándose estratégicamente en la tasa de entrada del tráfico o en la tasa de pérdidas de paquetes. Esta clase de algoritmos mantiene una única probabilidad que es utilizada para descartar el paquete cuando éste es encolado. Los algoritmos AQM basados en tasas son más rápidos para reaccionar a la congestión por lo que intentan reducir la diferencia de la tasa entre encolamiento y desencolamiento, logra una baja tasa de pérdidas, bajo retardo y mayor utilización del enlace. GREEN y BLUE son algunos de los algoritmos AQM más representativos que pertenecen a esta clase.

2.4.2.2.1 GREEN

GREEN es un algoritmo que elimina las tendencias en contra de conexiones TCP con RTT's más largos, resultando un alto grado de equidad y manteniendo, al mismo tiempo, una alta utilización del enlace, una baja pérdida de paquetes y unos tamaños de cola pequeños. GREEN aplica el conocimiento del comportamiento en estado estacionario de

conexiones TCP para descartar paquetes de forma activa, evitando así que flujos TCP de larga duración induzcan a congestión. También consigue que flujos cortos de RTT no estén reñidos con su parte correspondiente de ancho de banda. En consecuencia, GREEN asegura la equidad entre flujos mucho más que otros sistemas de gestión de colas.

La idea detrás del algoritmo GREEN es mantener una tasa de pérdidas baja y minimizar los retardos de encolamiento. Para este fin, el algoritmo ajusta la probabilidad de descarte de acuerdo a una tasa estimada de entrada del tráfico denominada X_{est} y en base a la capacidad esperada del enlace, C_e.

La tasa de entrada es obtenida mediante:

$$X_{est} = (1 - \exp(-Del / K))*(B / Del) + \exp(-Del / K)*X_{est}$$

donde *Del* es el retardo entre paquetes, *B* el tamaño del paquete y la variable *K* una constante de tiempo. La capacidad de *C* es asignada con un valor menor que la capacidad del enlace real *C*.

De tal manera, GREEN ajusta la tasa de notificación de la congestión incrementándola o reduciéndola en función de la capacidad esperada del enlace y la tasa estimada de entrada del tráfico. Es decir, si X_{est} es superior a C_e la tasa de notificación de la congestión P_d es incrementada por ΔP_e a una tasa de $1/\Delta T$. Inversamente, si X_{est} es inferior a C_e, P_d es reducida por ΔP_d a una tasa de $1/\Delta T$. Por lo que, el algoritmo GREEN, descarta o marca de forma probabilística el paquete que va llegando a una tasa:

$$P_d = P_d + \Delta P_d - U(X_{est} - C_e)$$

donde, $U(x) = +1$ *cuando* $x \geq 0$ y $U(x) = -1$ *cuando* $x < 0$.

2.4.2.2.2 BLUE

El algoritmo BLUE intenta conseguir los mismos objetivos que RED, una alta utilización del enlace y minimizar tanto el retardo como la tasa de pérdida de paquetes. BLUE alcanza estos objetivos de rendimiento adaptando su tasa de descarte basada en la pérdida de paquetes y la utilización del enlace, en lugar de la longitud instantánea o promedio de la cola. Usa una probabilidad de descarte para los paquetes que llegan a la cola. Intenta adaptar dicha probabilidad para proporcionar una realimentación apropiada a los sistemas finales. Si la cola se desborda constantemente, aumenta la probabilidad de descarte para descartar paquetes con mayor agresividad. Si la cola llega a estar vacía, disminuye la probabilidad de descarte.

BLUE realiza la gestión de colas basada en la pérdida de paquetes y la utilización del enlace. Mantiene una probabilidad de descarte, p_m, para marcar o descartar paquetes. Si la cola está continuamente descartando paquetes, p_m se incrementa en un factor $\delta 1$. Si la cola está vacía o el enlace está libre, p_m se disminuye en un factor $\delta 2$. El valor de $\delta 1$ se debe establecer estrictamente mayor que $\delta 2$. Esto se debe a que el enlace es subutilizado cuando la gestión de la congestión es demasiado agresiva o demasiado conservadora, pero la pérdida de paquetes se produce sólo cuando el mecanismo de congestión es demasiado conservador. BLUE utiliza un parámetro más, *freeze_time*, el cual determina el intervalo de tiempo entre dos actualizaciones sucesivas de *freeze_time*. Permite que los cambios en la probabilidad de descarte tengan efecto antes de que el valor se actualice de nuevo.

El algoritmo BLUE se puede ver en la **tabla 2.7.**

Algoritmo BLUE
En caso de pérdida de paquetes (or $Q_{len} > L$): if ((now – last_update) > freeze_time) $p_m \leftarrow p_m + \delta_1$ last_update ← now En caso de enlace libre: if ((now – last_update) > freeze_time) $p_m \leftarrow p_m - \delta_2$ last_update ← now

Tabla 2.7. Algoritmo BLUE.

La probabilidad de descarte, p_m, también se actualiza cuando la longitud de la cola excede un cierto valor, para dejar espacio para ráfagas transitorias y para controlar el retardo en la cola cuando el tamaño del *buffer* que se utiliza es de gran tamaño.

2.4.3.2 AQM basados en ocupación de la cola y medida de tasas.

Combinando los métodos anteriores se identifica un nuevo tipo de esquemas AQM. Estos esquemas monitorizan la ocupación instantánea de la cola en adición a la tasa de entrada del tráfico, la medida de tasa de pérdida o la medida de retardos, para determinar la probabilidad de descarte del paquete. Uno de los ejemplos más representativos de esta clase de esquemas AQM es *Random Exponential Marking* (REM).

2.4.2.3.1 REM

REM (*Random Exponential Marking,* Marcado Aleatorio Exponencial), es un algoritmo cuyo propósito es conseguir una alta utilización de la capacidad del enlace, bajo retardo y

pérdida de paquetes. Difiere de RED en dos cuestiones de diseño; REM utiliza una definición diferente de la medida de congestión y una función de probabilidad de descarte diferente. Utiliza una probabilidad exponencial para ajustar el tamaño de la cola y minimizar el retardo de encolamiento, definiendo primeramente una función que se basa en la información de la cola y en la tasa de llegada de paquetes.

Este algoritmo puede estabilizar la tasa de entrada en torno a la capacidad del enlace y el tamaño de la cola en torno a un pequeño "valor deseado", independientemente del número de usuarios que estén compartiendo el enlace.

2.4.2.4 Algoritmos basados en control de procesos

2.4.2.4.1 El lazo de control

Un sistema de control está definido como un conjunto de componentes que mantiene un resultado o una variable en un valor deseado. La forma simplificada para expresar un sistema de control es la que se muestra en la **figura 2.15.**

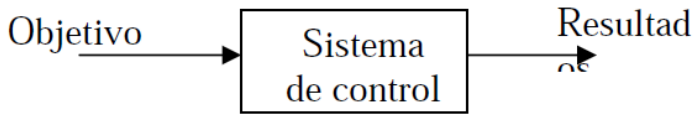


Figura 2.15. Planteamiento básico de un sistema de control.

- Objetivos de control: entrada o señales actuantes.
- Componentes del sistema de control.
- Resultados o salidas: salidas o variables controladas.

En general, el objetivo de un sistema de control es regular las salidas, siguiendo lo establecido previamente por las entradas a través de los elementos del sistema de control.

Un sistema de control en lazo abierto o sistema no realimentado. No tiene la realimentación de la salida hacia la entrada, por lo tanto el valor de la salida no tiene ningún efecto en cómo se calcula la señal de control.

Un sistema de control en lazo cerrado o sistema realimentado (**figura 2.16**) es un sistema en el que se realimenta la salida y se compara con el valor deseado de la misma.

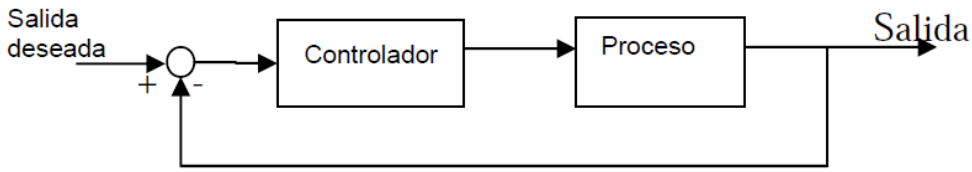


Figura 2.16. Sistema de control en lazo cerrado.

Cuando hablamos de realimentación usamos la palabra inglesa *feedback*.

Los sistemas de control realimentados se pueden clasificar de diversas formas. Así se puede hablar de sistemas lineales y sistemas no lineales, sistemas invariantes en el tiempo y sistemas variantes en el tiempo.

Cuando hablamos de sistemas lineales nos referimos a aquellos cuyos modelos son un conjunto de ecuaciones lineales.

Una ventaja de los sistemas de control en lazo cerrado es que la respuesta del sistema es relativamente insensible a las perturbaciones externas y a las variaciones internas de los parámetros del sistema.

Una situación en la que la realimentación puede dar no demasiados buenos resultados es cuando el sistema a controlar tiene un retraso muy grande, es decir, que desde que aplicamos la señal de control hasta que observamos cambios en la salida transcurre un tiempo muy grande.

Este es el caso que interesa, debido a que el control de congestión se puede ver como un sistema en el que la realimentación se produce a través de los descartes de los paquetes en las colas. Así, y teniendo en cuenta la existencia de un modelo matemático que refleja el comportamiento de TCP, se puede abordar el control de la cola de los *routers* como un problema clásico de control.

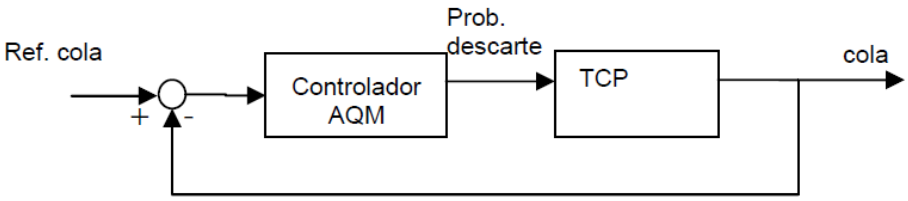


Figura 2.17. Sistema de control-Probabilidad de descarte.

2.4.2.4.2 Linealización del modelo TCP

El modelo TCP

Como punto de partida, se considera un modelo dinámico no lineal para describir el comportamiento de una red TCP/AQM, el cual ignora el mecanismo de *time-out*. El modelo matemático relaciona las variables de N fuentes homogéneas controladas por TCP y las de un *router* congestionado. Se describe por las siguientes ecuaciones diferenciales con retardo:

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - \frac{W(t) \cdot W\big(t - R(t)\big)}{2R\big(t - R(t)\big)} \cdot p\big(t - R(t)\big)$$

$$\frac{dq(t)}{dt} = \frac{N(t)}{R(t)} \cdot W(t) - C$$

donde:

$$R(t) = \frac{q(t)}{C} + T_p$$

W = tamaño de ventana TCP (paquetes)
q = longitud de la cola congestionada (paquetes)
R = tiempo de ida y vuelta (segundos)
C = capacidad del enlace (paquetes/segundo)
N = factor de carga (número de conexiones TCP)
p = probabilidad de marcado de paquete
T_p = retardo de propagación (segundos)

La primera ecuación diferencial describe la dinámica de control de ventana de TCP, es decir, el primer término indica que el tamaño de ventana será incrementado en uno cada tiempo de ida y vuelta (algoritmo de prevención de congestión); el segundo término indica que el tamaño de ventana se divide a la mitad en respuesta al marcado de paquete (p), comportamiento visto en Reno y sus variantes. Como es sabido, el emisor TCP es el encargado de la dinámica de ventana, mientras que los esquemas AQM son estrategias de control del *router*. Por lo tanto, después de un cierto tiempo de retardo, el emisor refleja en la dinámica de ventana el evento de marcado (pérdida o fijar un bit en la cabecera) de paquete; se puede ver en la primera ecuación donde la función *p(.)* viene retrasada.

La segunda ecuación diferencial describe la longitud de la cola (cuello de botella), como la diferencia entre la tasa de llegada de paquetes de las N conexiones TCP (*N(t)W(t)/R(t)*) y la capacidad de transmisión del enlace (C).

Las ecuaciones diferenciales se muestran en el diagrama de bloques de la **figura 2.18** donde se observa el control de ventana TCP, la carga y la dinámica de la cola.

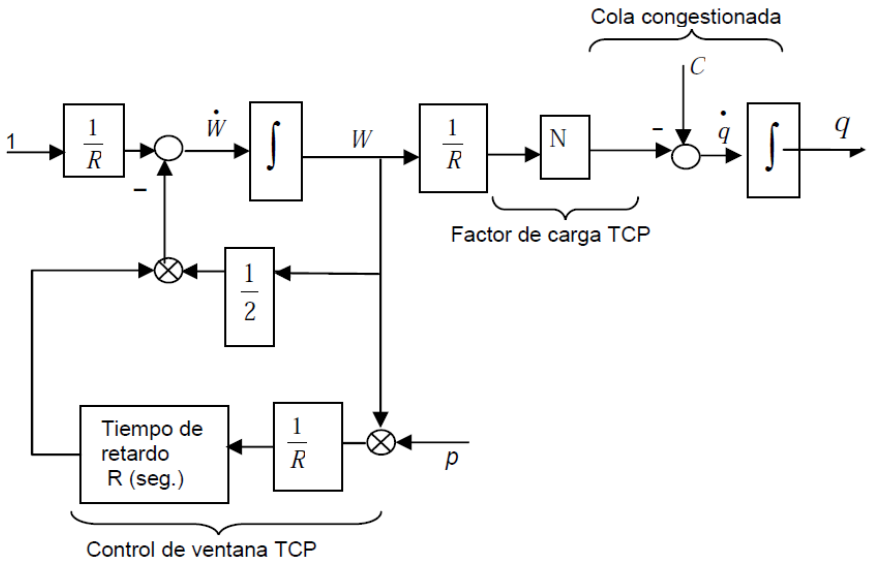


Figura 2.18. Diagrama de bloques de una conexión TCP.

Linealización

Para comprender mejor su comportamiento y control realimentado, se realiza la linealización en torno a un punto de operación. Con el modelo linealizado de pequeña señal, se pueden aplicar las técnicas de análisis y control de sistemas lineales.

La idea básica consiste en escribir ecuaciones para pequeñas perturbaciones en torno a un punto de equilibrio e ignorar términos de segundo orden, obteniendo así el modelo lineal, descrito como sigue:

Tomando (W, q) como el estado del sistema y p como la entrada; el punto de equilibro (W_0, q_0, p_0) del sistema (1) es definido por:

$$\dot{W} = 0 \rightarrow W_0^2 p_0 = 2; \quad \dot{q} = 0 \rightarrow W_0 = \frac{R_0 C}{N}$$

donde:

$$R_0 = \frac{q_0}{C} + T_p$$

Linealizando (1) entorno a su estado de equilibrio se obtiene:

$$\delta\dot{W}(t) = -\frac{2N}{R_0^2 C}\delta W(t) - \frac{R_0 C^2}{2N^2}\delta p(t - R_0)$$

$$\delta\dot{q}(t) = \frac{N}{R_0}\delta W(t) - \frac{1}{R_0}\delta q(t)$$

donde:

$$\delta W = W - W_0$$

$$\delta q = q - q_0$$

$$\delta p = p - p_0$$

Representan las perturbaciones en W, q, y p de sus valores de equilibrio. Los valores propios del TCP linealizado y de la dinámica de la cola (3) son respectivamente:

$$-\frac{2N}{R_0^2 C} \quad \left(o\ \frac{-2}{W_0 R_0}\right) \quad y\ -\frac{1}{R_0}$$

Ya que todos los parámetros de la red (*N, R, C*) son cantidades positivas, el valor negativo de estos valores indica que el estado de equilibrio es local y asintóticamente estable. Esto es, para p = p_0, todas las respuestas que comienzan "cerca" de (W_0, q_0) van a converger asintóticamente hacia (W_0, q_0). Una interpretación de la constante de tiempo de TCP $W_0R_0/2$, se obtiene expresando la linealización de la ecuación δW anterior como:

$$\delta\dot{W}(t) = -\lambda_0\delta W(t) - \frac{R_0 C^2}{2N^2} \cdot \delta p(t - R_0)$$

donde λ es la tasa de pérdida de paquete. Por lo tanto, la constante de tiempo es 1/ λ_0. En estado estable, la reducción del tamaño de la ventana $\frac{1}{2}W_0\lambda_0$ debe equilibrar el incremento del tamaño de ventana en 1/R_0. Por lo tanto $\lambda_0 = \frac{2}{w_0 R_0}$.

La linealización de la ecuación de la cola representa un retraso de primer orden con una constante de tiempo de R_0.

Para pasar del sistema del domino temporal al dominio de la frecuencia, se aplica la Transformada de Laplace, obteniendo las siguientes ecuaciones:

$$P_{tcp}(s) = -\frac{\frac{R_0 C^2}{2N^2}}{s + \frac{2N}{R_0^2 C}} e^{-sR_0} \qquad P_{queue}(s) = \frac{\frac{N}{R_0}}{s + \frac{1}{R_0}}$$

P_{tcp} denota la función de transferencia de TCP, P_{queue} la función de transferencia de la cola. El término e^{-sR} es la transformada del retardo en la planta. El sistema se representa gráficamente por el diagrama de bloques de la **figura 2.19.**

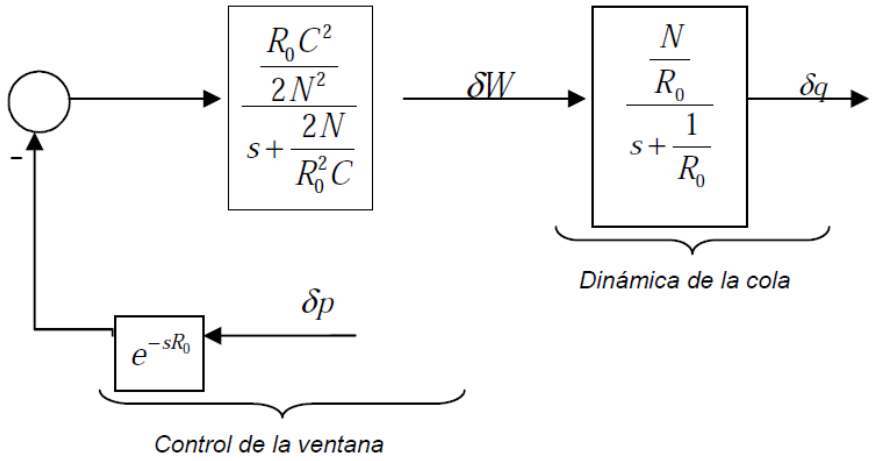


Figura 2.19. Diagrama de bloques del sistema lineal.

Se tiene:

$$P(s) = P_{tcp}(s) \cdot P_{queue}(s) = \frac{\left(\frac{C^2}{2N}\right) e^{-sR_0}}{\left(s + \frac{2N}{R_0^2 C}\right)\left(s + \frac{1}{R_0}\right)}$$

Donde P(s) es la función de transferencia de la planta o sistema TCP/AQM que se pretende controlar.

2.4.2.4.3 PID (P-PI-PID) – Control clásico

El algoritmo PID regula la longitud de la cola en torno a un valor objetivo conocido como "referencia" (q_{ref}). Las siglas PID significan Proporcional – Integral – Derivativo, estos tres nombres se refieren a las tres acciones que suma este controlador, es decir la salida de este controlador será proporcional a su entrada, proporcional a la integral de su entrada y proporcional a la derivada de su entrada. La entrada normalmente será el error, es decir la diferencia entre la referencia buscada y la salida real de la planta. El peso que cada una de las tres acciones ejerce sobre el resultado final depende de tres parámetros, que serán los objetivos de la sintonización de este controlador. En su versión continua, la forma de calcular la señal de control del PID es la siguiente:

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int e(t)dt \; + T_d \frac{de}{dt} \right)$$

Debido a que en este caso se trabaja con variables discretas, resultado del muestreo de la señal cada cierto periodo definido, se hace necesario utilizar una versión discreta de esta fórmula.

A continuación se ofrece un resumen de las tres acciones de este controlador y el significado real que tienen:

- ***Acción proporcional (P)*:** La señal de control debe tener un valor proporcional al de la señal de error, y cuanto más grande es la K_p, menor es el error en régimen permanente y más rápido actúa el regulador ante los cambios bruscos de la señal consigna.

- ***Acción integral (I)*:** Si sólo interviniese en el control la acción proporcional, aparece siempre un error permanente entre el valor consigna y el real de la planta. Para evitar este problema, se hace que la señal de control tenga una componente proporcional a la integral del error en el tiempo que compensa en cada instante el error acumulado.

- ***Acción derivativa (D)*:** En algunas ocasiones, la variable que se quiere controlar tiene mucha inercia. En este caso, es necesario considerar una componente proporcional a la derivada de la señal de error respecto al tiempo. La acción tiene

en cuenta la tendencia de la señal a controlar: actúa para compensar posibles errores que sólo están comenzando a aparecer.

PID es un mecanismo más robusto que RED, dado que evita comportamientos oscilatorios en la cola y responde mucho más rápido a cambios repentinos en el nivel de carga. Más adelante se realizará el estudio de este mecanismo.

2.4.2.5 ECN

En líneas anteriores se mencionaron dos maneras de informar a los sistemas finales de la congestión en la red. En este punto, se pasa a describir la forma de realizar la notificación de forma explícita, marcando el paquete, es decir, fijando un bit en su cabecera en vez de descartarlo, dado que el descarte implica una disminución en el rendimiento y un aumento en la latencia.

El IETF propuso un protocolo para el mecanismo de señalización explícita llamado **ECN** (*Explicit Congestion Notification*, Notificación de Congestión Explícita) usando bits en las cabeceras de IP y TCP, como se puede apreciar en la **figura 2.20.**

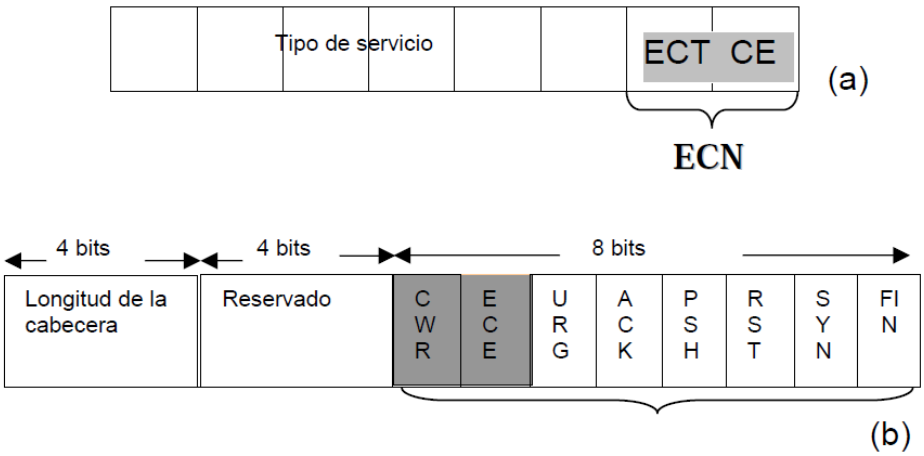


Figura 2.20. Campo ECN en la cabecera IP (a). Campo ECN en la cabecera TCP (b).

Un campo utilizado en la cabecera IP denominado ECN que está compuesto a su vez por dos *flags* ECT y CE. Los extremos de la red indican su capacidad ECN con los códigos “01” y “10”. El código “00” significa que el *host* no soporta ECN y el código “11” indica que el *host* soporta ECN y el *router* ha marcado congestión.

Cuando el receptor recibe el paquete de datos con el código “11”, fija el *flag* ECE en la cabecera TCP del siguiente paquete ACK a enviar. El emisor al recibir dicho ACK reduce su

ventana de congestión (como en el caso de descarte), fija el *flag* CWR de su siguiente paquete a transmitir para confirmar la recepción del ACK con el *flag* ECE fijado.

Dado que un usuario poco colaborador puede fijar el paquete e ignorar la señal de congestión del *router*, si el tamaño promedio de la cola excede un cierto umbral, la especificación estándar de ECN recomienda que los *routers* descarten los paquetes en vez de fijar el bit en la cabecera IP.

El protocolo TCP fue creado pensando en medios guiados donde la tasa de error es muy baja y el principal problema que puede ocasionar pérdidas de paquetes es la congestión.

CAPITULO 3

3 Plan de desarrollo.

Se indican los recursos necesarios para el desarrollo del proceso, sin embargo no se descarta la introducción o eliminación de algún recurso adicional.

3.1 INTRODUCCIÓN

3.1.1 Descripción del proyecto

El objetivo del trabajo de fin de grado es la realización de una serie de experimentos con el fin de poder comprender y analizar los resultados obtenidos al aplicar diferentes tipos de algoritmos de control.

3.2 ORGANIZACIÓN DEL PROYECTO

3.2.1 Modelo de proceso

3.2.1.1 Ciclo de vida

En cuanto al modelo de ciclo de vida que se ha decidido seguir ha sido el modelo incremental ya que es el más adecuado al existir una fuerte interacción con el usuario. A medida que se iban realizando experimentos y aplicando algoritmos, se incrementa el número de experimentos o el número de algoritmos aplicados cada vez que los anteriores se aplicaban con éxito.

3.2.1.2 Hitos

Los hitos marcados a lo largo del proyecto:

Hito	Descripción	Fin
Documentación	Adquisición de información y documentación de la misma	vie 11/05/12
Análisis del trabajo	Análisis del trabajo, instalación y extensión del simulador	lun 04/06/12
Diseño del trabajo	Diseño de los experimentos	jue 21/06/12
Implementación	Implementación de las simulaciones y scripts necesarios	jue 05/07/12
Revisión/Documentación	Documentar. Finalización del proyecto	vie 28/08/12

Tabla 3.1. Hitos

3.2.2 Responsabilidades del trabajo de fin de grado

Al tratarse de un trabajo realizado por una sola persona, las distintas responsabilidades del mismo recaen sobre esa persona.

3.3 Proceso administrativo

3.3.1 Restricciones administrativas

El trabajo no se podrá realizar a tiempo completo ya que se está trabajando con jornada laboral de 8 horas.

3.3.2 Suposiciones, dependencias y restricciones

El desarrollo de este trabajo se enmarca dentro de un sistema operativo de libre distribución empleando el simulador NS, el cual incluirá las extensiones necesarias para emplear los distintos algoritmos de control.

3.3.3 Identificación y análisis de riegos

En este apartado se describen los principales riesgos que se han encontrado así como una descripción detallada y el estudio de su Incertidumbre, impacto y en qué parte del proceso pueden darse.

Los riesgos que aparecen en el siguiente listado atienden a la siguiente clasificación:

- *Riesgos del proyecto*, amenazan al plan del proyecto. Si aparecen puede que el plan temporal y los costes aumenten.

- *Riesgos técnicos*, amenazan la calidad y la planificación. Identifican problemas potenciales de diseño, implementación, interfaz, etc.

- *Riesgos de negocio*, amenazan la viabilidad del sistema a construir. Los candidatos para los principales riesgos del negocio son:

    - Construir un producto o sistema excelente que, en realidad, nadie quiere (riesgo de mercado).

    - Construir un producto que no concuerda con el interés estratégico de la empresa (riesgo estratégico).

    - Construir un producto que el departamento de ventas no sabe cómo vender.

    - Perder el apoyo de la gestión debido a un cambio de enfoque o cambios de personal (riesgo de dirección).

    - Perder presupuesto o personal asignado (riesgos de presupuesto).

De la misma manera el impacto de dichos riesgos atienden a:

- ***Catastrófico***: si se produjera el riesgo, se fracasaría en el objetivo.
- ***Crítico***: supone una degradación del rendimiento del proceso y del sistema final.
- ***Marginal***: consiste en un fracaso de menor grado en un objetivo no primordial.
- ***Despreciable***: se trata de un inconveniente menor.

Nombre del riesgo	La fecha de entrega planteada está muy ajustada
Enunciado	El plazo de presentación previsto se está fijado, con lo que el plazo de entrega de dicho TFG se ve condicionado.
Contexto	A lo largo de todo el proceso.
Categoría del riesgo	Técnico
Incertidumbre	30%
Impacto	Marginal
Fase/actividad	A lo largo del proceso.
Consecuencias	Debido a que la fecha es inaplazable puede que el proyecto quede pobre en alguno de los aspectos menos importantes.

Nombre del riesgo	Falta de entendimiento en el “guión”
Enunciado	Puesto que gran parte del desarrollo será durante el mes de julio-agosto, puede ser complicado establecer contacto con el “cliente” de la aplicación.
Contexto	A lo largo de todo el proceso.
Categoría del riesgo	Del proceso
Incertidumbre	55%
Impacto	Despreciable /Crítico dependiendo del momento
Fase/actividad	Análisis del trabajo / Diseño del trabajo.
Consecuencias	Debido a que la fecha es inaplazable puede que el proyecto quede pobre en alguno de los aspectos menos importantes.

Nombre del riesgo	Retraso en la planificación
Enunciado	Debido a una planificación poco realista pueden producirse demoras e incumplimiento de las fechas establecidas en los planes de fases y de iteraciones.
Contexto	Este riesgo puede darse a lo largo de todo el proceso debido a una planificación errónea.
Categoría del riesgo	De proceso, concretamente de gestión.
Incertidumbre	60%
Impacto	Crítico
Fase/actividad	A lo largo del proceso.
Consecuencias	Retraso en la entrega de los distintos artefactos así como del producto final.

Nombre del riesgo	Falta de familiaridad con la tecnología establecida
Enunciado	Debido a la falta de conocimientos ante la tecnología posible que se produzca algún fallo (o atraso) durante su desarrollo.
Contexto	A lo largo de todo el proceso.
Categoría del riesgo	Del proyecto
Incertidumbre	20%
Impacto	Marginal
Fase/actividad	Diseño del trabajo/Implementación.
Consecuencias	Ralentización en el avance de las diferentes fases y por tanto aumento de costes (formación).

Nombre del riesgo	Proceso de diseño pobre
Enunciado	El proceso de diseño en los experimentos puede no ser correcto debido a la falta de experiencia.
Contexto	A lo largo de todo el proceso de diseño.
Categoría del riesgo	Del proceso
Incertidumbre	60%
Impacto	Crítico
Fase/actividad	Diseño del trabajo
Consecuencias	Un mal diseño o un diseño pobre podría llevar a graves problemas en la fase de implementación e incluso a una repetición del proceso de diseño y por ello una ralentización del trabajo.

Nombre del riesgo	Pérdida de trabajo
Enunciado	Las copias de seguridad son escasas. Se trabaja sobre una versión que no es la última. El equipo donde se almacenaba la última versión se estropea antes de realizar la copia de seguridad correspondiente.
Contexto	A lo largo de todo el proceso.
Categoría del riesgo	Técnico
Incertidumbre	10%
Impacto	Catastrófico
Fase/actividad	Cualquier fase
Consecuencias	Habrá que volver a realizar el trabajo perdido, aumento de coste y retraso en la planificación.

3.3.4 Plan de acción y monitorización

A continuación se exponen los diferentes planes de acción para cada uno de los riesgos identificados en el apartado anterior. Se proporciona el escenario en el que podría darse el riesgo, el proceso de monitorización del mismo junto a los puntos de comprobación oportunos y el tipo de estrategia y los pasos a seguir en caso de que el riesgo se desencadene.

Nombre del riesgo	La fecha de entrega planteada está muy ajustada
Escenario	Este no es un riesgo que propiamente pueda darse a lo largo del desarrollo, es algo que debe tenerse siempre presente.
Punto de comprobación	En cada fase de desarrollo, se estudiará el trabajo realizado frente al tiempo disponible
Estrategia	Reducción del riesgo
Plan de acción	Intentar avanzar lo máximo posible y no dejar cosas complicadas para el final
Monitorización	Seguimiento continuo del trabajo restante frente al tiempo disponible

Nombre del riesgo	Falta de entendimiento en el "guión"
Escenario	En el caso de que el riesgo se diera las consecuencias variarán. Dependerá de en qué fase del proyecto se produzca y de la importancia del requisito que se pase por alto. Deben entonces variar los puntos del proyecto que tienen relación con dichos requisitos.
Punto de comprobación	Debe comprobarse a lo largo de todo el desarrollo del proceso, especialmente en las fases iniciales
Estrategia	Investigar el riesgo
Plan de acción	Revisión de la especificación de los requisitos y el modelo de los casos de uso, así como resolución de todas las dudas posibles antes del mes de agosto.
Monitorización	Continuos encuentros o llamadas entre el desarrollador y el cliente

Nombre del riesgo	Retraso en la planificación
Escenario	En el caso de que se produzca el riesgo, debe modificarse el calendario, para ello habrá que penalizar las actividades menos relevantes si es posible.
Punto de comprobación	Debe comprobarse a lo largo de todo el proceso de desarrollo
Estrategia	Reservar el riesgo
Plan de acción	Revisión periódica del calendario, e ir con un margen de tiempo favorable
Monitorización	Reuniones para modificar fechas.

Nombre del riesgo	Falta de familiaridad con la tecnología establecida
Escenario	En el caso de que acontezca el riesgo, debe identificarse el problema de forma detallada, para facilitar la búsqueda de información
Punto de comprobación	Al inicio de cada fase en que pueda darse este caso
Estrategia	Reservar el riesgo
Plan de acción	Búsqueda intensiva de materiales de apoyo como pueden ser manuales o libros de los lenguajes y/o herramientas utilizadas
Monitorización	Revisión de conocimientos adquiridos

Nombre del riesgo	Proceso de diseño pobre
Escenario	En caso de producirse un mal diseño o un diseño pobre podría ser necesario incluir una iteración más en la etapa de diseño para incluir mejoras, con el consecuente retraso
Punto de comprobación	Será necesario realizar comprobaciones durante toda la etapa de diseño y en la finalización del mismo
Estrategia	Reducción del riesgo
Plan de acción	Realizar una revisión exhaustiva de la etapa de diseño para asegurar que se está conforme con lo realizado hasta el momento.
Monitorización	Seguimiento de las partes de la etapa de diseño

Nombre del riesgo	Pérdida de trabajo
Escenario	En caso de que se diera el riesgo deberá hacerse una estimación del tiempo disponible en función del trabajo perdido.
Punto de comprobación	A lo largo de todo el proceso de desarrollo
Estrategia	Reservar el riesgo
Plan de acción	Durante todo el desarrollo haré copias de seguridad en servidores externos o discos extraíbles..
Monitorización	Seguimiento del tiempo disponible y el estado del trabajo

3.4 Técnicas

Para no hacer muy extenso este punto las técnicas empleadas se incluyen en el anexo D.

3.4.1 Sistema operativo

Para el sistema operativo se ha optado por una versión de libre distribución. En este caso se trata de la versión 11.10 de Ubuntu, para la cual se comentan, en el anexo A, los pasos a realizar para instalar el simulador NS.

3.5 Herramientas Software empleadas

A continuación se lleva a cabo una breve descripción de las herramientas que se han empleado en la elaboración del proyecto:

3.5.1 VIM

El nombre Vim es una contracción de «Vi IMproved», lo que podríamos traducir por «Vi mejorado». Es decir: Vim se basa en Vi, el cual es, el más clásico editor de texto a pantalla completa de Unix. Por ello, aunque en el mundo Unix abundan los editores de texto, Vi es el único editor —junto con Ed— que podemos tener la seguridad de encontrar en cualquier instalación de Unix.

Vim es uno de los editores de texto más completos que existen. Es extremadamente eficiente y posibilita el máximo rendimiento con el mínimo esfuerzo. Está específicamente diseñado para reducir el número de pulsaciones necesarias para editar un documento así como el tiempo que se tarda para realizarlas.

La principal característica de ambos es los diferentes modos que presentan para alternar entre ellos y realizar diversas operaciones, lo que les diferencia de la mayoría de editores comunes, los cuales tienen un único modo para introducir los comandos.

Los modos básicos que presenta VIM son 6:

- Modo comandos
- Modo inserción
- Modo línea de comandos
- Modo visual
- Modo selección
- Modo Ex (semejante a modo línea de comandos)

3.5.2 GVIM

GVIM es una versión gráfica del editor de textos Vim. Mantiene las funcionalidades del Vim y funciona con las bibliotecas gtk. Además añade menús y un entorno gráfico (funciona *fuera* de la consola/terminal).

Su principal ventaja, los menús desplegables, implican una curva de aprendizaje algo menos dura para los nuevos usuarios. Su principal desventaja es que no está instalado por defecto en todos los sistemas (por ejemplo, no está en los servidores sin entorno gráfico).

3.5.3 Lenguaje C++

C++ es un lenguaje de programación, diseñado como extensión del lenguaje de programación C.

Es uno de los lenguajes más potentes, ya que permite trabajar tanto a alto como a bajo nivel.

C++ abarca tres paradigmas de la programación:

- Programación estructurada.
- Programación genérica.
- Programación orientada a objetos.

Las principales características de C++ son las facilidades que proporciona para la programación orientada a objetos y para el uso de plantillas o programación genérica (templates).

Presenta una serie de propiedades que no poseen muchos lenguajes de alto nivel, como es la posibilidad de redefinir operadores (sobrecarga de operadores) y la identificación de tipos en tiempo de ejecución (RTTI).

C++ es el lenguaje con el que está implementado NS-2.

3.5.4 Lenguaje OTcl / Tcl /Tk

OTcl (Object Tcl) es una extensión orientada a objetos de Tcl (Tool Command Language), lenguaje de comandos interpretado y multiplataforma cuya característica principal es la posibilidad de facilitar el uso de aplicaciones en C/C++ dentro del propio intérprete.

Otra extensión muy conocida que es distribuida con el propio Tcl es Tk (Tool Kit), que proporciona un intérprete que añade la capacidad de crear interfaces gráficas de usuario.

Tcl se usa principalmente para el desarrollo rápido de prototipos, aplicaciones, scripts, interfaces gráficas y pruebas.

Debido a las características de extensibilidad mencionadas y a que OTcl es el otro lenguaje empleado en el simulador NS-2 para su implementación (junto a C++), se ha utilizado también este lenguaje orientado a objetos para extender el modelo del simulador.

3.5.5 NS / NAM

NS (Network Simulator) (http://www.isi.edu/nsnam/ns/), es un simulador de eventos discretos para redes, especializado en simulación de redes IP a nivel de paquete que permite la simulación tanto para redes cableadas como no cableadas, locales o vía satélite, con entornos diversos y topologías complejas.

Network Simulator proporciona una buena plataforma para la investigación en redes, avalada por una amplia difusión entre investigadores y que posee especial interés en interacciones multiprotocolo tales como protocolos de transporte, sesión, aplicación, algoritmos de encaminamiento y control de congestión.

Originalmente, NS surgió a finales de los años 80 como evolución de REAL Network Simulator, desarrollado bajo la supervisión del proyecto VINT (Virtual InterNetwork Testbed) y respaldado por DARPA (Defense Advanced Research Projects Agency).

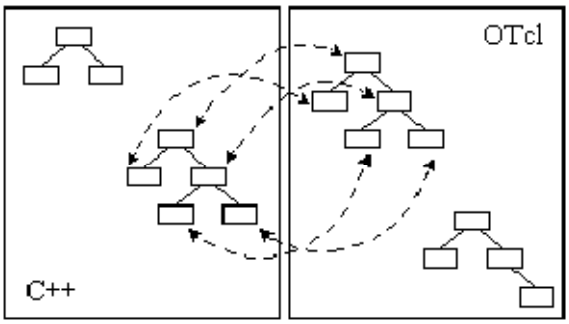
Actualmente el NS pertenece a un grupo de investigadores y desarrolladores de la Universidad de Berkeley, LBL (Lawrence Berkeley Laboratory), USC/ISI (University of Southern California/Information Sciences Institute) y Xerox PARC (Palo Alto Research Center).

La versión empleada en este proyecto es la dos (aunque actualmente se encuentra la versión tres), de la cual existen múltiples subversiones. El principal cambio desde la versión 1 ha sido una mejor subdivisión de las clases de objetos que componen el núcleo del simulador, lo cual permite un mejor desarrollo del mismo.

NS-2 es un simulador gratuito que se suministra con el código fuente completo. El simulador está pensado para ejecutarse en cualquier distribución de Linux, pero también existe una versión diseñada para ejecutarse sobre Windows. Sin embargo esta última versión no está testada tanto como la original, por lo que puede contener errores de funcionamiento.

NS está construido sobre la base del uso de dos tecnologías diferentes, C++ y OTcl que condicionan su arquitectura. Esta decisión se ha tomado por un motivo simple y que viene de las características de cada uno de estos dos lenguajes. El OTcl es un lenguaje interpretado o lenguaje de script con una sintaxis sencilla, mientras que C++ es un lenguaje con estructuras más complejas y que necesita ser compilado. Estas características proporcionan al uso del lenguaje OTcl una gran rapidez de desarrollo, y al C++ una ejecución mucho más rápida y eficiente de las simulaciones.

Es por esto que parece razonable pensar que un usuario que aborde el uso del NS como herramienta para realizar múltiples simulaciones cambiando sólo pequeños parámetros necesite de la agilidad de un lenguaje interpretado en lugar de tener que recompilar cada vez todo el sistema, esto es por lo que NS presenta a OTcl como interfaz entre el usuario y el núcleo. Por otra parte este núcleo está desarrollado en C++, permitiendo unas buenas prestaciones en cuanto a velocidad de simulación.



Figurar 3.1. Dualidad presente en NS

La solución de los desarrolladores de NS fue la de diseñar una jerarquía de clases que soportara los casos necesarios para la simulación, y replicarlo en ambos lenguajes. Si bien, como se puede observar en la **figura 3.1**, es cierto que hay clases puras de C++ que no necesitan ser controladas durante la simulación, y también hay clases completamente implementadas en OTcl que no tienen su respaldo en C++.

La arquitectura de NS se distribuye en varias capas. Comenzaremos a verlo de la forma que se muestra en la **figura 3.2**, de abajo arriba, que está quizás algo alejada de lo habitual, ya que en esta ocasión tenemos en la parte inferior la capa más cercana al usuario, la interfaz que le ofrece. Bien, como decíamos la interfaz con el usuario se asienta en el lenguaje TCL, pero presenta el problema de no ser orientado a objetos, por lo que se haría imposible mantener una jerarquía similar a la de C++. Por eso se diseñó una extensión de Tcl llamada OTcl que soportaba la orientación a objetos. Por otra parte el núcleo del simulador, que está compuesto principalmente por el sistema programador de eventos y por los componentes de red tiene una implementación en C++, la jerarquía compilada. Para poder unir estos dos bloques, se diseñó el sistema tclcl que hace de interfaz entre ambos, realmente es la interfaz del intérprete de OTcl que corre el simulador.

Cuando una clase se instancia esta se registra en el sistema Tcl y se crea una clase interpretada a imagen de ella en el sistema Tcl. Esta doble cara no quiere decir que sea una copia de la clase en otra zona de la aplicación, si no que es más bien una referencia, siendo accesibles los elementos de la clase compilada desde los miembros de la clase interpretada.

A la hora de desarrollar extensiones para NS hay que tener conciencia de esta dualidad. Todos los elementos desarrollados en C++ que vayan a tener su imagen

en OTcl han de heredar de la clase TclObject, que define los métodos necesarios para esta tarea.

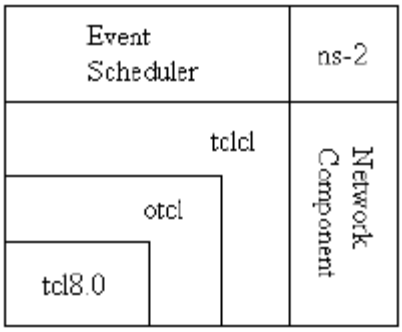


Figura 3.2. Arquitectura del simulador NS.

NAM (Network Animator, Animador de Red) (http://www.isi.edu/nsnam/nam/) es un visualizador gráfico de las simulaciones de NS-2. Es capaz de reproducir una animación del estado de la red variando a lo largo del tiempo. En las animaciones con NAM se pueden ver los paquetes viajando a través de la red o en la cola. También podemos interactuar generando otras gráficas, viendo las cabeceras de los paquetes o marcando paquetes para ver en qué momento son descartados o por qué nodos pasan. NAM se instala junto con NS-2 al instalar el paquete *all-in-one* y el comando que lo ejecuta es *nam*.

Las principales áreas y funciones de la herramienta NAM son las siguientes:

- *Animation Area*: en esta zona se visualiza el escenario y la animación del mismo.
- *Zoom In y Zoom Out*: permiten acercar o alejar el escenario y abarcar un mayor rango de visión de la simulación.
- *Stop/Play Animation*: detienen e inician, respectivamente, la animación.
- *Current Animation Time*: indica el tiempo actual de simulación.
- *Step*: valor que indica lo rápido que evolucionará la simulación (en milisegundos).
- *Menú*: agrupa varias opciones útiles (grabar la animación, imprimir el área de animación, ver la energía de los nodos, filtrar el tipo de paquetes a visualizar: datos, mac, routing, etc.).

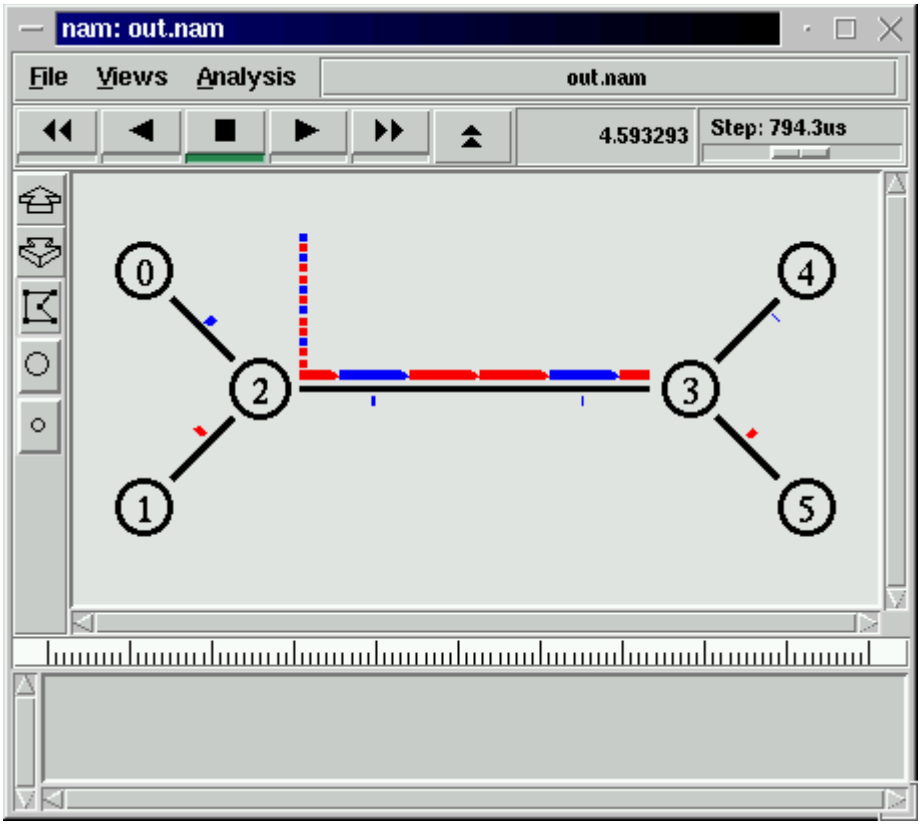


Figura 3.3.Interfaz de Nam

3.5.6 Gnuplot

Gnuplot (http://www.gnuplot.info/) es un programa en línea de comandos que permite dibujar gráficas de funciones en dos y tres dimensiones a través de las fórmulas que las definen. También puede dibujar gráficos usando tablas de coordenadas (formato sólo texto) generadas como salida de otros programas.

El software tiene copyright pero se distribuye libremente y está disponible en muchas plataformas. Es una herramienta para dibujar gráficas, pero muy completa y personalizable. Se pueden generar gráficas en 3D, guardarlas automáticamente como ficheros de imágenes, etc. Es muy útil para realizar scripts que automáticamente dibujen la gráfica y generen la imagen durante una batería de simulaciones por ejemplo.

3.5.7 AWK

AWK es un lenguaje de programación diseñado para procesar datos basados en texto, ya sean flujos de datos o ficheros, que emplea tipos de datos basados en listas asociativas y expresiones regulares, y se encuentra disponible en todas las versiones Unix.

El nombre AWK cuando está escrito todo en minúsculas (awk), hace referencia a la aplicación de Unix que interpreta programas escritos en lenguaje de programación AWK.

AWK se ha utilizado junto a Sed, éste último en menor medida debido a sus menores prestaciones (Sed es un editor de textos en línea de comando, también empleado para procesar y modificar líneas de texto), para extraer y analizar los datos generados por el simulador NS.

Con AWK se pueden procesar los ficheros de texto, filtrar los datos y obtener medidas significativas sobre los valores generados, haciendo uso tanto de patrones como de expresiones regulares y programación.

3.5.8 Enterprise Architect

Enterprise Architect (EA) (http://www.sparxsystems.com.au/) es una herramienta flexible, robusta y completa de modelado UML bajo plataforma Windows.

Abarca íntegramente el ciclo de vida de un proyecto, cubriendo el desarrollo de software desde la especificación de requisitos, análisis y diseño, hasta los modelos de testing y mantenimiento de software.

Además de ser una de las herramientas más completas que se encuentran actualmente, la elección de EA como software de modelado se ha realizado atendiendo también a la funcionalidad de ingeniería inversa que presenta, que permite generar diagramas básicos UML en varios lenguajes de programación.

3.5.9 Microsoft Project 2010

Microsoft Project (o MSP) es un software de administración de proyectos diseñado, desarrollado y comercializado por Microsoft para asistir a administradores de proyectos en el desarrollo de planes, asignación de recursos a tareas, dar seguimiento al progreso, administrar presupuesto y analizar cargas de trabajo.

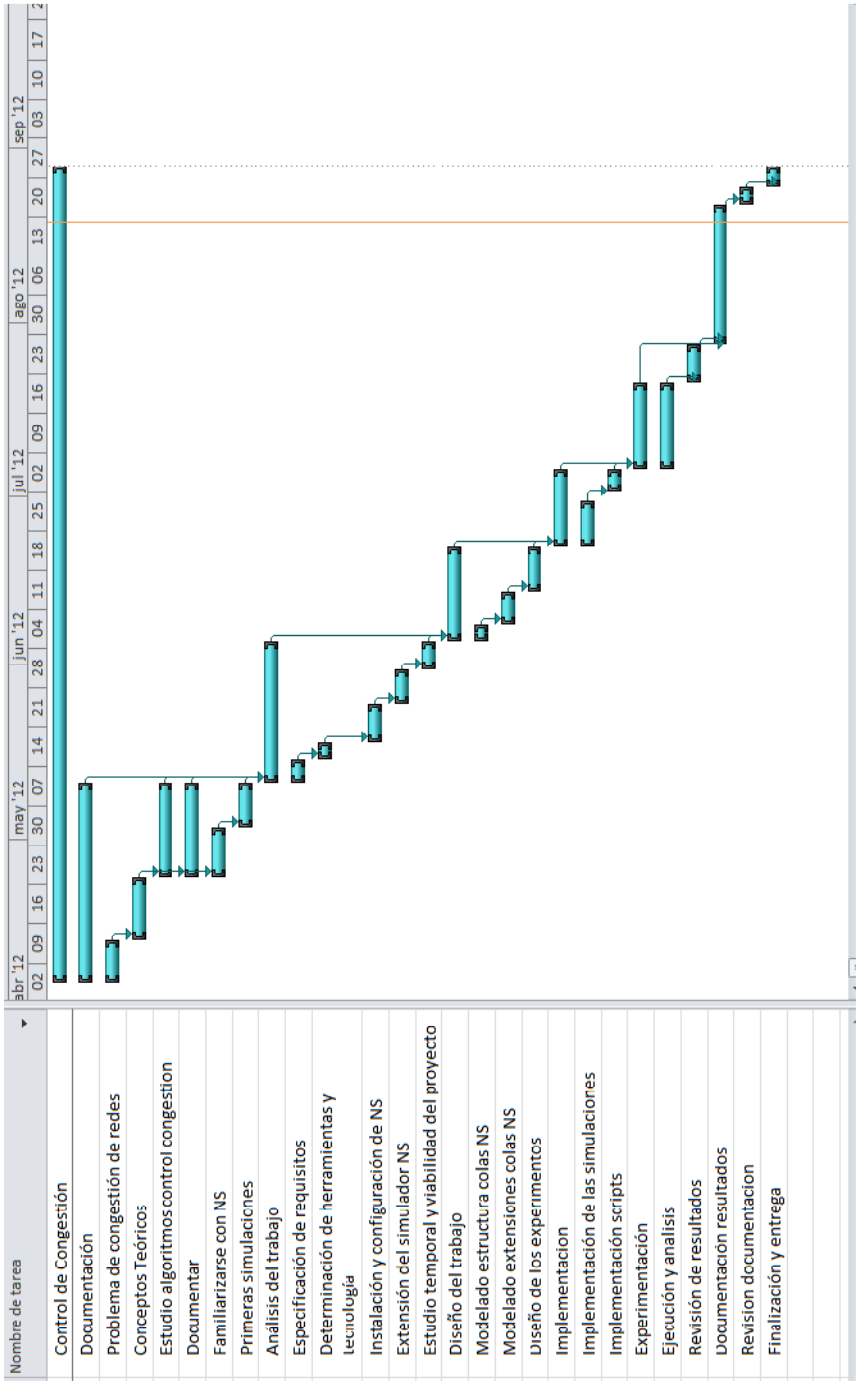
Se ha elegido esta herramienta ya que su uso se empleó en una asignatura del curso puente Grado en Ingeniería Informática

3.6 Identificación de tareas y planificación

Nombre de la tarea	Duración	Inicio	Fin
Control de Congestión	104 días	jue 05/04/12	mar 28/08/12
Documentación	26 días	jue 05/04/12	jue 10/05/12
Problema de congestión de redes	6 días	jue 05/04/12	jue 12/04/12
Conceptos Teóricos	7 días	vie 13/04/12	lun 23/04/12
Estudio algoritmos control congestión	13 días	mar 24/04/12	jue 10/05/12
Documentar	13 días	mar 24/04/12	jue 10/05/12
Familiarizarse con NS	7 días	mar 24/04/12	mié 02/05/12
Primeras simulaciones	6 días	jue 03/05/12	vie 11/05/12
Análisis del trabajo	17 días	vie 11/05/12	lun 04/06/12
Especificación de requisitos	2 días	vie 11/05/12	lun 14/05/12
Determinación de herramientas y tecnología	3 días	mar 15/05/12	jue 17/05/12
Instalación y configuración de NS	5 días	vie 18/05/12	jue 24/05/12
Extensión del simulador NS	4 días	vie 25/05/12	mié 30/05/12
Estudio temporal y viabilidad del proyecto	3 días	jue 31/05/12	lun 04/06/12
Diseño del trabajo	13 días	mar 05/06/12	jue 21/06/12
Modelado estructura colas NS	3 días	mar 05/06/12	jue 07/06/12
Modelado extensiones colas NS	4 días	vie 08/06/12	mié 13/06/12
Diseño de los experimentos	6 días	jue 14/06/12	jue 21/06/12
Implementación	10 días	vie 22/06/12	jue 05/07/12
Implementación de las simulaciones	6 días	vie 22/06/12	vie 29/06/12
Implementación scripts	4 días	lun 02/07/12	jue 05/07/12
Experimentación	11 días	vie 06/07/12	vie 20/07/12

Ejecución y análisis	11 días	vie 06/07/12	vie 20/07/12
Revisión de resultados	6 días	sáb 21/07/12	vie 27/07/12
Documentación resultados	18 días	sáb 28/07/12	mar 21/08/12
Revisión documentación	3 días	mié 22/08/12	vie 24/08/12
Finalización y entrega	3 días	sáb 25/08/12	mar 28/08/12

3.6.1 Diagrama de Gantt



Parte II

CAPITULO 4

4. Introducción y análisis

4.1 Requisitos

Los requisitos que surgieron para la ampliación de este software fueron los siguientes:

1. El simulador donde debemos introducir las nuevas técnicas de control de congestión debe ser NS-2
2. El simulador debe permitir ejecutar simulaciones de cualquier tipo de tráfico empleando como control AQM (*active queue management*) el algoritmo RED, así como sus variantes ARED,FRED y SRED.
3. El simulador debe permitir ejecutar simulaciones de cualquier tipo de tráfico empleando como control AQM (*active queue management*) el algoritmo GREEN.
4. El simulador debe permitir ejecutar simulaciones de cualquier tipo de tráfico empleando como control AQM (*active queue management*) el algoritmo BLUE.
5. El simulador debe permitir ejecutar simulaciones de cualquier tipo de tráfico empleando como control AQM (*active queue management*) el algoritmo PI.
6. El simulador debe permitir ejecutar simulaciones de cualquier tipo de tráfico empleando como control AQM (*active queue management*) el algoritmo PID.

Para poder realizar de manera correcta la extensión del simulador, primero se debe comprender como está organizado el sistema, así como su arquitectura.

4.2 Visión global

NS-2 está construido sobre la base del uso de dos tecnologías diferentes, C++ y OTcl que condicionan su arquitectura. Esta decisión se ha tomado por un motivo simple y que viene de las características de cada uno de estos dos lenguajes. El OTcl es un lenguaje interpretado o lenguaje de script con una sintaxis sencilla, mientras que C++ es un lenguaje con estructuras más complejas y que necesita ser compilado. Estas características proporcionan al uso del lenguaje OTcl una gran rapidez de desarrollo, y al C++ una ejecución mucho más rápida y eficiente de las simulaciones.

Es por esto que parece razonable pensar que un usuario que aborde el uso del NS como herramienta para realizar múltiples simulaciones cambiando sólo pequeños parámetros necesite de la agilidad de un lenguaje interpretado en lugar de tener que recompilar cada vez todo el sistema, esto es por lo que NS presenta a OTcl como interfaz entre el usuario y el núcleo. Por otra parte este núcleo está desarrollado en C++, permitiendo unas buenas prestaciones en cuanto a velocidad de simulación.

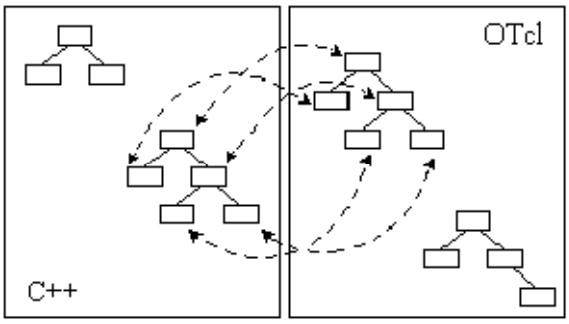


Figura 4.1 Dualidad presente en NS.

La solución de los desarrolladores de NS fue la de diseñar una jerarquía de clases que soportara los casos necesarios para la simulación, y replicarlo en ambos lenguajes. Si bien, como se puede observar en la **figura 4.1**, es cierto que hay clases puras de C++ que no necesitan ser controladas durante la simulación, y también hay clases completamente implementadas en OTcl que no tienen su respaldo en C++.

4.2.1 Arquitectura. Jerarquía de clases.

La arquitectura de NS se distribuye en varias capas. Comenzaremos a verlo de la forma que se muestra en la **figura 4.2**, de abajo arriba, que está quizás algo alejada de lo habitual, ya que en esta ocasión tenemos en la parte inferior la capa más cercana al usuario, la interfaz que le ofrece. Bien, como decíamos la interfaz con el usuario se asienta en el lenguaje TCL, pero presenta el problema de no ser orientado a objetos, por lo que se haría imposible mantener una jerarquía similar a la de C++. Por eso se diseñó una extensión de Tcl llamada OTcl que soportaba la orientación a objetos. Por otra parte el núcleo del simulador, que está compuesto principalmente por el sistema programador de eventos y por los componentes de red tiene una implementación en C++, la jerarquía compilada. Para poder unir estos dos bloques, se diseñó el sistema tclcl que hace de

interfaz entre ambos, realmente es la interfaz del intérprete de OTcl que corre el simulador.

Cuando una clase se instancia esta se registra en el sistema Tcl y se crea una clase interpretada a imagen de ella en el sistema Tcl. Esta doble cara no quiere decir que sea una copia de la clase en otra zona de la aplicación, si no que es más bien una referencia, Cuando una clase se instancia esta se registra en el sistema Tcl y se crea una clase interpretada a imagen de ella en el sistema Tcl. Esta doble cara no quiere decir que sea una copia de la clase en otra zona de la aplicación, si no que es más bien una referencia, siendo accesibles los elementos de la clase compilada desde los miembros de la clase interpretada.

A la hora de desarrollar extensiones para NS hay que tener conciencia de esta dualidad. Todos los elementos desarrollados en C++ que vayan a tener su imagen en OTcl han de heredar de la clase TclObject, que define los métodos necesarios para esta tarea.

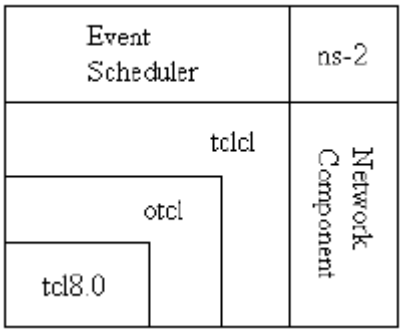


Figura 4.2. Arquitectura del simulador NS.

Una vez vista la arquitectura del simulador NS, se mostrará una visión general de la jerarquía de clases del simulador NS. Cómo tenemos que extender el simulador ya que este trabajo de fin de grado se basa en la inclusión de nuevos algoritmos y en la realización de experimentos empleando dichos algoritmos, nos debemos fijar en los otros AQM como RED o PI, comprobando que todos ellos son especializaciones de la clase *Queue*. En la siguiente figura se muestra una visión general de la jerarquía de clases del simulador NS-2.

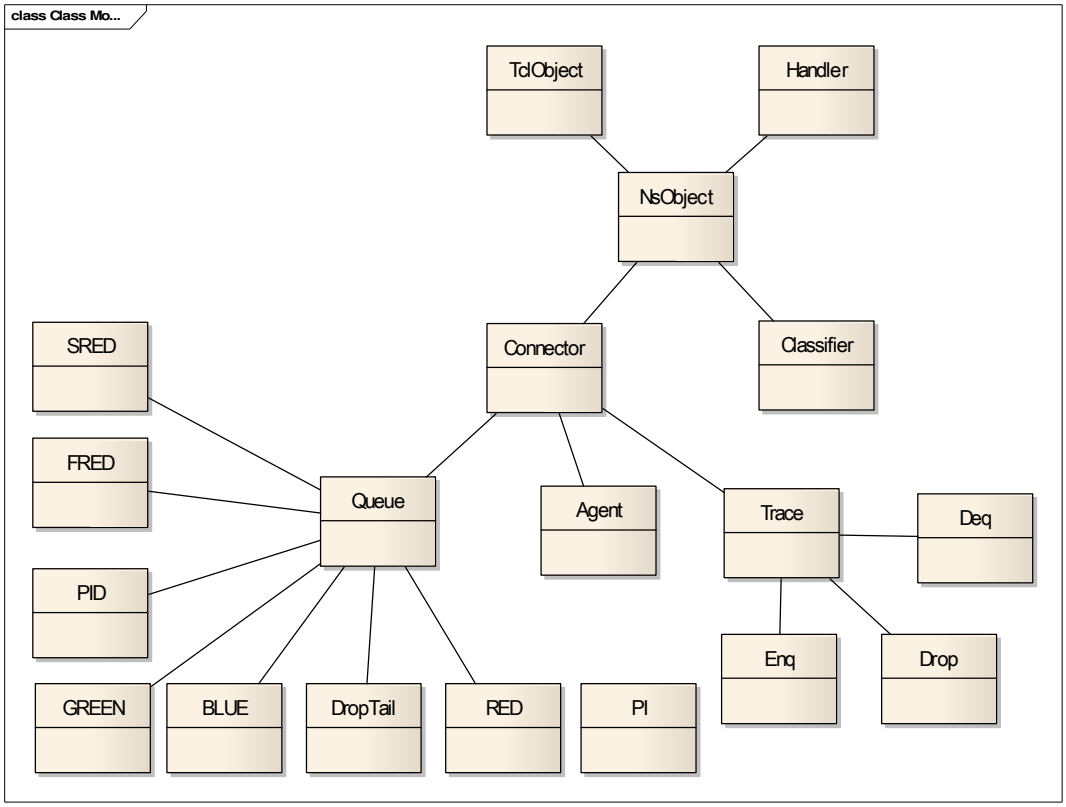
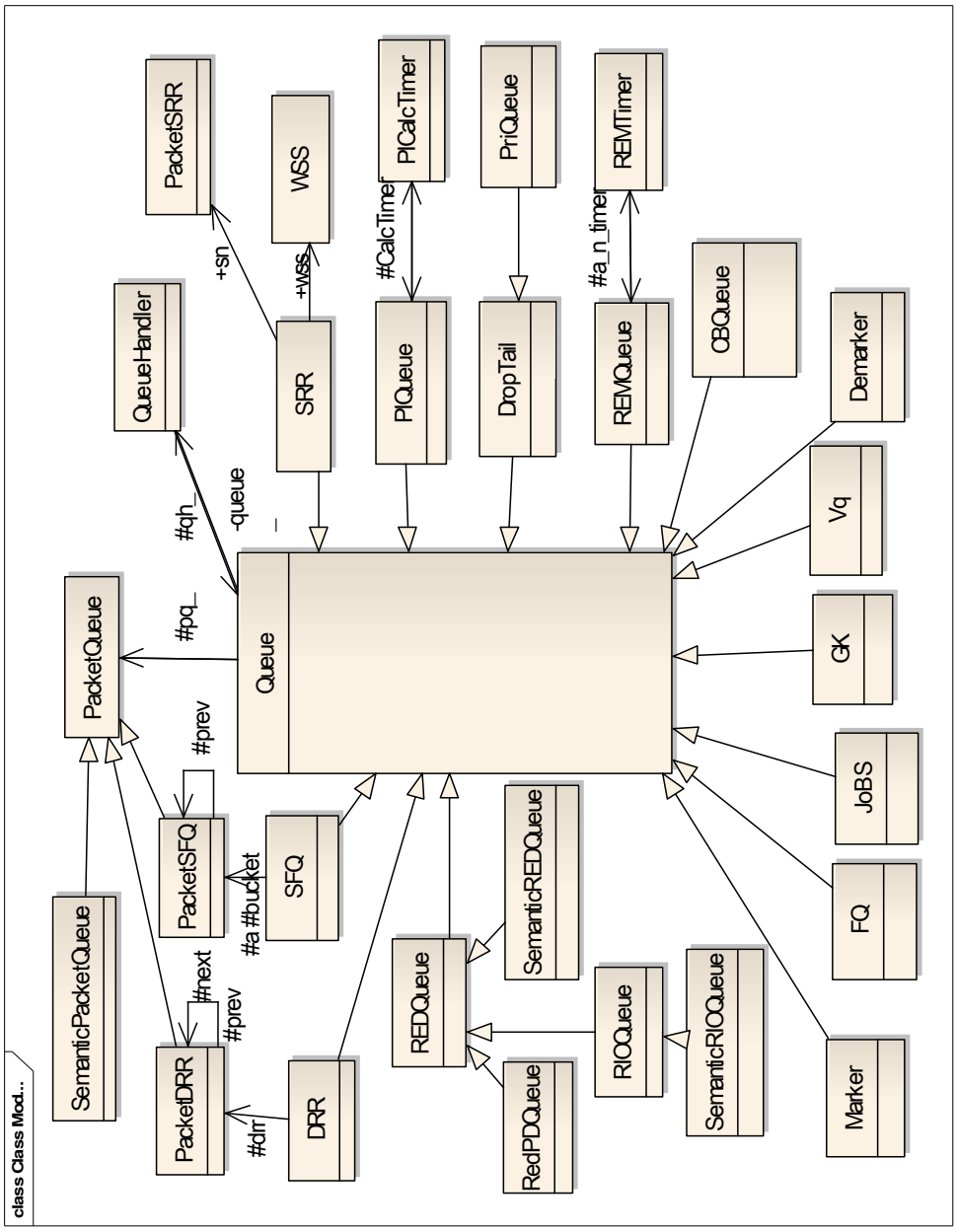


Figura 4.3. Jerarquía de clases.

4.2.2 Subsistema de colas

En la **figura 4.4** tenemos una visión simplificada pero bastante completa de lo que es la jerarquía de clases que tienen que ver con el manejo de las colas en los nodos. Si nos fijamos con atención veremos que no es algo demasiado complicado. Todas las técnicas de gestión de la cola implementan la clase *Queue*, ésta a su vez, mantiene una cola de paquetes a través de la clase *PaquetQueue* que es la que actúa realmente como contenedora de estos paquetes.

Ciertos controladores requieren de especializaciones de la clase *PaquetQueue*, en cambio otras necesitan muestrear la señal, para ello hacen uso de una especialización de la clase *TimerHandler* que le proporcione la ejecución de un método cada cierto intervalo de tiempo.


class Class Mod...
PacketSRR
WSS
PtCalcTimer
PriQueue
REMTimer
+sn
+wss
#CalcTimer
#a_n_timer
QueueHandler
SRR
PtQueue
DropTail
REMQueue
CBQueue
Demarker
#qh_
-queue
Vq
PacketQueue
#pq_
Queue
GK
#prev
JoBS
PacketSFQ
SFQ
SemanticREDQueue
#a #bucket
FQ
SemanticPacketQueue
#next
#prev
REDQueue
RIOQueue
SemanticRIOQueue
PacketDRR
DRR
RedPDQueue
Marker
#drr

CAPITULO 5

5. Extensión del simulador

5.1 Visión global.

A continuación se mostrará una visión global sobre la estructura de las clases que intervienen a la hora de extender el simulador con algún nuevo controlador.

Para añadir un nuevo algoritmo de control y descarte de paquetes se deben incluir las siguientes clases.

La clase base será nuestra propia implementación de la clase *Queue*, esta clase será la que defina los comportamientos ante la llegada de los nuevos paquetes a la cola. La cola no está mantenida en esta especialización de *Queue*, en realidad es una instancia de la clase *PaquetQueue* que sí tiene realmente los paquetes (es básicamente una cola FIFO) y es esta instancia la que nuestra clase, denominada en la **figura 5.1** como *COLAQueue*, debe manejar.

Además si se necesitan muestrear los datos cada cierto periodo fijo, tiene que hacer su aparición la clase *TimerHandler*. Es una clase que deberemos especializar para definir que en el método expire(), que es el método que se ejecuta cuando ocurre el evento de fin de tiempo, se llame a la función de nuestra clase que muestree la señal.

La clase de entre las que están en la **figura 5.1** y que hemos definido como necesaria para que todo funcione es *COLAClass*, la cual es una implementación de la clase *TclClass*, y lo que hace es registrar la clase *COLAQueue* con un nombre en su lugar en la jerarquía interpretada.

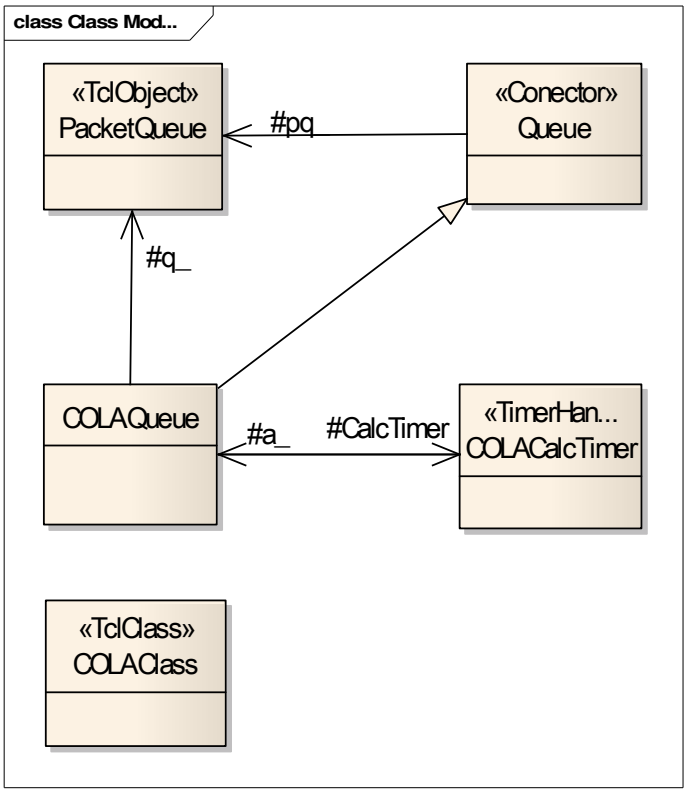


Figura 5.1. Clases de una cola-Relaciones

5.2. Extensión del simulador.

Cuando hablamos de extender NS-2 nos estamos refiriendo al hecho de modificar el código de NS o añadir nuevo código para incrementar sus funcionalidades. En estos casos siempre es necesario volver a compilar NS-2. En [6] se explica cómo realizar estos pasos mediante ejemplos. En este caso nosotros supondremos que ya tenemos los ficheros que queremos añadir a NS-2, añadiremos varios tipos de colas pero hablaremos por ejemplo de FRED , mediante los archivos fred.cc y fred.h, que se adjuntan a este trabajo fin de grado Esta cola implementa distintas variante de red *Flow Random Early Drop* (ya descrito en el punto 2.4.2.1.2).

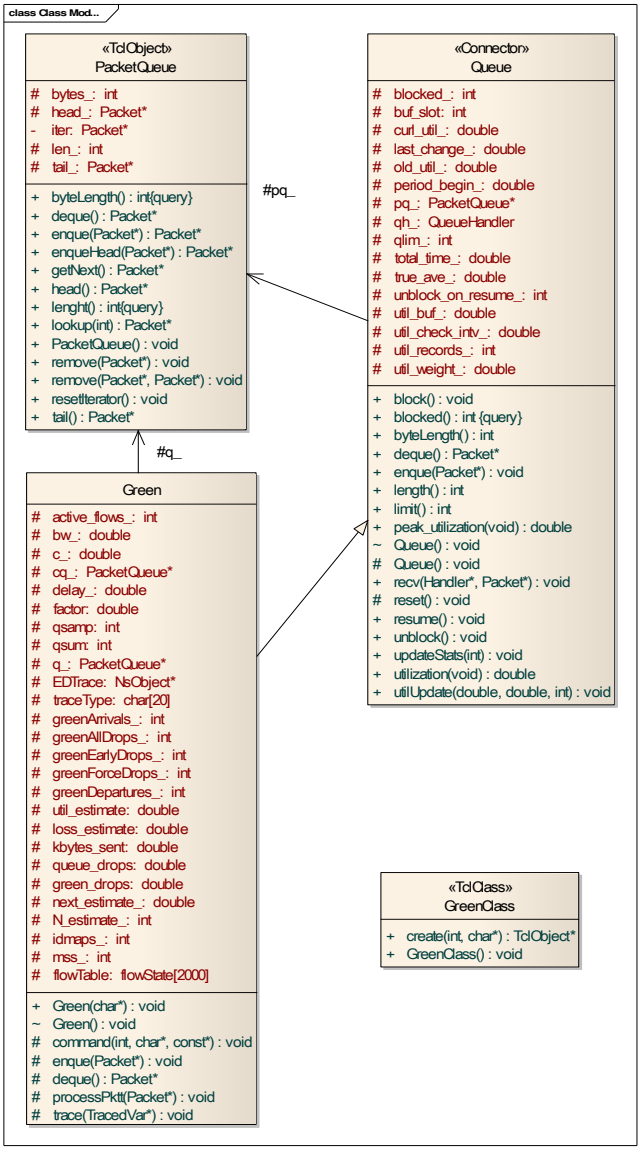
Habitualmente para todas las colas que añadamos y en general para cualquier nuevo elemento necesitaremos dos ficheros fuente con código en C++. Uno de ellos tendrá extensión .h y es el fichero de cabecera donde se definirán las clases y métodos a usar. El otro tendrá extensión .cc, en el que estarán implementados los métodos. Además las variables que se quieran modificar desde los ficheros Tcl, deberán incluirse, con su valor

por defecto, en el fichero *ns-default.tcl* incluido en la distribución de NS-2. Para poder ver detalladamente los pasos para la inclusión de dicha cola consultar el Anexo B. Además el Anexo A contendrá las modificaciones necesarias que se tuvieron que hacer del paquete fuente NS-2 para poder instalarse sobre la versión 11.10 de Ubuntu (versión sobre la que se realizaron las pruebas de simulación).

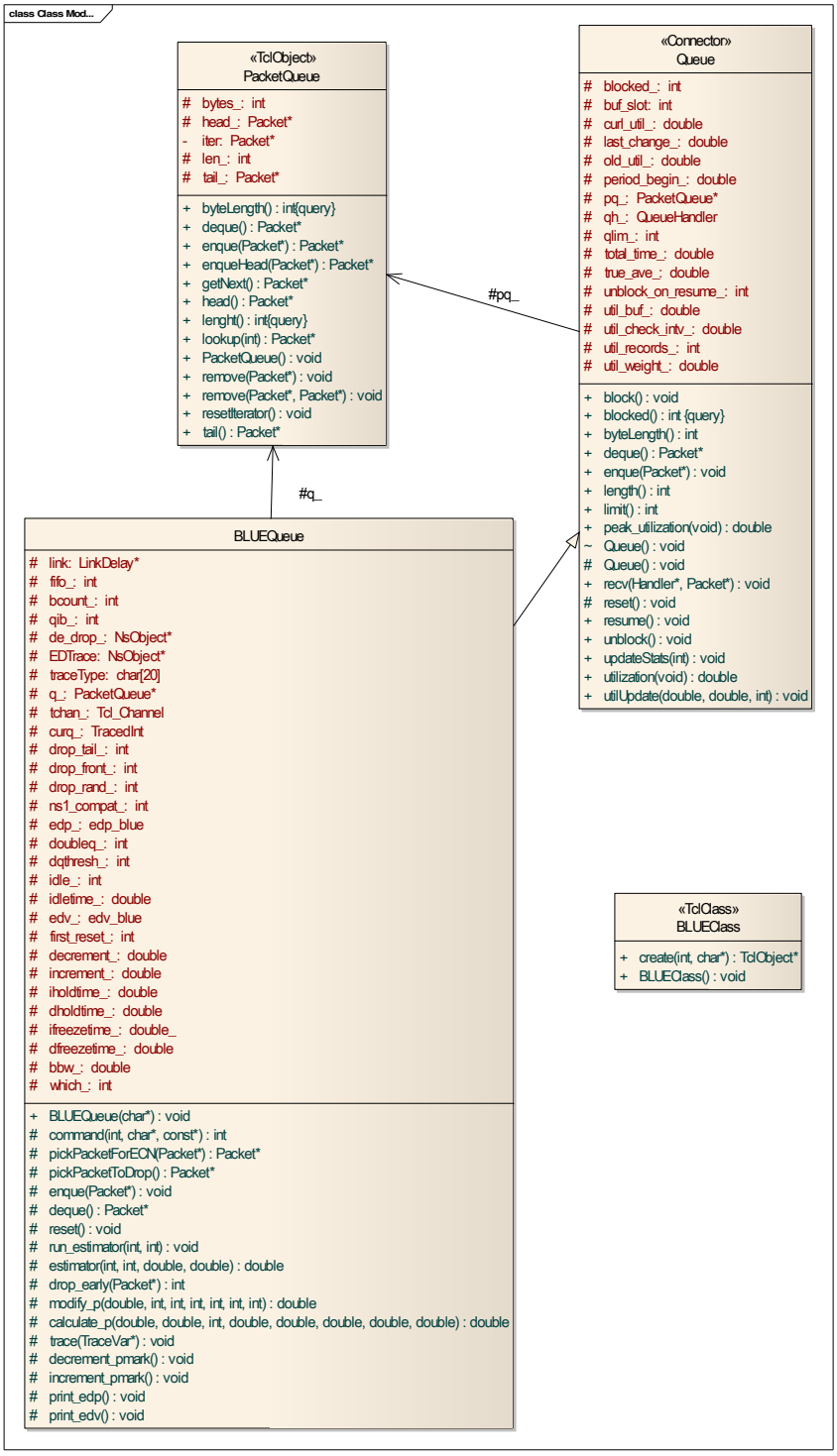
5.3 Colas

5.3.1 Vista lógica

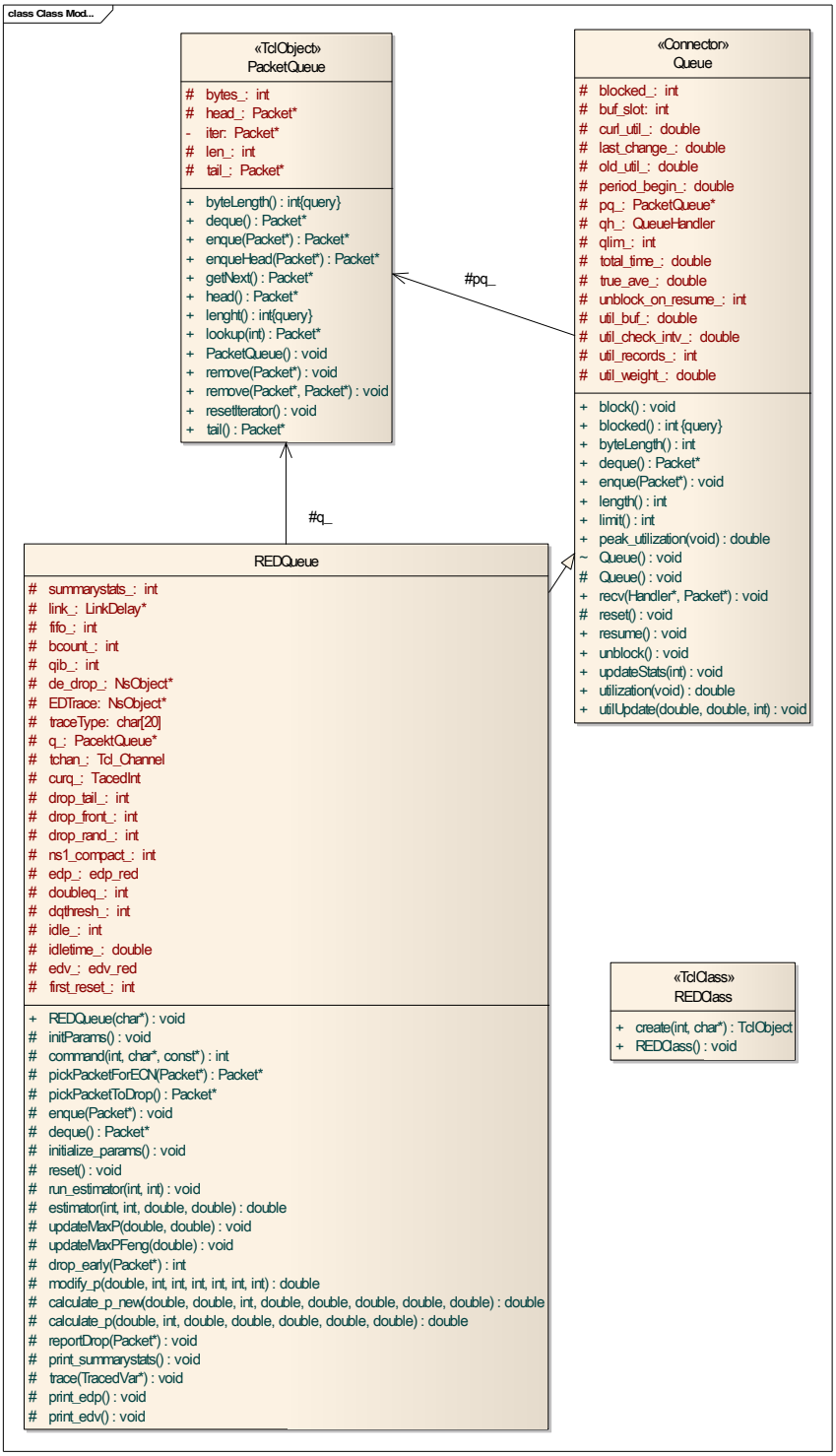
GREEN



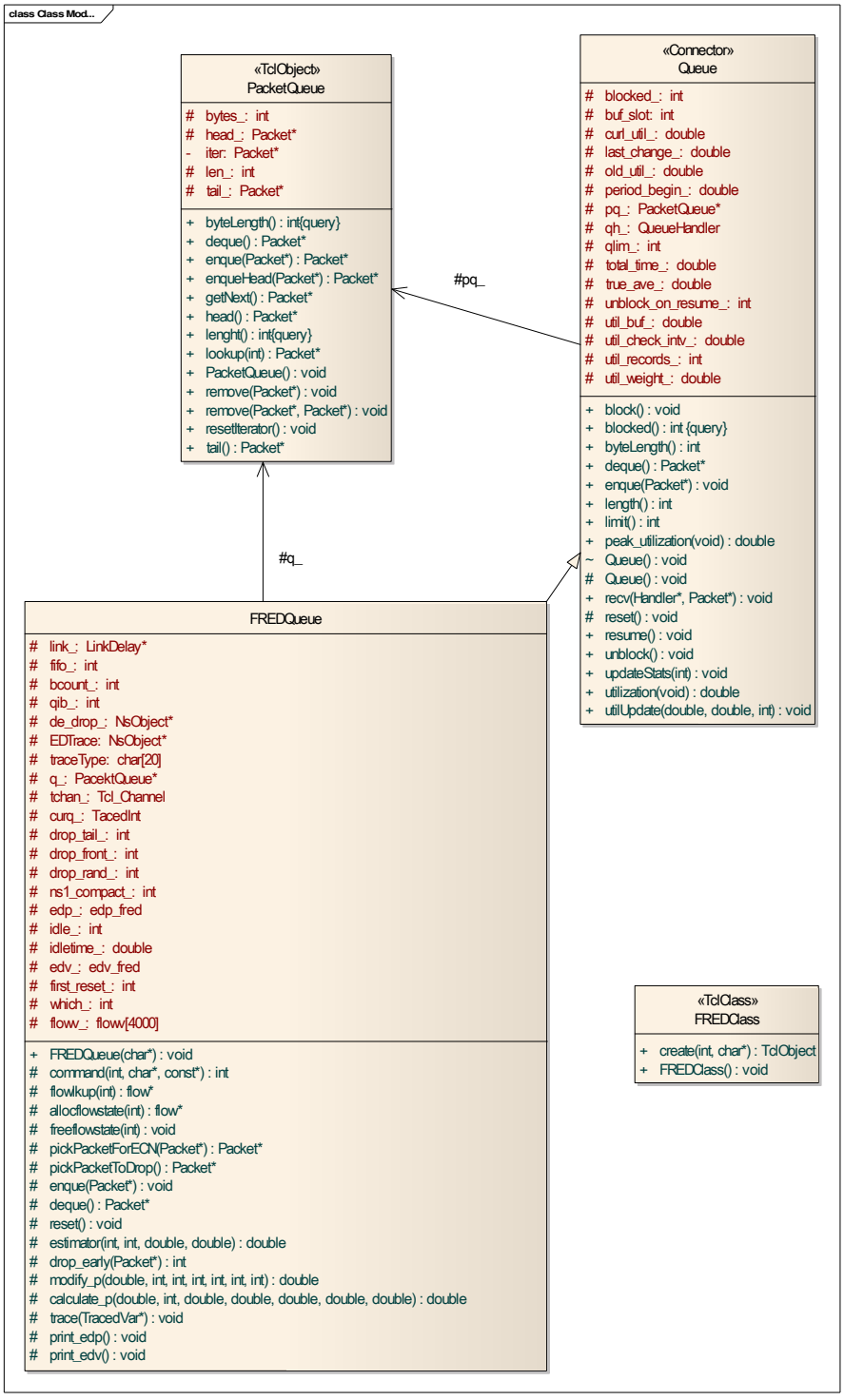
BLUE


class Class Mod...
«TclObject»
PacketQueue
bytes_: int
head_: Packet*
- iter: Packet*
len_: int
tail_: Packet*
+ byteLength() : int{query}
+ deque() : Packet*
+ enque(Packet*) : Packet*
+ enqueHead(Packet*) : Packet*
+ getNext() : Packet*
+ head() : Packet*
+ lenght() : int{query}
+ lookup(int) : Packet*
+ PacketQueue() : void
+ remove(Packet*) : void
+ remove(Packet*, Packet*) : void
+ resetIterator() : void
+ tail() : Packet*
«Connector»
Queue
blocked_: int
buf_slot: int
curl_util_: double
last_change_: double
old_util_: double
period_begin_: double
pq_: PacketQueue*
qh_: QueueHandler
qlim_: int
total_time_: double
true_ave_: double
unblock_on_resume_: int
util_buf_: double
util_check_intv_: double
util_records_: int
util_weight_: double
+ block() : void
+ blocked() : int {query}
+ byteLength() : int
+ deque() : Packet*
+ enque(Packet*) : void
+ length() : int
+ limit() : int
+ peak_utilization(void) : double
~ Queue() : void
# Queue() : void
+ recv(Handler*, Packet*) : void
# reset() : void
+ resume() : void
+ unblock() : void
+ updateStats(int) : void
+ utilization(void) : double
+ utilUpdate(double, double, int) : void
#pq_
#q_
BLUEQueue
link: LinkDelay*
fifo_: int
bcount_: int
qib_: int
de_drop_: NsObject*
EDTrace: NsObject*
traceType: char[20]
q_: PacketQueue*
tchan_: Tcl_Channel
curq_: TracedInt
drop_tail_: int
drop_front_: int
drop_rand_: int
ns1_compat_: int
edp_: edp_blue
doubleq_: int
dqthresh_: int
idle_: int
idletime_: double
edv_: edv_blue
first_reset_: int
decrement_: double
increment_: double
iholdtime_: double
dholdtime_: double
ifreezetime_: double_
dfreezetime_: double
bbw_: double
which_: int
+ BLUEQueue(char*) : void
# command(int, char*, const*) : int
# pickPacketForECN(Packet*) : Packet*
# pickPacketToDrop() : Packet*
# enque(Packet*) : void
# deque() : Packet*
# reset() : void
# run_estimator(int, int) : void
# estimator(int, int, double, double) : double
# drop_early(Packet*) : int
# modify_p(double, int, int, int, int, int) : double
# calculate_p(double, double, int, double, double, double, double, double) : double
# trace(TraceVar*) : void
# decrement_pmark() : void
# increment_pmark() : void
# print_edp() : void
# print_edv() : void
«TclClass»
BLUEClass
+ create(int, char*) : TclObject*
+ BLUEClass() : void

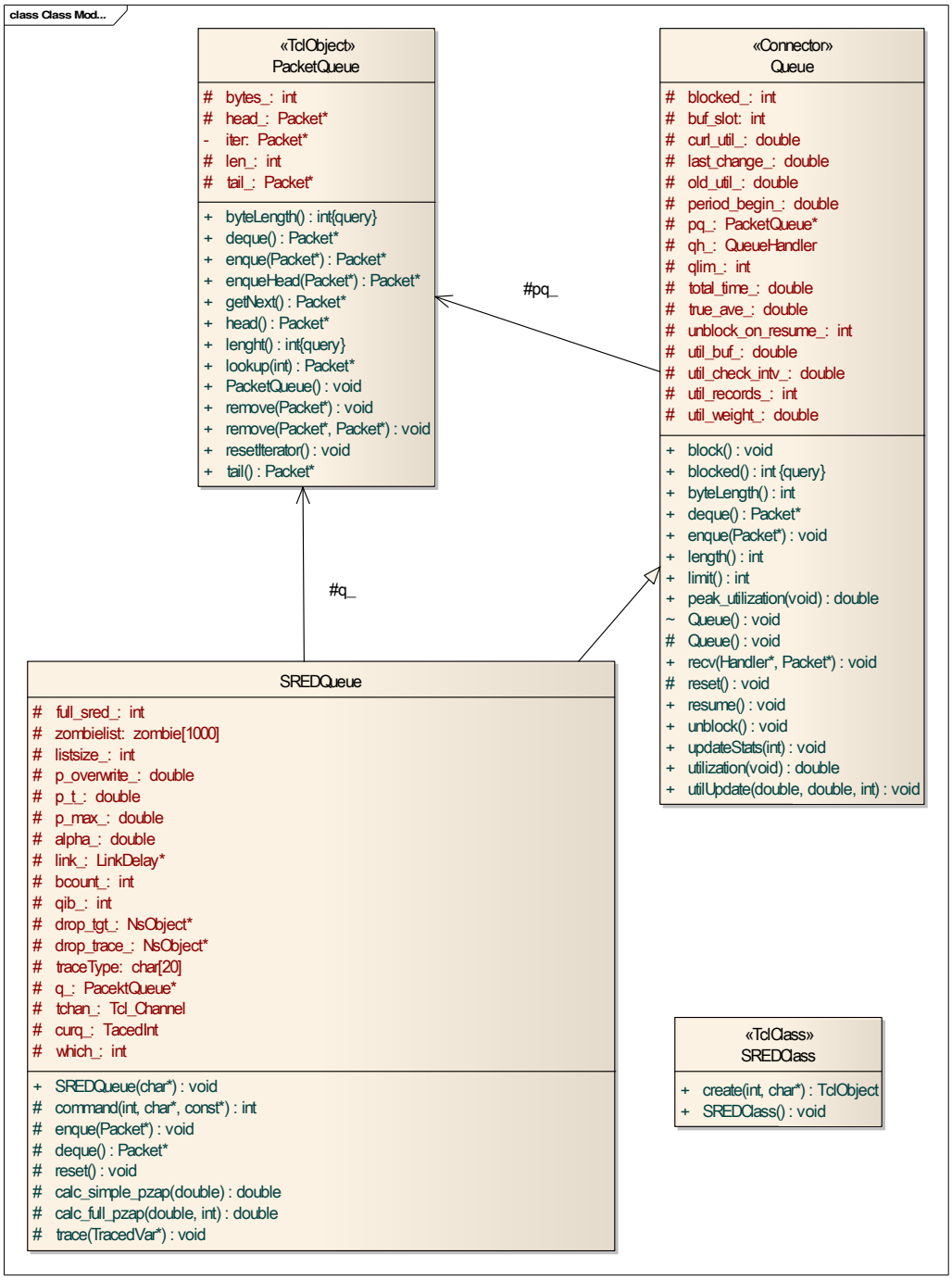
RED


class Class Mod...
«TclObject»
PacketQueue
bytes_: int
head_: Packet*
- iter: Packet*
len_: int
tail_: Packet*
+ byteLength() : int{query}
+ deque() : Packet*
+ enque(Packet*) : Packet*
+ enqueHead(Packet*) : Packet*
+ getNext() : Packet*
+ head() : Packet*
+ lenght() : int{query}
+ lookup(int) : Packet*
+ PacketQueue() : void
+ remove(Packet*) : void
+ remove(Packet*, Packet*) : void
+ resetIterator() : void
+ tail() : Packet*
«Connector»
Queue
blocked_: int
buf_slot: int
curl_util_: double
last_change_: double
old_util_: double
period_begin_: double
pq_: PacketQueue*
qh_: QueueHandler
qlim_: int
total_time_: double
true_ave_: double
unblock_on_resume_: int
util_buf_: double
util_check_intv_: double
util_records_: int
util_weight_: double
+ block() : void
+ blocked() : int {query}
+ byteLength() : int
+ deque() : Packet*
+ enque(Packet*) : void
+ length() : int
+ limit() : int
+ peak_utilization(void) : double
~ Queue() : void
# Queue() : void
+ recv(Handler*, Packet*) : void
# reset() : void
+ resume() : void
+ unblock() : void
+ updateStats(int) : void
+ utilization(void) : double
+ utilUpdate(double, double, int) : void
#pq_
#q_
REDQueue
summarystats_: int
link_: LinkDelay*
fifo_: int
bcount_: int
qib_: int
de_drop_: NsObject*
EDTrace: NsObject*
traceType: char[20]
q_: PacektQueue*
tchan_: Tcl_Channel
curq_: TacedInt
drop_tail_: int
drop_front_: int
drop_rand_: int
ns1_compact_: int
edp_: edp_red
doubleq_: int
dqthresh_: int
idle_: int
idletime_: double
edv_: edv_red
first_reset_: int
+ REDQueue(char*) : void
# initParams() : void
# command(int, char*, const*) : int
# pickPacketForECN(Packet*) : Packet*
# pickPacketToDrop() : Packet*
# enque(Packet*) : void
# deque() : Packet*
# initialize_params() : void
# reset() : void
# run_estimator(int, int) : void
# estimator(int, int, double, double) : double
# updateMaxP(double, double) : void
# updateMaxPFeng(double) : void
# drop_early(Packet*) : int
# modify_p(double, int, int, int, int, int) : double
# calculate_p_new(double, double, int, double, double, double, double, double) : double
# calculate_p(double, int, double, double, double, double, double) : double
# reportDrop(Packet*) : void
# print_summarystats() : void
# trace(TracedVar*) : void
# print_edp() : void
# print_edv() : void
«TclClass»
REDClass
+ create(int, char*) : TclObject
+ REDClass() : void

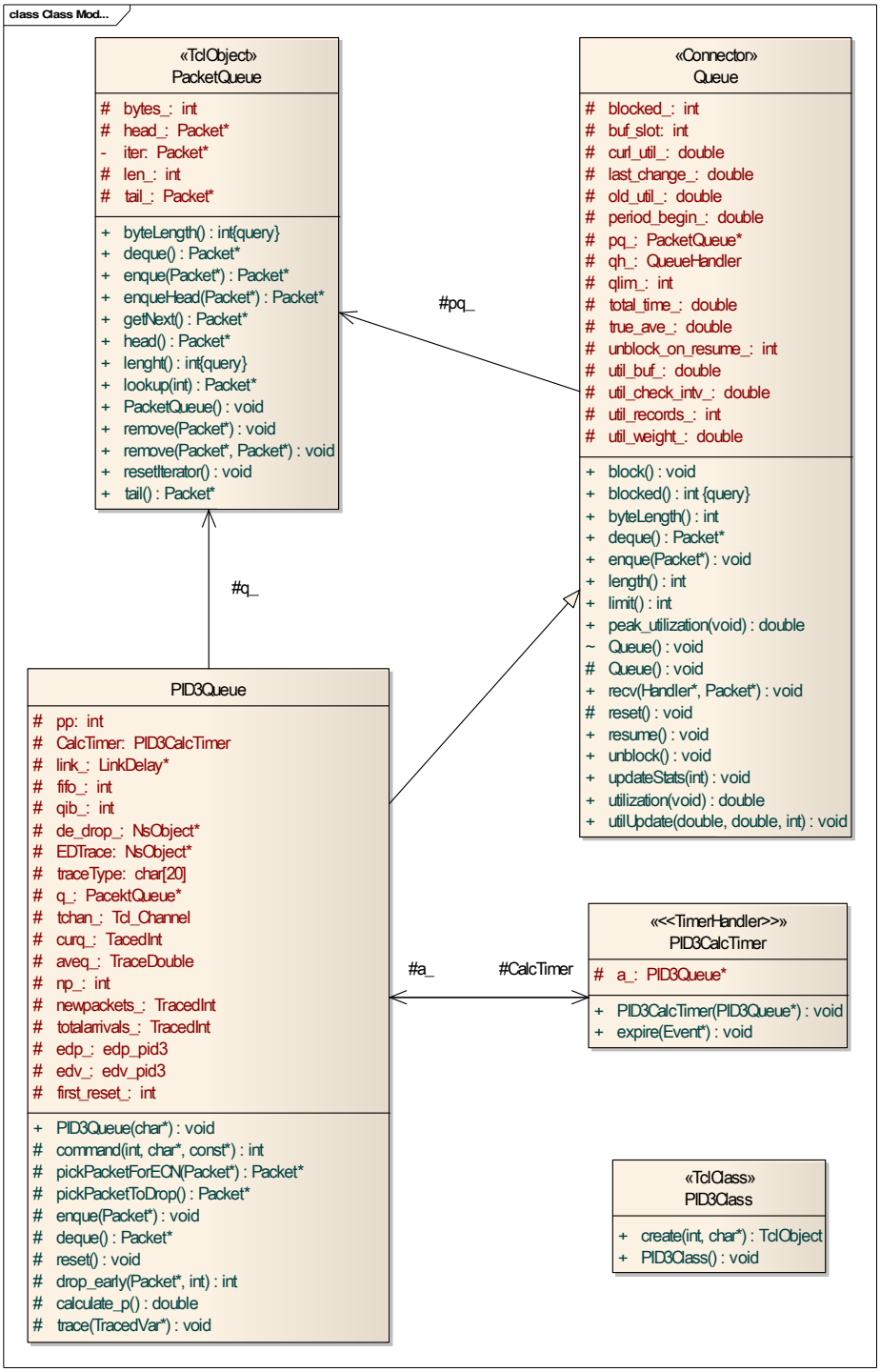
FRED


class Class Mod...
«TclObject»
PacketQueue
bytes_: int
head_: Packet*
- iter: Packet*
len_: int
tail_: Packet*
+ byteLength() : int{query}
+ deque() : Packet*
+ enque(Packet*) : Packet*
+ enqueHead(Packet*) : Packet*
+ getNext() : Packet*
+ head() : Packet*
+ lenght() : int{query}
+ lookup(int) : Packet*
+ PacketQueue() : void
+ remove(Packet*) : void
+ remove(Packet*, Packet*) : void
+ resetIterator() : void
+ tail() : Packet*
«Connector»
Queue
blocked_: int
buf_slot: int
curl_util_: double
last_change_: double
old_util_: double
period_begin_: double
pq_: PacketQueue*
qh_: QueueHandler
qlim_: int
total_time_: double
true_ave_: double
unblock_on_resume_: int
util_buf_: double
util_check_intv_: double
util_records_: int
util_weight_: double
+ block() : void
+ blocked() : int {query}
+ byteLength() : int
+ deque() : Packet*
+ enque(Packet*) : void
+ length() : int
+ limit() : int
+ peak_utilization(void) : double
~ Queue() : void
# Queue() : void
+ recv(Handler*, Packet*) : void
# reset() : void
+ resume() : void
+ unblock() : void
+ updateStats(int) : void
+ utilization(void) : double
+ utilUpdate(double, double, int) : void
#pq_
#q_
FREDQueue
link_: LinkDelay*
fifo_: int
bcount_: int
qib_: int
de_drop_: NsObject*
EDTrace: NsObject*
traceType: char[20]
q_: PacektQueue*
tchan_: Tcl_Channel
curq_: TacedInt
drop_tail_: int
drop_front_: int
drop_rand_: int
ns1_compat_: int
edp_: edp_fred
idle_: int
idletime_: double
edv_: edv_fred
first_reset_: int
which_: int
flow_: flow[4000]
+ FREDQueue(char*) : void
# command(int, char*, const*) : int
# flowlkup(int) : flow*
# allocflowstate(int) : flow*
# freeflowstate(int) : void
# pickPacketForECN(Packet*) : Packet*
# pickPacketToDrop() : Packet*
# enque(Packet*) : void
# deque() : Packet*
# reset() : void
# estimator(int, int, double, double) : double
# drop_early(Packet*) : int
# modify_p(double, int, int, int, int, int, int) : double
# calculate_p(double, int, double, double, double, double, double) : double
# trace(TracedVar*) : void
# print_edp() : void
# print_edv() : void
«TclClass»
FREDClass
+ create(int, char*) : TclObject
+ FREDClass() : void

SRED


class Class Mod...
«TclObject»
PacketQueue
bytes_: int
head_: Packet*
- iter: Packet*
len_: int
tail_: Packet*
+ byteLength() : int{query}
+ deque() : Packet*
+ enque(Packet*) : Packet*
+ enqueHead(Packet*) : Packet*
+ getNext() : Packet*
+ head() : Packet*
+ lenght() : int{query}
+ lookup(int) : Packet*
+ PacketQueue() : void
+ remove(Packet*) : void
+ remove(Packet*, Packet*) : void
+ resetIterator() : void
+ tail() : Packet*
«Connector»
Queue
blocked_: int
buf_slot: int
curl_util_: double
last_change_: double
old_util_: double
period_begin_: double
pq_: PacketQueue*
qh_: QueueHandler
qlim_: int
total_time_: double
true_ave_: double
unblock_on_resume_: int
util_buf_: double
util_check_intv_: double
util_records_: int
util_weight_: double
+ block() : void
+ blocked() : int {query}
+ byteLength() : int
+ deque() : Packet*
+ enque(Packet*) : void
+ length() : int
+ limit() : int
+ peak_utilization(void) : double
~ Queue() : void
# Queue() : void
+ recv(Handler*, Packet*) : void
# reset() : void
+ resume() : void
+ unblock() : void
+ updateStats(int) : void
+ utilization(void) : double
+ utilUpdate(double, double, int) : void
#pq_
#q_
SREDQueue
full_sred_: int
zombielist: zombie[1000]
listsize_: int
p_overwrite_: double
p_t_: double
p_max_: double
alpha_: double
link_: LinkDelay*
bcount_: int
qib_: int
drop_tgt_: NsObject*
drop_trace_: NsObject*
traceType: char[20]
q_: PacektQueue*
tchan_: Tcl_Channel
curq_: TacedInt
which_: int
+ SREDQueue(char*) : void
# command(int, char*, const*) : int
# enque(Packet*) : void
# deque() : Packet*
# reset() : void
# calc_simple_pzap(double) : double
# calc_full_pzap(double, int) : double
# trace(TracedVar*) : void
«TclClass»
SREDClass
+ create(int, char*) : TclObject
+ SREDClass() : void

PID3


class Class Mod...
«TclObject»
PacketQueue
bytes_: int
head_: Packet*
- iter: Packet*
len_: int
tail_: Packet*
+ byteLength() : int{query}
+ deque() : Packet*
+ enque(Packet*) : Packet*
+ enqueHead(Packet*) : Packet*
+ getNext() : Packet*
+ head() : Packet*
+ lenght() : int{query}
+ lookup(int) : Packet*
+ PacketQueue() : void
+ remove(Packet*) : void
+ remove(Packet*, Packet*) : void
+ resetIterator() : void
+ tail() : Packet*
«Connector»
Queue
blocked_: int
buf_slot: int
curl_util_: double
last_change_: double
old_util_: double
period_begin_: double
pq_: PacketQueue*
qh_: QueueHandler
qlim_: int
total_time_: double
true_ave_: double
unblock_on_resume_: int
util_buf_: double
util_check_intv_: double
util_records_: int
util_weight_: double
+ block() : void
+ blocked() : int {query}
+ byteLength() : int
+ deque() : Packet*
+ enque(Packet*) : void
+ length() : int
+ limit() : int
+ peak_utilization(void) : double
~ Queue() : void
# Queue() : void
+ recv(Handler*, Packet*) : void
# reset() : void
+ resume() : void
+ unblock() : void
+ updateStats(int) : void
+ utilization(void) : double
+ utilUpdate(double, double, int) : void
#pq_
#q_
PID3Queue
pp: int
CalcTimer: PID3CalcTimer
link_: LinkDelay*
fifo_: int
qib_: int
de_drop_: NsObject*
EDTrace: NsObject*
traceType: char[20]
q_: PacektQueue*
tchan_: Tcl_Channel
curq_: TacedInt
aveq_: TraceDouble
np_: int
newpackets_: TracedInt
totalarrivals_: TracedInt
edp_: edp_pid3
edv_: edv_pid3
first_reset_: int
+ PID3Queue(char*) : void
# command(int, char*, const*) : int
# pickPacketForECN(Packet*) : Packet*
# pickPacketToDrop() : Packet*
# enque(Packet*) : void
# deque() : Packet*
# reset() : void
# drop_early(Packet*, int) : int
# calculate_p() : double
# trace(TracedVar*) : void
«<<TimerHandler>>»
PID3CalcTimer
a_: PID3Queue*
+ PID3CalcTimer(PID3Queue*) : void
+ expire(Event*) : void
#a_
#CalcTimer
«TclClass»
PID3Class
+ create(int, char*) : TclObject
+ PID3Class() : void

5.3.2 Diseño detallado de las clases

Green

Estructura: flowState

- rtt_actual_ *double*
  Valor para el flujo rtt actual.
- flowid_ *int*
  Identificador del flujo.
- active_ *bool*
  Valor booleano que indica si el flujo está activo o no.

Clase: Green

Responsabilidad:

Atributos:

- Protegidos:

- qsamp *TracedInt*
  Tamaño actual de la cola.
- qsum *int*
  Número de paquetes que se utilizan para obtener el tamaño medio de la cola.
- q_, cq_ *PacketQueue**
  Buffer de almacenamiento de los paquetes de la cola. Normalmente FIFO.
- EDTrace *NsObject**
  Destino para trazar eventos al que se envían los paquetes descartados de forma temprana.
- traceType *char*
  Tipo de traza.
- tchan_ *Tcl_Channel*
  Canal (usualmente fichero) en el que se escribirán los resultados de las trazas. Vinculado a OTcl.
- greenArrivals_ *int*
  Número de paquetes llegados a la cola de Green.
- greenAllDrops_ *int*
  Número total de paquetes descartados por Green.
- greenEarlyDrops_ *int*
  Número de descartes tempranos por Green.
- greenForcedDrops_ *int*
  Número de descartes forzados por Green.
- greenDepartures_ *int*

Número de paquetes que salen de la cola de Green.

- factor *double*
  Parámetro gamma.
- util_estimate *double*
  Fracción del ancho de banda utilizada en un determinado intervalo.
- loss_estimate *double*
  Fracción de paquetes perdidos en un determinado intervalo.
- kbytes_sent *double*
  Cantidad de bytes enviados.
- queue_drops *double*
  Paquetes descartados por desbordamiento de la cola en un intervalo.
- green_drops *double*
  Paquetes descartados por Green en un intervalo.
- next_estimate_ *double*
  Siguiente estimación.
- N_estimate_ *int*
  Número de estimaciones.
- active_flows_ *int*
  Número de flujos activos.
- idmaps_ *int*
  Para utilizar IDMaps o no.
- bw_ *double*
  Ancho de banda del enlace de salida al cual está asociada la cola.
- delay_ *double*
  Retardo de propagación del enlace de salida.
- mss_ *int*
  Tamaño del segmento máximo del paquete. c_ *double*
  Constante utilizada en la ecuación de probabilidad.
- flowTable *flowState*
  Struct de flujos.

Métodos

<u>-Públicos:</u>

- Green()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.
- ~Green()
  Método desconstructor. Se libera el espacio requerido por

Green.

-Protegidos:

- command() *int*
  Método heredado de la clase *ObjectTcl* y a través del cual se le pueden enviar mensajes desde la interfaz Tcl a esta clase.
- enque() *void*
  Método que se ejecuta al llegar un paquete entrante.
- deque() *Packet**
  Elimina de la cola el paquete correspondiente en caso de descarte y lo devuelve para el caso en el que se necesite trazar los descartados.
- trace() *void*
  Permite la selección y activación de los atributos que se van a trazar.
- processPkt() *int*
  Método utilizado por Green para procesar los paquetes.

Blue

Estructura: edp_blue

- mean_pktsize *int*
  Tamaño medio en *bytes* de los paquetes que se alojan en la cola.
- bytes *int*
  Valor booleano que indica si la cola está medida en bytes.
- wait *int*
  Valor booleano que indica si espera entre paquetes descartados.
- setbit *int*
  Valor booleano que indica si se usa el mecanismo de ECN en lugar del descarte temprano.
- gentle *int*
  Valor booleano que indica si se incrementa lentamente la probabilidad de descarte cuando el tamaño medio de la cola supera el umbral máximo.
- th_min *double*
  Umbral mínimo del tamaño medio de la cola.
- th_max *double*
  Umbral máximo del tamaño medio de la cola.

- max_p_inv *double*
  Inverso de la probabilidad máxima, 1/max_p.
- q_w *double*
  Peso de la cola asociado a la muestra del tamaño de la cola.
- ptc *double*
  Constante de tiempo del paquete medida en paquetes/segundo. Se calcula en función de los parámetros suministrados por el usuario.

Estructura: edv_blue

- v_ave *TracedDouble*
  Tamaño medio de la cola calculado con el algoritmo EWMA.
- v_prob1 *TracedDouble*
  Probabilidad de descarte temprano de los paquetes entrantes.
- v_slope *double*
  Utilizado en el cálculo del tamaño medio de la cola. Actualmente está obsoleto.
- v_prob *double*
  Probabilidad de descarte de paquetes.
- v_a, v_b *double*
  Intervienen en el cálculo de la probabilidad de descarte. v_prob = v_a * v_ave + v_b
- v_c, v_d *double*
  Utilizados para el modo “gentle”.
- count, count_bytes *int*
  Número de paquetes y número de bytes llegados desde el último descarte.
- old *int*
  Valor booleano que toma el valor 0 cuando el promedio de la cola excede primero el umbral.

Clase: Blue

Responsabilidad:

Atributos:

- Protegidos:

- link_ *LinkDelay**
  Enlace asociado al nodo para esta cola de salida.
- fifo_ *int*
  Valor booleano que indica si la cola será FIFO o no.
- q_ *PacketQueue**
  Buffer de almacenamiento de los paquetes de la cola. Normalmente FIFO.
- bcount_ *int*
  Contador de bytes.
- qib_ *int*
  Valor booleano que indica si la cola se debe medir en *bytes* en lugar de paquetes. Vinculado a OTcl.
- de_drop_ *NsObject**
  Destino al que se envían los paquetes descartados, en caso de haberlo.
- curq_ *TracedInt*
  Tamaño actual de la cola. Vinculado a OTcl.
- drop_tail_ *int*
  Valor booleano que indica si se descarta el último paquete.
- drop_front_ *int*
  Valor booleano que indica si se descarta el primer paquete.
- drop_rand_ *int*
  Valor booleano que indica si el paquete se descarta de forma aleatoria.
- ns1_compat_ *int*
  Utilizado para la compatibilidad con ns-1.
- edp_ *edp_blue*
  Struct de parámetros de entrada a BLUE. Detallado anteriormente.
- doubleq_, dqthresh_ *int*
  Utilizados para experimentos con prioridad para paquetes pequeños.
- idle_ *int*
  Valor booleano que indica si la cola está libre.
- idletime_ *double*

Tiempo que lleva libre la cola.

- edv_ *edv_blue*
  Struct de variables características que ha de mantener BLUE. Detallado anteriormente.
- first_reset_ *int*
  Valor booleano que indica si ya se ha ejecutado al menos una vez el método *reset()*.

* nota: La clase BLUEQueue también dispone de los atributos protegidos: EDTrace, traceType y tchan_; los cuales ya se comentaron en el apartado atributos protegidos de la clase Green. Y además de los atributos descritos, tiene también los atributos protegidos de tipo *double*: decrement_, increment_, iholdtime_, dholdtime_, ifreezetime_, dfreezetime_ y bbw_; y which_ de tipo *int*.

Métodos

-Públicos:

- BLUEQueue()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.

-Protegidos:

- pickPacketForECN() *Packet**
  Devuelve el paquete que hay que marcar en el caso de que el ECN esté activado.
- pickPacketToDrop() *Packet**
  Devuelve el paquete que hay que descartar, en el caso de que se haya decidido que se debe hacer tal cosa.
- reset() *void*
  Reinicializa todas la variables del controlador.
- run_estimator() *void*
  Actualmente se encuentra obsoleto. Sólo está incluido para compatibilidad con versiones anteriores.
- estimator() *double*
  Calcula el tamaño medio de la cola.
- drop_early() *int*
  Decide si el paquete entrante ha de ser marcado, descartado, o encolado sin marcar.
- modify_p() *double*
  Realiza periodos uniformes en lugar de periodos geométricos.
- calculate_p() *double*
  Método que recalcula cada periodo de muestreo la probabilidad de descarte de un paquete entrante.

- decrement_pmark() *void*
  Método empleado para calcular la probabilidad de descarte.
- increment_pmark() *void*
  Método empleado para calcular la probabilidad de descarte.
- print_edp() *void*
  Método que imprime por pantalla algunos datos de la simulación.
- print_edv() *void*
  Método que imprime por pantalla algunos datos de la simulación.

*nota: BLUEQueue también dispone de los métodos protegidos: command(), enque(), deque() y trace(); los cuales ya se comentaron en el apartado métodos protegidos de la clase Green.

RED

Estructura: edp_red

- idle_pktsize *int*
  Tamaño medio del paquete utilizado durante los tiempos de inactividad.
- th_min_pkts *double*
  Umbral mínimo del tamaño medio de la cola mantenido siempre en paquetes.
- th_max_pkts *double*
  Umbral máximo del tamaño medio de la cola mantenido siempre en paquetes.
- max_p_inv *double*
  Parámetro definido para controlar la probabilidad de descarte de un paquete. Esta probabilidad se encuentra entre 0 y 1/max_p, siendo max_p la probabilidad máxima. Si se trata de ARED, la max_p_inv inicial.
- mark_p *double*
  Utilizado para marcar los paquetes correspondientes cuando p < mark_p; y para descartar los paquetes correspondientes cuando p > mark_p.
- use_mark_p *int*
  Valor booleano que indica si se utiliza mark_p solamente para decidir cuándo se descarta.

- adaptive *int*
  Valor booleano que indica si se utiliza RED por defecto o ARED.
- cautious *int*
  Valor booleano que indica si se utiliza RED por defecto o si se usa para no marcar/descartar cuando la cola instantánea es mucho inferior a la media.
- delay *double*
  Retardo del enlace.

*nota: En la estructura edp_red también tenemos los atributos de tipo *double*: alpha, beta, interval, targetdelay, top y bottom; y feng_adaptive de tipo *int*. Aunque todos ellos hacen referencia a ARED.
Además de los atributos descritos, también dispone de los siguientes: mean_pktsize, bytes, wait, setbit, gentle, th_min, th_max, q_w y ptc; los cuales ya han sido comentados en el apartado estructura edp_blue de BLUE.

Estructura: edv_red

- cur_max_p *TracedDouble*
  Probabilidad máxima actual.
- lastset *double*
  Última vez adaptada. Utilizado para ARED.
- Status *enum {}*
  Atributo de tipo enumerado.
- status *Status*
  Atributo de tipo enumerado utilizado en Feng´s para ARED.

*nota: La estructura edv_red también dispone de los atributos: v_ave y v_prob1 de tipo *TracedDouble*; v_slope, v_prob, v_a, v_b, v_c y v_d de tipo *double*; y count, count_bytes y old de tipo *int*. Todos ellos descritos en el apartado estructura edv_blue de BLUE.

Clase: REDQueue

Responsabilidad:

Atributos:
- Protegidos:

- summarystats_ *int*
  Valor booleano que indica si se imprime por pantalla el verdadero tamaño medio de la cola.
- edp_ *edp_red*

Struct de parámetros de entrada a RED. Detallado anteriormente.

- edv_ *edv_red*
  Struct de variables características que ha de mantener RED. Detallado anteriormente.

*nota: La clase REDQueue también dispone de los atributos protegidos: link_, fifo_, q_, bcount_, qib_, de_drop_, EDTrace, traceType, tchan_, curq_, drop_tail_, drop_front_, drop_rand_, ns1_compat_, doubleq_, dqthresh_, idle_, idletime_ y first_reset_; los cuales ya se comentaron en el apartado atributos protegidos de las clases Green y BLUEQueue.

Métodos

-Públicos:

- REDQueue()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.

-Protegidos:

- initParams() *void*
  Inicializa los atributos de RED.
- initialize_params() *void*
  Inicializa los parámetros de RED.
- updateMaxP() *void*
  Actualiza la probabilidad máxima para mantener el tamaño medio de la cola dentro del intervalo objetivo. Se utiliza sólo para ARED.
- updateMaxPFeng() *void*
  Actualiza la probabilidad máxima siguiendo el código de Feng et al. Se utiliza sólo para ARED.
- calculate_p_new() *double*
  Calcula la probabilidad de descarte.
- reportDrop() *void*
  Actualmente en desuso.
- print_summarystats *void*
  Método que muestra por pantalla un resumen estadístico.

*nota: REDQueue también dispone de los métodos protegidos: command(), enque(), pickPacketForECN(), pickPacketToDrop(), deque(), reset(), run_estimator(), estimator(), drop_early(), modify_p(), calculate_p(), trace(), print_edp() y print_edv(); los cuales ya se comentaron en el apartado métodos protegidos de las clases Green y BLUEQueue.

FRED

Estructura: edp_fred

- minq *double*
  Utilizado para FRED.

*nota: En la estructura edp_fred también tenemos los atributos: mean_pktsize, bytes, wait, setbit, gentle, th_min, th_max, max_p_inv, q_w y ptc; los cuales ya han sido comentados en el apartado estructura edp_blue de BLUE.

Estructura: edv_fred

- avgcq, maxq *double*
  Utilizados para FRED.
- Nactive *int*
  Utilizado para FRED.

*nota: La estructura edv_fred también dispone de los atributos: v_ave, v_prob1, v_prob, v_a, v_b, v_c, v_d, count, count_bytes y old. Todos ellos descritos en el apartado estructura edv_blue de BLUE.

Estructura: flowv

- state *int*
  Estado del flujo.
- fid *int*
  Identificador del flujo.
- qlen *int*
  Cantidad instantánea de paquetes del flujo en la cola.
- strike *int*
  Número de ocasiones en que el flujo no responde a la notificación de congestión.

Clase: FREDQueue

Responsabilidad:

Atributos:

- Protegidos:

- edp_ *edp_fred*
  Struct de parámetros de entrada a FRED. Detallado anteriormente.

- edv_ *edv_fred*
  Struct de variables características que ha de mantener FRED. Detallado anteriormente.
- flowv_ *flowv*
  Struct de flujos.

*nota: La clase FREDQueue también dispone de los atributos protegidos: link_, fifo_, q_, bcount_, qib_, de_drop_, EDTrace, traceType, tchan_, curq_, drop_tail_, drop_front_, drop_rand_, ns1_compat_, idle_, idletime_, first_reset_ y which_; los cuales ya se comentaron en el apartado atributos protegidos de las clases Green y BLUEQueue.

Métodos

-Públicos:

- FREDQueue()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.

-Protegidos:

- flowlkup() *flowv**
  Método que localiza un determinado flujo.
- allocflowstate() *flowv**
  Método que maneja el estado del flujo.
- freeflowstate() *void*
  Método que libera el estado del flujo.

*nota: FREDQueue también dispone de los métodos protegidos: command(), enque(), pickPacketForECN(), pickPacketToDrop(), deque(), reset(), estimator(), drop_early(), modify_p(), calculate_p(), trace(), print_edp() y print_edv(); los cuales ya se comentaron en el apartado métodos protegidos de las clases Green y BLUEQueue.

SRED

Estructura: zombie

- fid *int*
  Identificador del flujo.
- count *int*
  Número de paquetes del flujo en esta estructura zombi.
- timestamp *double*
  Tiempo del flujo.

Clase: SREDQueue

Responsabilidad:

Atributos:

- Protegidos:

- full_sred_ *int*
  Valor booleano que indica si se utiliza SRED completo o SRED simple.
- zombielist *zombie*
  Struct de zombis.
- listsize_ *int*
  Tamaño de la lista.
- p_overwrite_ *double*
  Probabilidad de sobrescribir un zombi.
- p_t_ *double*
  Frecuencia de choque.
- p_max_ *double*
  Probabilidad máxima de descarte.
- alpha_ *double*
  Utilizado para calcular la frecuencia de choque.
- drop_tgt_ *NsObject**
  Destino al que se envían los paquetes descartados, en caso de haberlo.
- drop_trace_ *NsObject**
  Destino para trazar eventos al que se envían los paquetes descartados de forma temprana.

*nota: La clase SREDQueue también dispone de los atributos protegidos: link_, q_, bcount_, qib_, traceType, tchan_, curq_ y which_; los cuales ya se comentaron en el apartado atributos protegidos de las clases Green y BLUEQueue.

Métodos
-Públicos:

- SREDQueue()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.

-Protegidos:

- calc_psred() *double*
  Calcula la probabilidad de descarte.
- calc_simple_pzap() *double*
  Método que calcula la probabilidad de descarte para SRED simple.
- calc_full_pzap() *double*
  Método que calcula la probabilidad de descarte para SRED completo.

*nota: SREDQueue también dispone de los métodos protegidos: command(), enque(), deque(), reset() y trace(); los cuales ya se comentaron en el apartado métodos protegidos de las clases Green y BLUEQueue.

PID3

Estructura: edp_pid3

- typepid *int*
  Tipo de PID usado.
- kp, ki, kd *double*
  Constantes proporcional, integral y derivativa del controlador.
- ti, td *double*
  Tiempos integral y derivativo, se usan en el caso de estar activada zn.
- b *double*
  Peso de la referencia para el PID tipo D.
- n *double*
  Parámetro N del PID tipo D.
- zn *int*
  Valor booleano que indica si estamos usando como parámetros los tiempos integral y derivativo en vez de las respectivas constantes.
- with_ave *int*
  Valor booleano que indica si se controla el valor medio de

la cola en lugar del instantáneo (EWMA-PID).

- ewma_weight *double*
  Peso del EWMA en el cálculo de la cola media.
- t *double*
  Periodo de muestreo de la señal.
- qref *double*
  Referencia para el controlador PID en torno al cual ha de estabilizarse la cola.

*nota: En la estructura edp_pid3 también tenemos los atributos: mean_pktsize, bytes y setbit; los cuales ya han sido comentados en el apartado estructura edp_blue de BLUE.

Estructura: edv_pid3

- v_prob *TracedDouble*
  Probabilidad de descarte temprano de los paquetes entrantes.
- countini *int*
  Variable actualmente en desuso.
- qold, qoldold *int*
  Tamaño de la cola del periodo anterior y de hace dos periodos respectivamente.
- eold, eoldold *double*
  Error de la señal del periodo anterior y de hace dos periodos respectivamente.
- d_term_old, i_term_old *double*
  Términos derivativo e integral del periodo anterior para el PID tipo D.
- earlydrops, forceddrops *TracedInt*
  Número de paquetes descartados de forma prematura/forzada desde el inicio de la simulación.

*nota: La estructura edv_pid3 también dispone de los atributos: count y count_bytes. Ambos descritos en el apartado estructura edv_blue de BLUE.

Clase: PID3Queue

Responsabilidad:

Atributos:

- Protegidos:

- pp *int*
  Actualmente en desuso.
- CalcTimer *PID3CalcTimer*
  Temporizador asociado a esta cola. Cuando finaliza su tiempo llama al método calculate_p().
- aveq_ *TracedDouble*
  Tamaño medio de la cola calculada con el algoritmo EWMA. Vinculada a OTcl.
- np_ *int*
  Atributo que ayuda a trazar newpackets_.
- newpackets_ *TracedInt*
  Número de paquetes llegados a la cola en este periodo de muestreo. Vinculado a OTcl.
- totalarrivals_ *TracedInt*
  Número total de paquetes llegados a la cola, muestreados en cada periodo.
- edp_ *edp_pid3*
  Struct de parámetros de entrada al controlador. Detallado anteriormente.
- edv_ *edv_pid3*
  Struct de variables características que ha de mantener este tipo de cola. Detallado anteriormente.

*nota: La clase PID3Queue también dispone de los atributos protegidos: link_, fifo_, q_, qib_, de_drop_, EDTrace, traceType, tchan_, curq_ y first_reset_; los cuales ya se comentaron en el apartado atributos protegidos de las clases Green y BLUEQueue.

Métodos

-Públicos:

- PID3Queue()
  Método constructor. Se inicializan las variables y se realizan las ligaduras con OTcl.

-Protegidos:

*nota: PID3Queue dispone de los métodos protegidos: command(), enque(), pickPacketForECN(), pickPacketToDrop(), deque(), reset(), drop_early(), calculate_p() y trace(); los cuales ya se comentaron en el apartado métodos protegidos de las clases Green y BLUEQueue.

Parte III

CAPITULO 6

6. Diseño de los experimentos

6.1 Métricas

Longitud de la cola

La mayoría de los algoritmos AQM que se van a analizar tienen como objetivo estabilizar el tamaño de la cola del *router* en torno a un valor de referencia o acotarlo entre dos valores prefijados. Por este motivo, es necesario el uso del tamaño de la cola a lo largo del tiempo como valor de medida, para comprobar si el resultado obtenido se ajusta a las expectativas y poder observar cómo ha sido la reacción del sistema afectado por el algoritmo. Se proporcionarán el tamaño instantáneo de la cola y el tamaño medio de la cola:

Tamaño instantáneo de la cola: Es el número de paquetes que están ocupando la cola en cada instante de tiempo. Suelen ser valores muy oscilatorios observándose como una señal con bastante ruido.

Tamaño medio de la cola: Este valor también es calculado para cada instante de tiempo. La forma de hacerlo es mediante el uso de una técnica de media móvil, en concreto la denominada EWMA (*Exponentially Weighted Moving Average*), que otorga un peso a cada muestra y éste decrece exponencialmente según la muestra se aleja en el tiempo de la actual, concediendo más importancia así a las observaciones recientes. La ocupación media de la cola se estima cada vez que llega un paquete. El algoritmo es el siguiente:

$$media_k=(1-w_q)*media_{k-1}+w_q*q$$

- $media_k$: es el tamaño medio de la cola estimado en el instante k.
- w_q: es un factor de peso que indica en qué medida se da más importancia a las muestras actuales.
- q: es el tamaño real de la cola en el momento de la muestra.

Para w_q se ha elegido el valor 0.002, dicho valor filtra de forma moderada las fluctuaciones de la cola.

Probabilidad de descarte

Muchos de los controladores AQM se basan en la generación de una probabilidad de descarte para cada paquete que llega. Desde el punto de vista de control, esta probabilidad constituye la salida del controlador y la entrada al sistema a estabilizar. Por todo esto, parece lógico que uno de los valores principales al que se debe prestar atención sea precisamente éste. Interesa que la probabilidad se mantenga lo más baja posible, evitando pérdidas de paquetes con sus correspondientes reenvíos.

QACD (Quadratic Average of Control Deviation)

Este índice da una idea de cuánto se aleja de media el tamaño real de la cola del que se había fijado como referencia. Por tanto, sólo es válido para aquellos controladores que fijan un valor en torno al cual estabilizar la cola. Desde el punto de vista de control, la evolución del QACD da una idea del funcionamiento en estado estacionario del sistema. Es deseable mantener este índice lo más bajo posible. Valores altos, indicarían que el sistema se encuentra estabilizado lejos de la referencia, o bien, que las oscilaciones en torno a ella son amplias.

$$S_e = \sqrt{\frac{1}{M+1}\sum_{i=0}^{M} e_i^2} = \sqrt{\frac{1}{M+1}\sum_{i=0}^{M} (q_i - q_{ref})^2}$$

Tasa de paquetes perdidos

Uno de los objetivos de las técnicas AQM es evitar que se penalicen los tráficos a ráfagas. Para ello es importante mantener una tasa de paquetes perdidos estable impidiendo que se pierdan las ráfagas completas cada vez que llegan. La tasa de paquetes perdidos se calcula de la siguiente forma:

$$T_p = \frac{nº\ paquetes\ llegados}{nº\ paquetes\ perdidos}$$

Jain´s fairness index

Este índice actúa como evaluación de la imparcialidad del controlador respecto a la distribución del ancho de banda entre los diferentes flujos que atraviesan el enlace.

$$jfi = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

- n: es el número de flujos que atraviesa el enlace
- x_i: es la tasa de transferencia del flujo i

Este índice varía entre 0 y 1, alcanzando el máximo valor cuando todos los flujos comparten el mismo ancho de banda. Como característica reseñable cabe decir que es independiente del número de flujos y de las unidades en las que se mide la tasa de transferencia.

Utilización del enlace

Se refiere a la utilización del enlace en el cuello de botella. En los periodos largos de congestión, la utilización debería aproximarse siempre al máximo (la unidad), debido a que la tasa de llegada de paquetes es mayor de la que el enlace puede transportar. Cuando en esos momentos la cola del *router* se vacía, ya sea por la disminución del tráfico o por una congestión de la cola muy agresiva, la capacidad del enlace queda desaprovechada.

$$utilizacion = \frac{bytes\_enviados}{C \cdot T_s}$$

- bytes_enviados: número de bytes transmitidos al enlace en el periodo de muestreo.
- C: capacidad del enlace (bytes/s).
- T_s: periodo de muestreo.

Media, Varianza y desviación típica

Las métricas anteriores van variando a lo largo del tiempo, a medida que avanza la simulación. De algunas de ellas es posible calcular la media, la varianza y la desviación típica.

6.2 Topología dumbbell

La topología de red se define como la forma en la que están conectados los elementos de una subred de conmutación, es decir, hace referencia a la disposición geométrica resultante del enlazado entre los nodos y las líneas.

Para poder observar de una forma clara los efectos que se producen en el nivel de carga de tráfico al emplear los algoritmos de control, se ha optado por la topología más sencilla para los tres primeros experimentos, ya que es la que más información reporta en los casos en los que se estudia la congestión en una red. Mientras, en el cuarto experimento, se ha utilizado una topología con múltiples cuellos de botella puesto que es una configuración que se asemeja más a la realidad.

Dicha topología sencilla es la que normalmente se conoce como topología *dumbbell*. *Dumbbell* es una palabra inglesa que se traduce como "mancuerna" y hace alusión a la forma que toma su geometría al verla representada sobre el papel.

Como se puede apreciar en la **figura 6.1** este tipo de disposición está formado únicamente por un enlace entre dos *routers* que actúa como cuello de botella de las conexiones que se establecen entre los *hosts* conectados a cada uno de ellos. Normalmente los *hosts* de un lado del cuello de botella actúan como fuentes de tráfico y los situados al otro lado, como sumideros (sin olvidar que también generan los ACK). En el *router* donde se genera la conexión, es en el que hay que situar el algoritmo de AQM, asociado a la cola del cuello de botella. El ancho de banda del enlace que conforma el cuello de botella ha de ser menor que el de los otros enlaces para que se produzca la congestión.

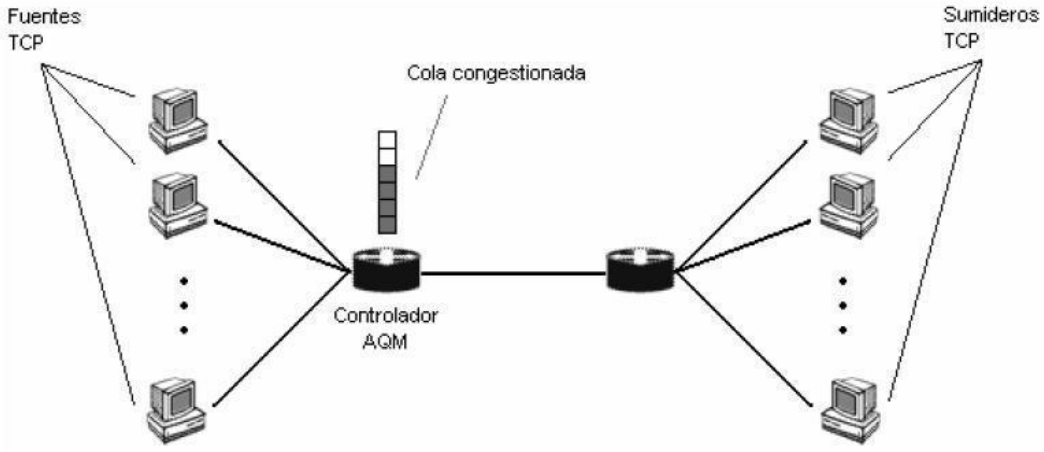


Figura 6.1. Topología *Dumbbell*.

6.3 Tipos de experimentos

6.3.1 Experimento I

Este experimento (y el segundo, similar a este primero pero ampliando el número de flujos al doble) se basa en el realizado en el articulo *"A control theoretical approach to congestion control of TCP/AQM networks"* [ALV3] y, con el cual, podrán compararse resultados una vez hecha la simulación. Se caracteriza por un entorno en el que el número de flujos TCP es pequeño, al igual que el ancho de banda del enlace (250 paquetes/segundo).

Como se puede observar en la **figura 6.2,** esta versión de la topología *dumbbell* está formada por 40 *hosts* que actúan como orígenes de tráfico, conectados virtualmente uno a uno con otros 40 equipos que hacen de sumideros. Los equipos transmiten datos utilizando como protocolo de aplicación FTP, asentado sobre TCP en su versión *New Reno*.

La capacidad de los enlaces (C) que conectan los *hosts* con los nodos es de 10 Mb/s y su tiempo de propagación (Tp) es de 25 ms. En el cuello de botella se aplica el algoritmo AQM con un tamaño de *buffer* de 600 paquetes, un tiempo de propagación de 60 ms y un ancho de banda, en este caso, de 2 Mb/s. En el resto de enlaces se utiliza *Drop Tail* como técnica para gestionar la cola.

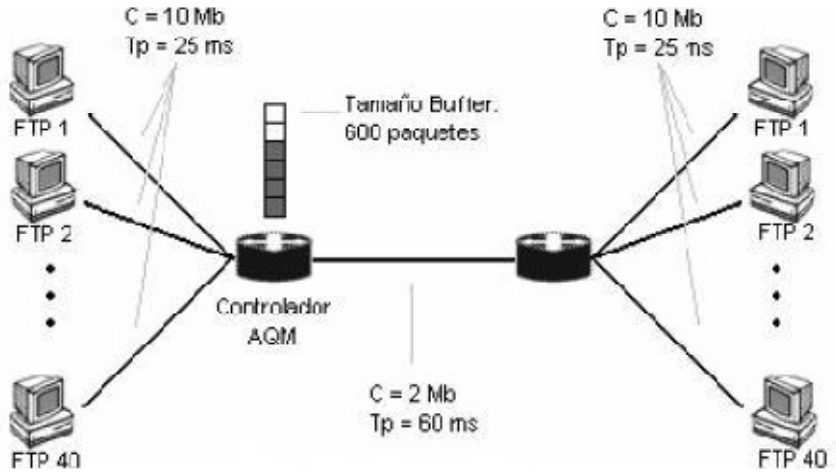


Figura 6.2. Topología experimento I y II.

6.3.1 Experimento II

Este segundo experimento se lleva a cabo prácticamente con los mismos datos que el anterior, salvo el número de flujos FTP que intervienen en la simulación. En esta ocasión,

se incrementa el número de terminales a 80, doblando así la cantidad inicial de flujos FTP, con el objetivo de comprobar el funcionamiento de los algoritmos empleados al duplicarse dicho número.

En ambos experimentos se utilizan ocho algoritmos de control de congestión y se comparan los resultados entre ambas topologías y entre los algoritmos seleccionados, los cuales son:

- *DropTail*
- RED
- ARED
- FRED
- SRED
- GREEN
- BLUE
- PID3

6.3.2 Experimento III

El siguiente experimento también se basa en la misma topología que los dos algoritmos anteriores, aunque con valores diferentes.

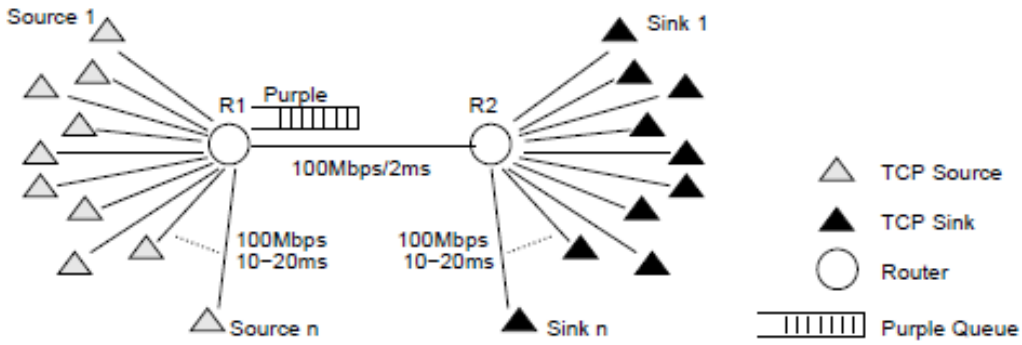


Figura 6.3.Topología del experimento III.

Consta de un enlace congestionado con una conexión de 100Mbps y un retado de 2ms. El número de flujos varía para los experimentos, cuyos valores se fijan en 40,60 y 80 conexiones. Dichos enlaces tienen una conexión de 100 Mbps y un retardo variable entre 10 y 20 ms. El tamaño del buffer de la cola es de 660 paquetes y cada simulación tiene una duración de 100 segundos.

La elección de está simulación se debe a que se pueden comparar los resultados obtenidos con los contenidos en el articulo "PURPLE: *Predictive Active Queue Management Utilizing Congestion Information*" [PLE1]. En dicho artículo, se realiza la comparación de tres algoritmos con la topología explicada anteriormente. No se cuenta con la disponibilidad del algoritmo PURPLE, pero se desarrollará la comparación con los algoritmos RED y ARED.

6.3.2 Experimento IV

Para este experimento se ha optado por la topología del articulo *"Design, validation and experimental testing of a robust AQM control"* [MAN]. Como se puede ver en la figura 6.4, está topología consta de múltiples cuellos de botella.

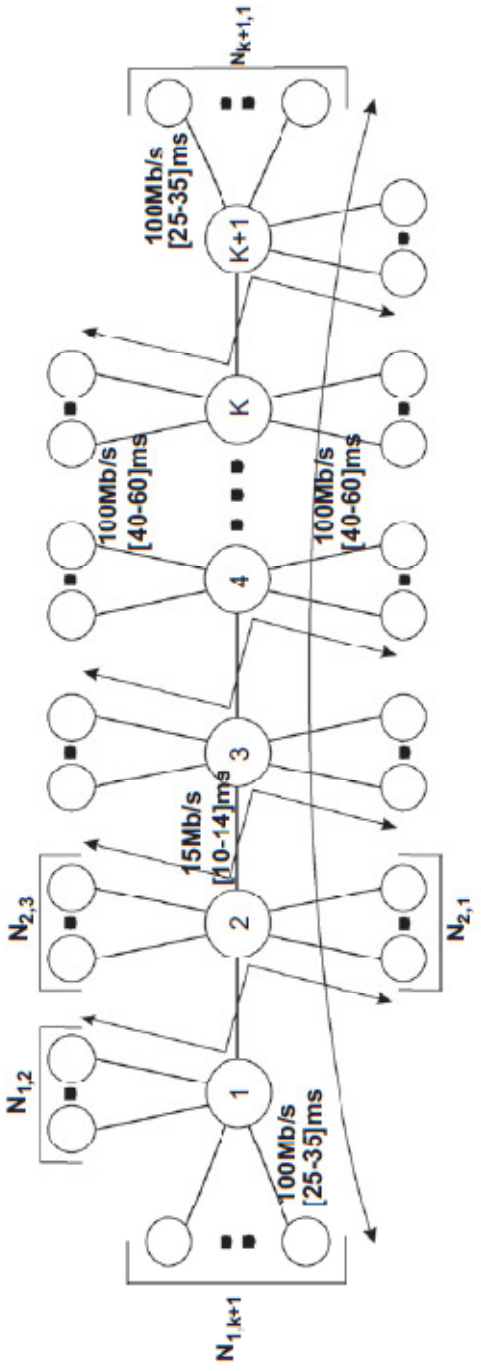


Figura 6.4. Topología con múltiples cuellos de botella.

Esta topología consta de varios *routers* conectados entre sí, por los cuales atraviesan paquetes de unos nodos conectados extremo a extremo y, además, cada *router* es atravesado también por los paquetes que cruzan de arriba abajo (pasan del nodo k al nodo k+1) como se puede apreciar en la **figura 6.4**.

Este experimento se realiza con los algoritmos RED, REM y PI, visualizando su comportamiento cuando hay varias colas congestionadas, llegando a ser un caso más real que los anteriores experimentos desarrollados.

En dicho experimento se emplea un número de conexiones entre los extremos de 30 flujos, siendo el mismo número de flujos los que atraviesan de arriba abajo los *routers*. El número colas congestionadas será de 5. Los datos serán:

- Entre los *routers* habrá una conexión de 15 Mbps, con un retardo variable de entre 10 y 14 ms.
- En los nodos extremos la capacidad será de 100 Mbps, con un retardo variable de entre 25 y 35 ms.
- En los nodos que cruzan los *routers* la conexión será de 100 Mbps también, pero el retardo variará entre 40 y 60 ms.

Los resultados obtenidos serán analizados y comparados con los que se encuentran en el artículo.

6.3.2 Experimento V

La topología de este experimento se corresponde con la siguiente figura.

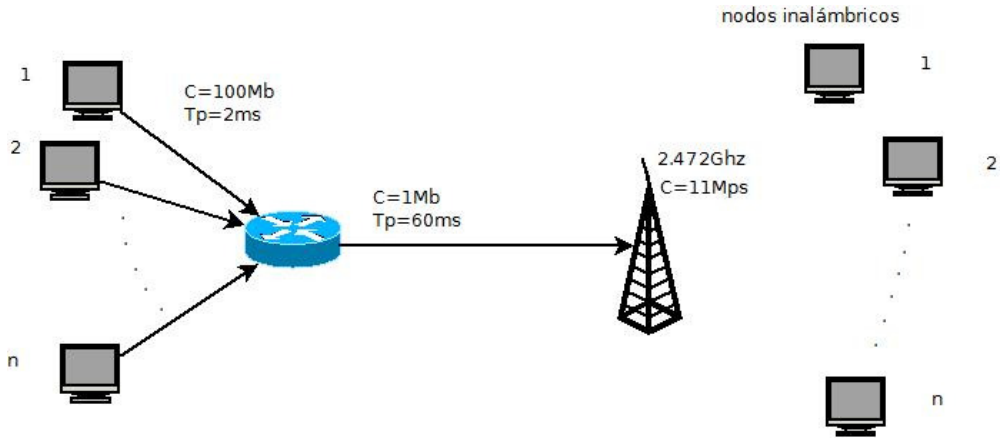


Figura 6.5. Topología experimento V

Como se puede apreciar en la figura anterior, la topología empleada se asemeja a la topología *dummbell,* pero en esta ocasión consta de terminales inalámbricos. Está formada por 100 *hosts* que actúan como orígenes de tráfico, conectados virtualmente uno a uno con otros 100 equipos que hacen de sumideros (enlaces inalámbricos). Los equipos generan tráfico empleando los protocolos TCP (*WestwoodNR*) y UDP.

La conexión entre el *router* y la estación base tendrá una capacidad de 1 Mb con un retardo de 60 ms. El tamaño de la cola será de 350 paquetes, mientras la conexión de los nodos al *router* tiene una capacidad de 100 Mb/s y un retardo de 2 ms. Hay que añadir que al usar una WLAN, la estación base transmite en la banda de las microondas a 2.472 GHz, empleando el canal 13 del estándar IEEE 801.11 b, con una tasa de transferencia máxima de 11 Mb/s. En cualquier caso, los enlaces inalámbricos permanecerán en la misma posición ya que no se ha definido un patrón de movimiento.

CAPITULO 7

7. Resultado de los experimentos

7.1 Experimento I

Este experimento, basado en el articulo "*A control theoretical approach to congestion control of TCP/AQM networks*" [ALV3], contempla una topología y una situación de tráfico sencilla, que permitirá observar de forma simple las características de cada uno de los controladores.

Para los algoritmos BLUE y FRED, los parámetros utilizados en este caso han sido:

- *minth* = 70
- *maxth* = 120
- *wq* = 0,002

Los parámetros usados para el algoritmo RED son los siguientes:

- *minth* = 70
- *maxth* = 120
- *pmax* = 0,1 (utilizado también para SRED)
- *wq* = 0,002

Se ha dicho que las técnicas AQM basadas en control, en este caso el PID3, funcionan muestreando la señal. Para ellas habrá que definir un tiempo de muestreo. Para calcularlo, se ha utilizado la fórmula que proporcionan en [MAR2]:

$$\left.\begin{array}{l}\tau_1 = \dfrac{R_0^2 C}{2N} \\ \tau_2 = R_0\end{array}\right\} \; T_s = 0.2\sqrt{\tau_1^2 + \tau_2^2}$$

Para este primer experimento resulta un valor de *Ts* = 0,2013 s.

Los otros parámetros para el PID3 son los siguientes:

- *Kp* = -0,0022
- *Ti* = 1,8207
- *Td* = 0,32

- *wq* = 0,002
- *qref* = 120 (Valor de la referencia, se mide en paquetes)

Para visualizar los resultados del experimento se representarán las graficas de las métricas correspondientes al tamaño de la cola, utilización del enlace, probabilidad de descarte y tasa de pérdidas para cada controlador empleado. Se comentarán los resultados obtenidos.

Tamaño de la cola

La **figura 7.1** muestra el tamaño de la cola resultante de aplicar el algoritmo *DropTail*.

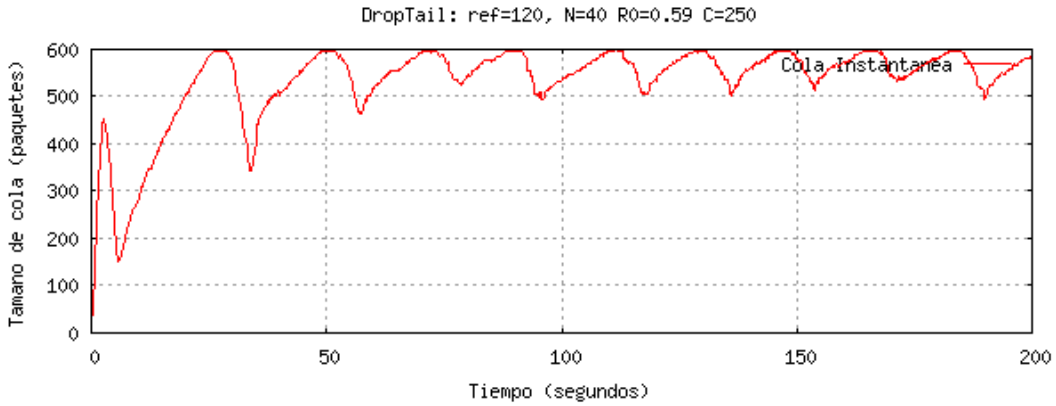


Figura 7.1. Cola instantánea *DropTail*

Como es sabido, el algoritmo funciona descartando paquetes sólo cuando la cola se llena, con lo cual el aspecto de la gráfica debería ser continuo y no con dientes de sierra. Esta forma se debe al efecto de la sincronización global entre flujos que produce el fenómeno de las colas llenas. Mientras la cola no está llena, no se produce ningún descarte de paquetes y los agentes TCP no tienen notificación de congestión alguna, por lo que van aumentando progresivamente el tamaño de su ventana en cada RTT. El problema surge porque *Drop Tail* sólo es capaz de notificar la congestión una vez la cola se ha llenado. Cuando la cola llega a su tope, se produce una oleada de descartes, lo que provoca la presencia de oscilaciones tan grandes.

Estas oscilaciones son muy perjudiciales para el funcionamiento de los agentes TCP debido a que su mecanismo de detección de paquetes perdidos se fundamenta en una estimación del RTT en base a las muestras obtenidas en tiempos anteriores. Si el RTT varía mucho, esta estimación puede no ser adecuada y provocar retransmisiones innecesarias de paquetes en caso de ser una estimación errónea por defecto o provocar

esperas demasiado largas antes de la retransmisión de un paquete perdido, produciendo un bajo rendimiento en las aplicaciones.

El tiempo de encolado es proporcional al tamaño de la cola que se mantiene en cada momento y es una parte importante del RTT. Por este motivo, las oscilaciones en la cola hacen que el RTT varíe con ellas.

Por otra parte, el uso de este algoritmo también tiene como contrapartida que las colas se mantienen con un nivel alto de ocupación, lo que provoca que el RTT sea también grande (además de variable). Normalmente en las aplicaciones que usan TCP para comunicarse, el RTT es un factor que se quiere reducir, por lo que generalmente se desea que los tamaños de las colas se mantengan en un nivel bajo.

A continuación se mostrarán las gráficas del resto de algoritmos empleados:

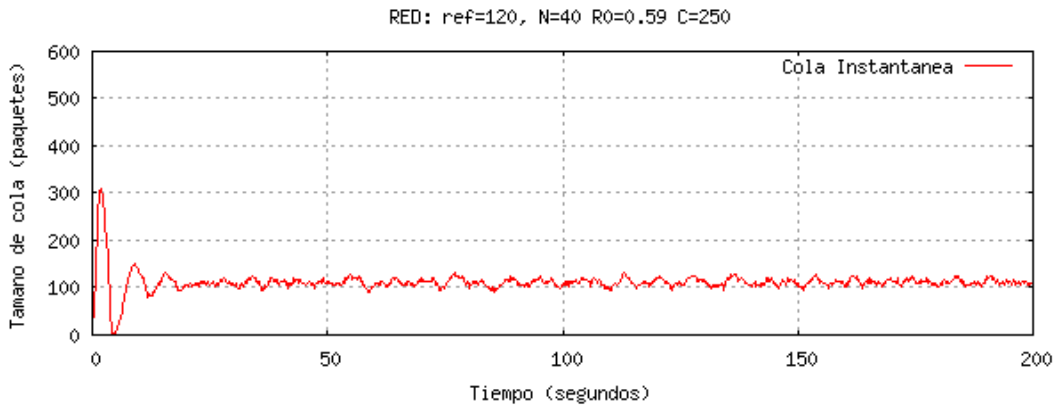


Figura 7.2. Cola instantánea RED

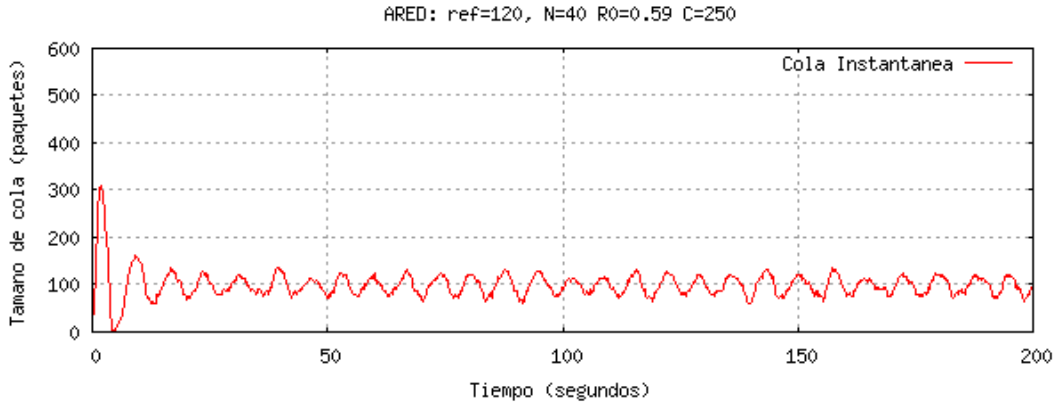


Figura 7.3. Cola instantánea ARED

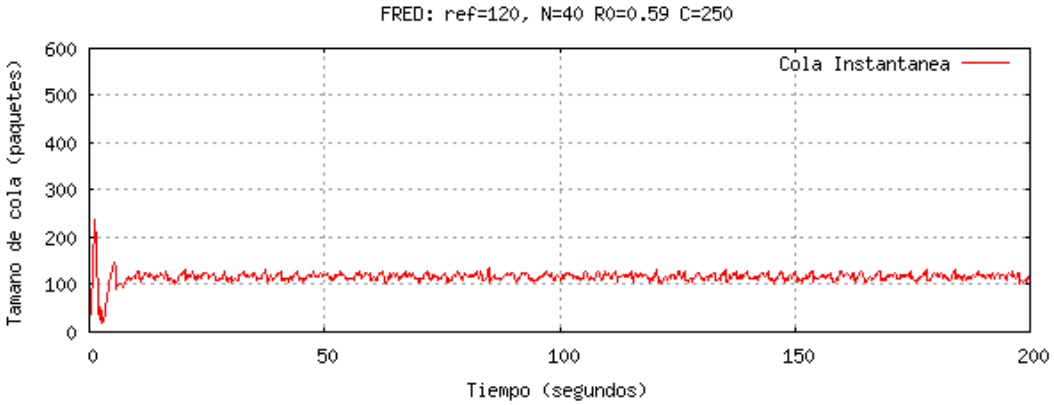


Figura 7.4 Cola instantánea FRED

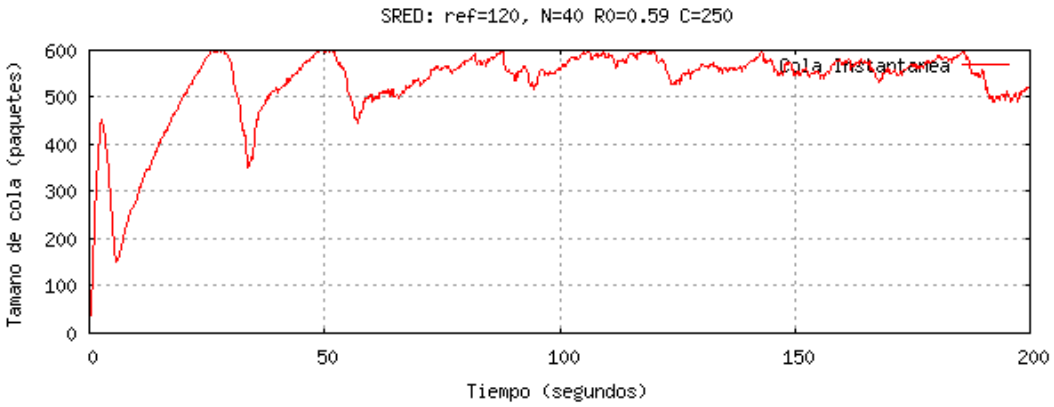


Figura 7.5 Cola instantánea SRED

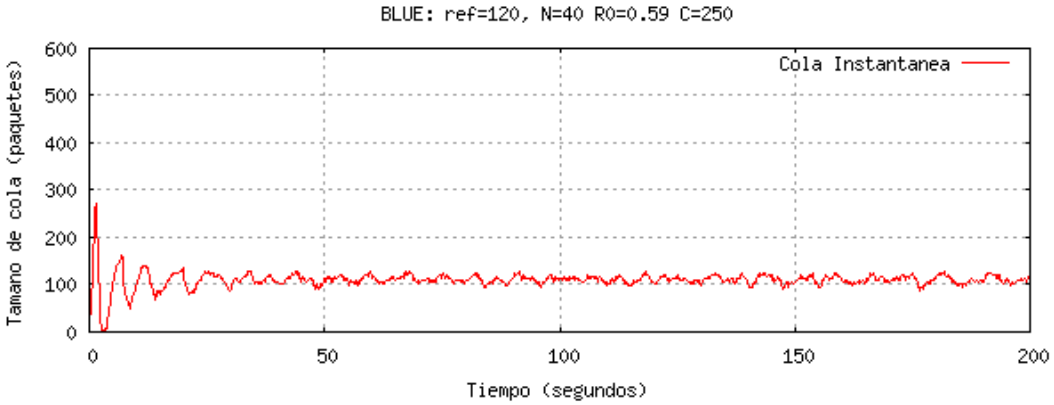


Figura 7.6. Cola instantánea BLUE

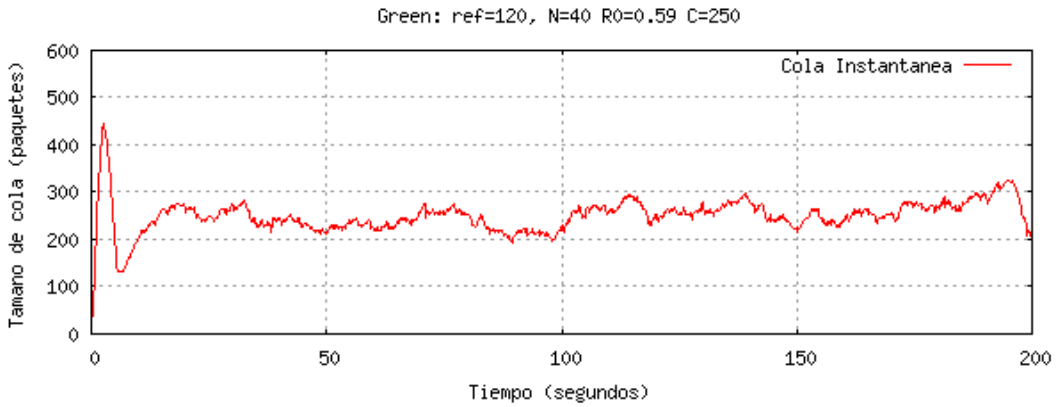


Figura 7.7. Cola instantánea GREEN

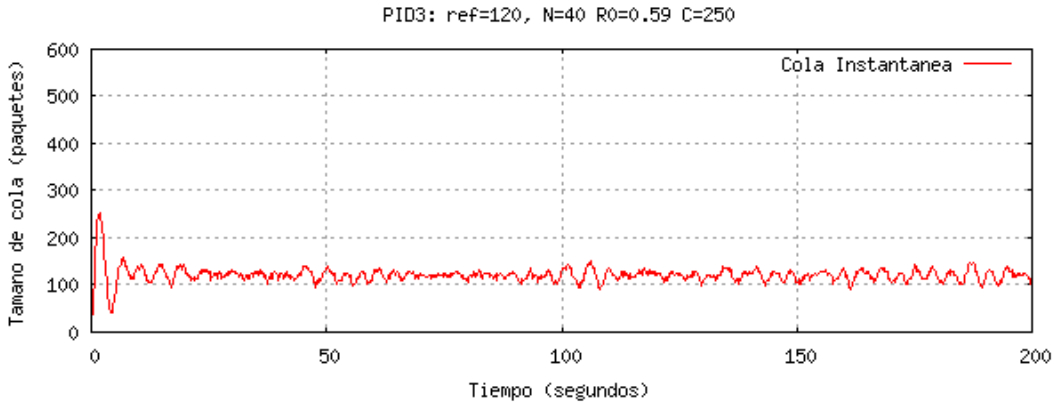


Figura 7.8. Cola instantánea PID3

- **RED:** Comienza con una oscilación de corta duración hasta que se logra estabilizar alrededor del valor 100. Esto es debido a los umbrales que tiene definidos, estando dichos umbrales entre el mínimo, en 70, y el máximo, en 120.
- **ARED:** La oscilación con la que comienza al principio es exactamente igual a la del algoritmo anterior. Como puede observarse en la **figura 7.3** se mantiene entre los valores definidos por los umbrales, pero tiene una oscilación algo superior al algoritmo RED.
- **FRED:** En la **figura 7.4** podemos observar los resultados de este algoritmo. Como se puede apreciar, al comienzo de la simulación se produce una oscilación, pero mucho más corta y controlada que en los algoritmos anteriores. El tamaño de la cola se estabiliza en torno al valor 100,

continuando con una mayor regularidad, en comparación con los algoritmos RED y ARED.

- **SRED:** Este algoritmo muestra al principio un funcionamiento muy similar al algoritmo *DropTail,* ya que pueden apreciarse los dientes de sierra en su gráfica. Con lo cual los descartes se van produciendo en el momento que la cola se llena. Más o menos, hacia la mitad de la simulación, la cola intenta estabilizarse pero de una forma no regular.
- **BLUE:** Como se puede observar ,este algoritmo tiene un comportamiento muy similar al de RED, aunque éste consta de un transitorio más largo hasta que logra estabilizarse. En la **figura 7.6** se puede apreciar como BLUE presenta unas oscilaciones más acentuadas en algunos tramos comparadas con RED, aunque ambos simulan seguir una referencia constante.
- **GREEN:** Este algoritmo presenta una tamaño de cola entre 200 y 300 paquetes, aunque el tamaño de la cola no es muy grande, se presentan unas notables oscilaciones a lo largo de la simulación.
- **PID:** Se puede observar en la **figura 7.8** cómo se presenta una rápida respuesta del sistema y cómo consigue estabilizarse en torno al valor de referencia 120, sin presentar grandes oscilaciones.

La siguiente figura muestra las gráficas solapadas del experimento:

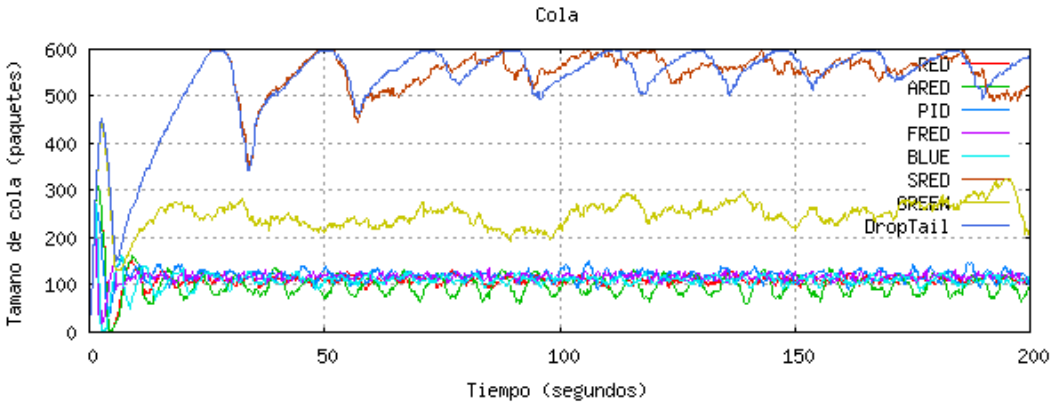


Figura 7.9. Colas instantáneas

Como conclusión, se podría destacar que los algoritmos RED, ARED, FRED, BLUE y PID3 tienen un comportamiento similar. Todos ellos simulan seguir más o menos una referencia fija , aunque el algoritmo PID3 es el único de ellos que realmente sigue el valor

de referencia con el que se simula. Los algoritmos RED, ARED, FRED y BLUE se mantienen entre los umbrales definidos.

En cambio, SRED y GREEN no siguen este comportamiento.

SRED presenta al principio un comportamiento similar a *Drop Tail*, aunque después intenta estabilizar la ocupación de la cola en niveles altos, teniendo en cuenta la carga de tráfico en la red.

Por otro lado, GREEN mantiene unos tamaños de cola pequeños durante toda la simulación menos al principio, evitando así que flujos TCP de larga duración induzcan a congestión.

Utilización del enlace

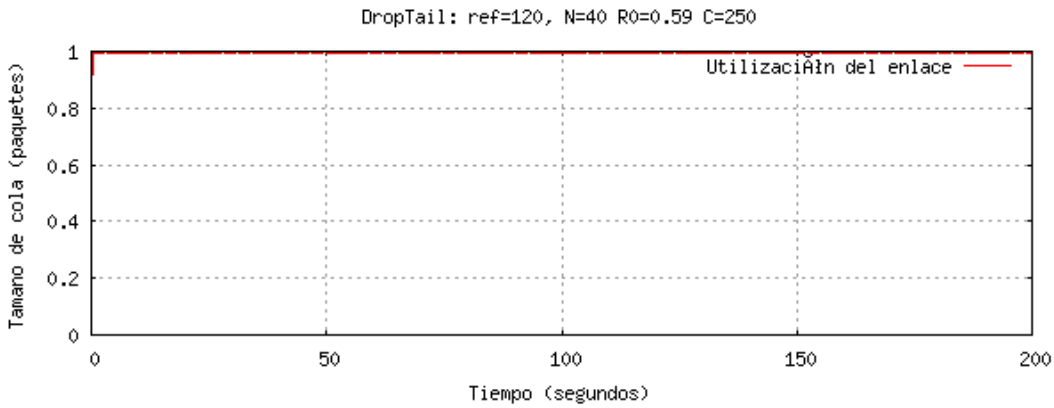


Figura 7.10. Utilización del enlace *DropTail*

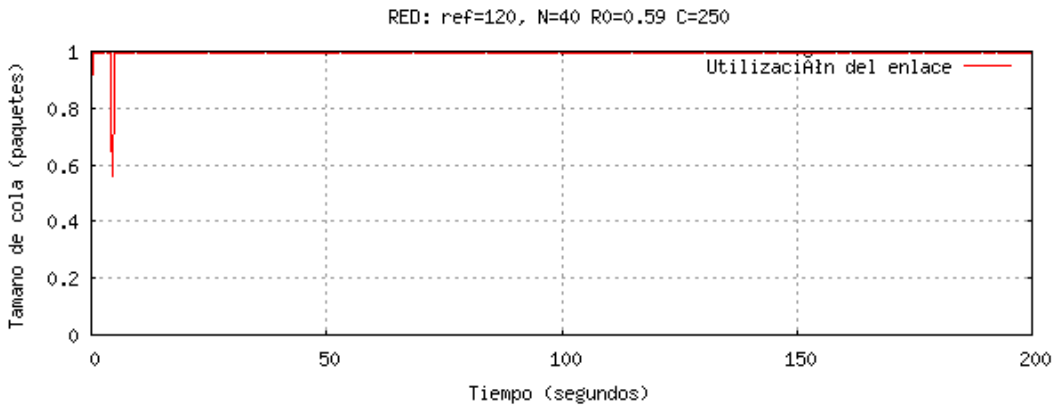


Figura 7.11. Utilización del enlace RED

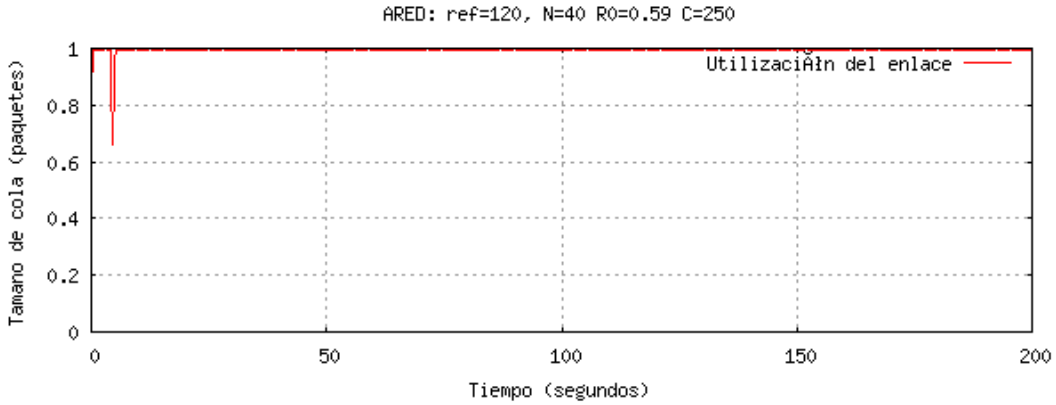


Figura 7.12. Utilización del enlace ARED

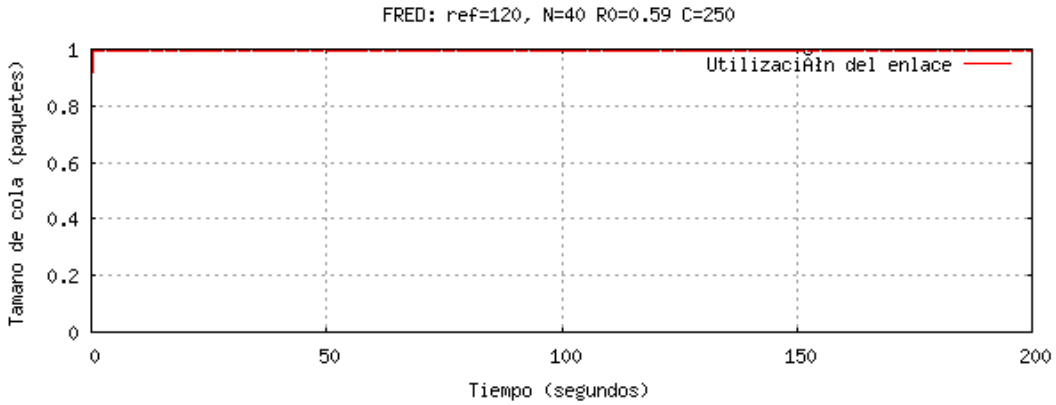


Figura 7.13. Utilización del enlace FRED

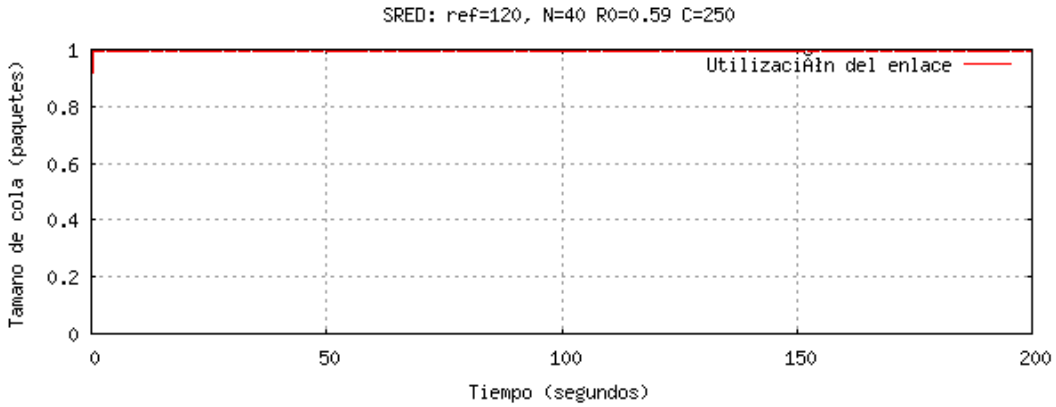


Figura 7.14. Utilización del enlace SRED

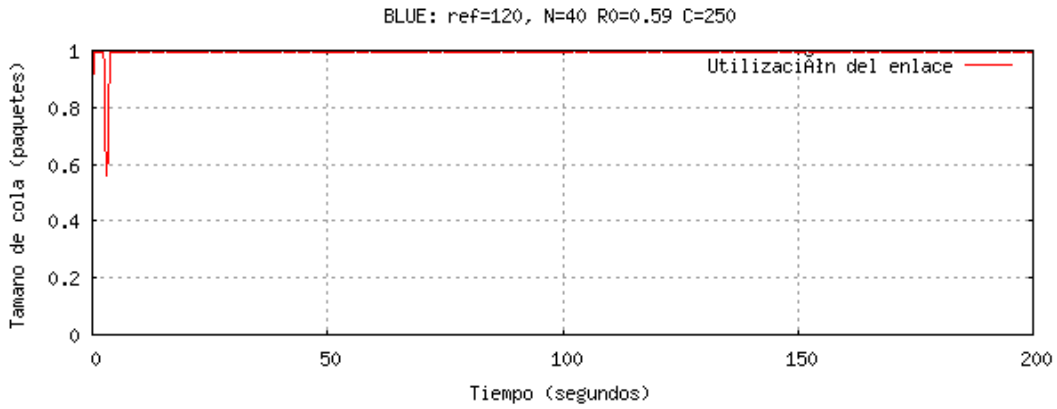


Figura 7.15. Utilización del enlace BLUE

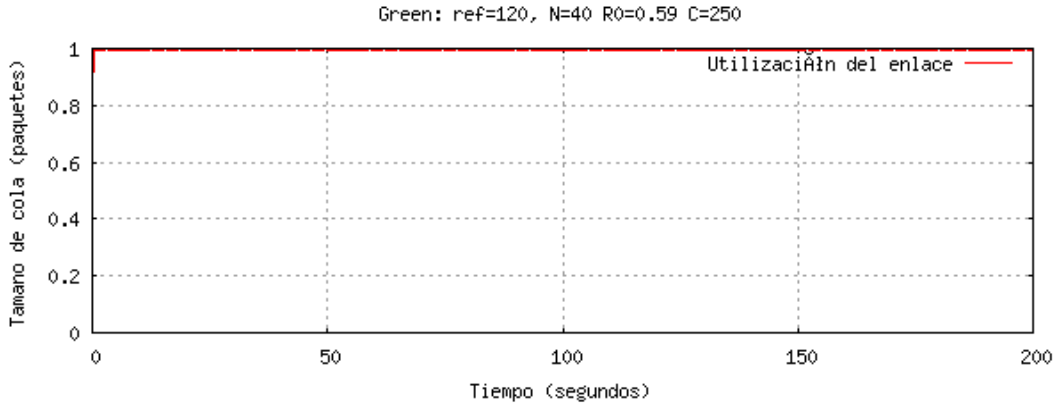


Figura 7.16. Utilización del enlace GREEN

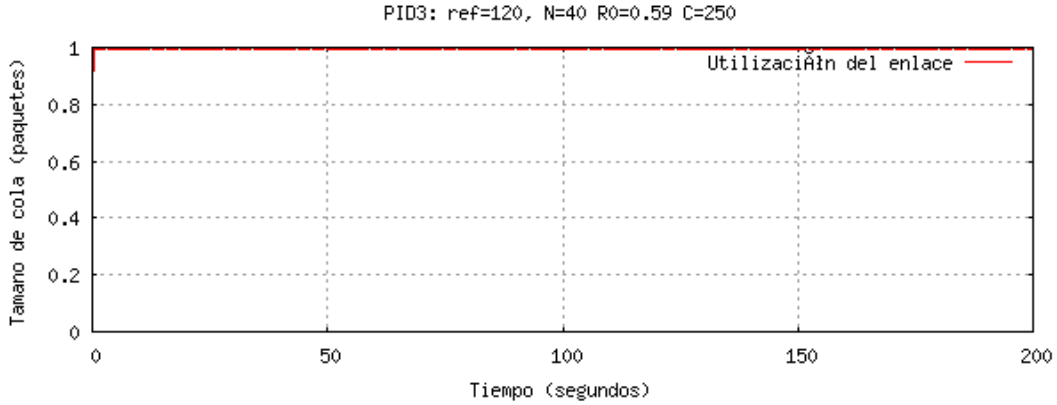


Figura 7.17. Utilización del enlace PID

Como puede apreciarse en las figuras anteriores, la utilización del canal por parte de los algoritmos es completa, sin desaprovechar en ningún momento de la simulación el ancho de banda disponible.

Probabilidad de descarte

A continuación mostraremos la probabilidad de descarte de los algoritmos RED, ARED, FRED, BLUE y PID3.

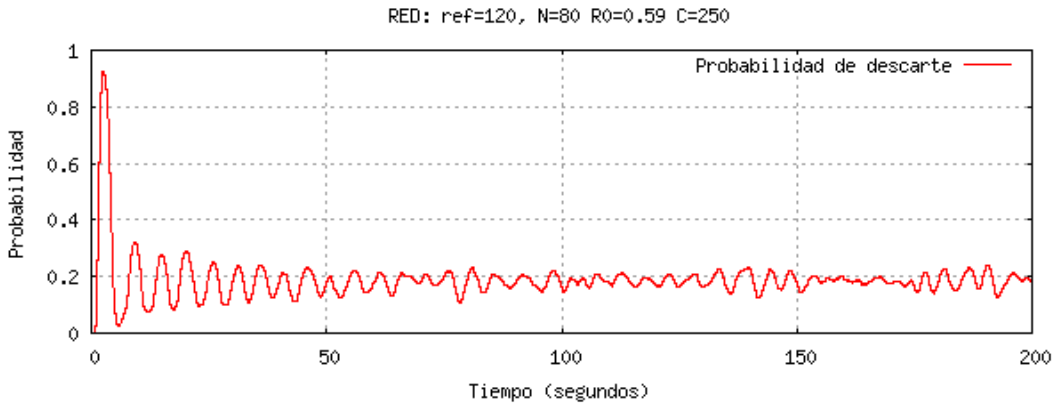


Figura 7.18 Probabilidad de descarte RED

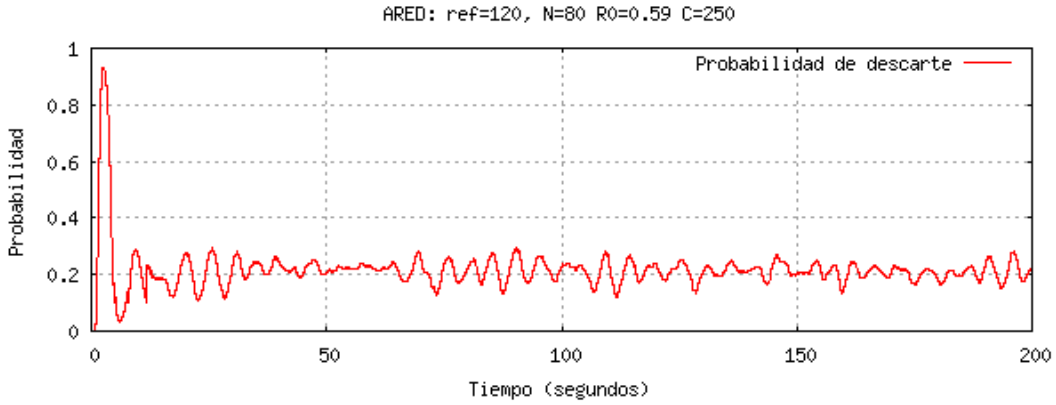


Figura 7.19. Probabilidad de descarte ARED

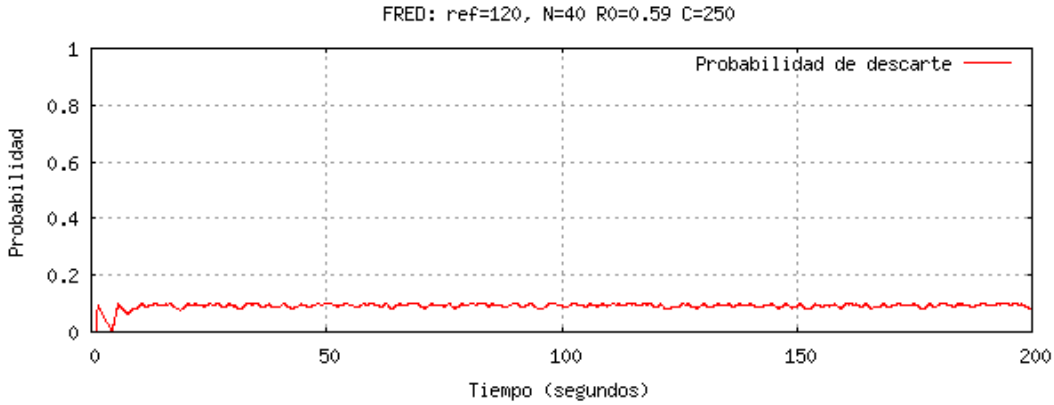


Figura 7.20. Probabilidad de descarte FRED

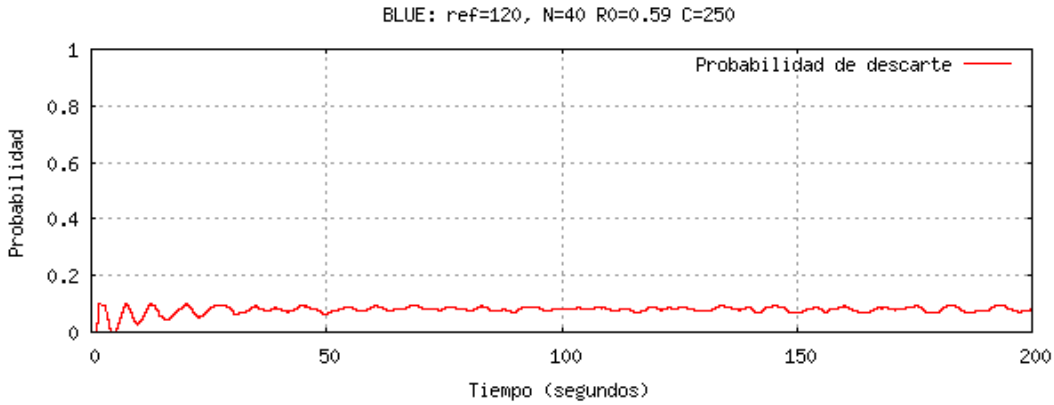


Figura 7.21. Probabilidad de descarte BLUE

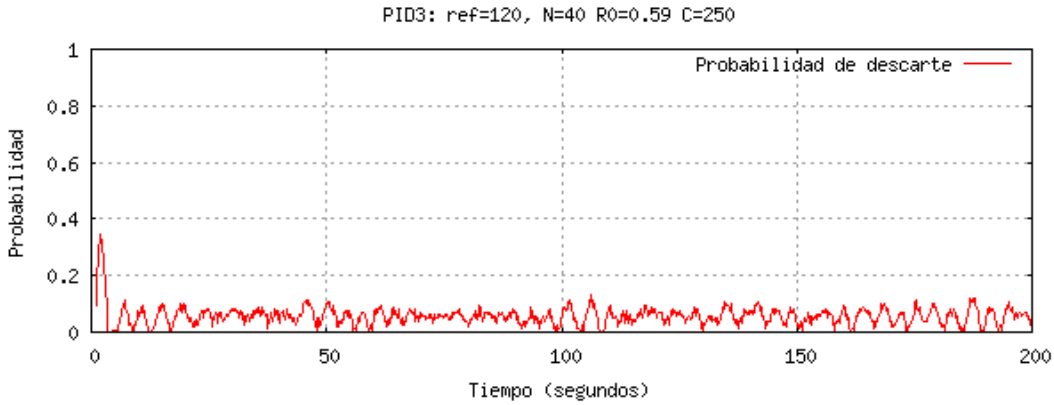


Figura 7.22. Probabilidad de descarte PID3

- **RED:** Como se puede observar en la **figura 7.18** el algoritmo RED tiene un pico elevado de probabilidad de descarte de paquetes al comienzo de la simulación, hasta que llega al punto en el que se estabiliza sobre el valor 0.2 .
- **ARED:** Tiene un comportamiento muy similar al algoritmo RED como puede apreciarse, comienza con un pico elevado de probabilidad de descarte hasta que se estabiliza. Cuando se analice más en detalle, se observará cómo ARED tiene un valor ligeramente superior.
- **FRED:** Este algoritmo no sufre variación, manteniéndose una probabilidad de descarte muy regular, en torno al valor 0.1.
- **BLUE:** Comportamiento muy similar al anterior como se puede observar en la **figura 7.21**, con una probabilidad de 0.1, pero con un número de oscilaciones ligeramente superiores al algoritmo FRED.
- **PID3:** Consigue mantener una probabilidad de descarte con poca variación. Se puede observar, por la **figura 7.22**, cómo oscila entre los valores 0 y 0,1. Aunque como puede observarse, consta de un mayor número de oscilaciones frente a las que presentan FRED y BLUE.

La siguiente figura muestra las probabilidades de los algoritmos solapadas:

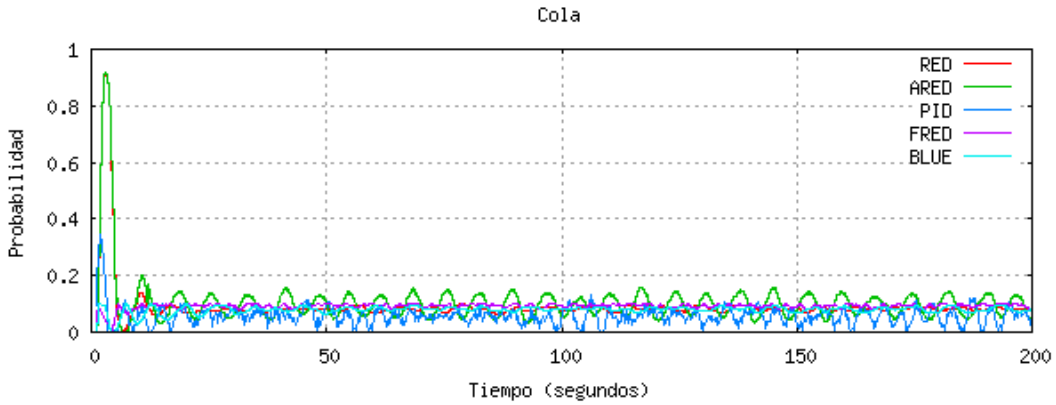


Figura 7.23. Probabilidad de descarte

Como conclusión se podría destacar que los algoritmos RED, ARED, FRED, BLUE y PID3 tienen un comportamiento similar. Todos ellos tienden a mantenerse entre unos valores de probabilidad de descarte una vez que se estabilizan. Como puede apreciarse, las oscilaciones y la probabilidad de descarte de RED y ARED son mayores que las de los algoritmos BLUE y FRED, siendo este último el que mantiene una probabilidad más regular. En cuanto a PID, se puede observar que su probabilidad varía entre 0 y 0.1,

consiguiendo en ciertos momentos una probabilidad de descarte menor y, como se observa con un análisis más detallado, esto permite que su probabilidad media de descarte sea menor.

Tasa de pérdida

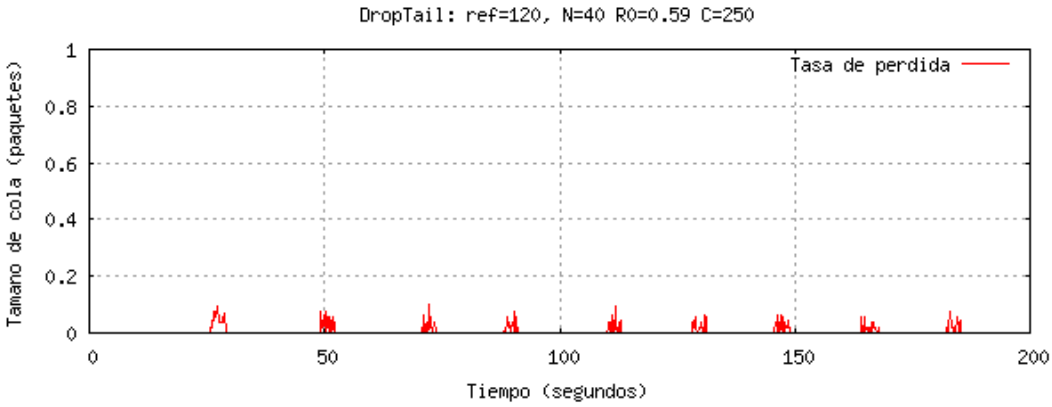


Figura 7.24. Tasa de pérdida *DropTail*

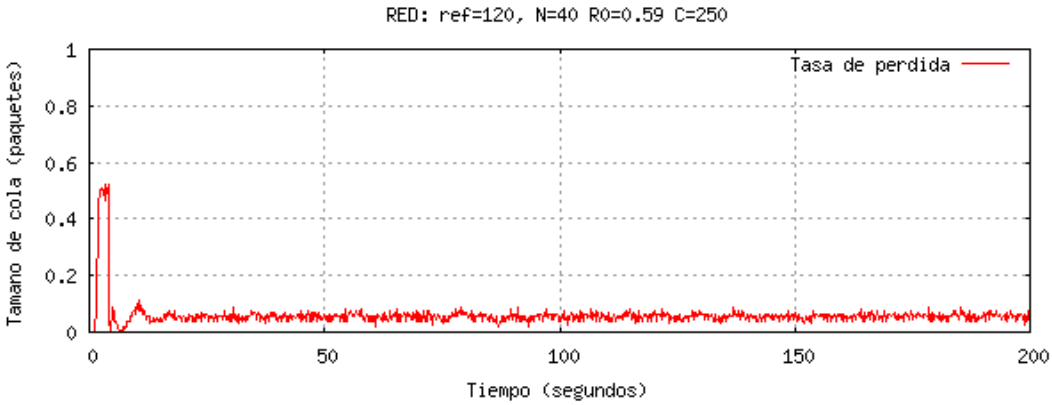


Figura 7.25. Tasa de pérdida RED

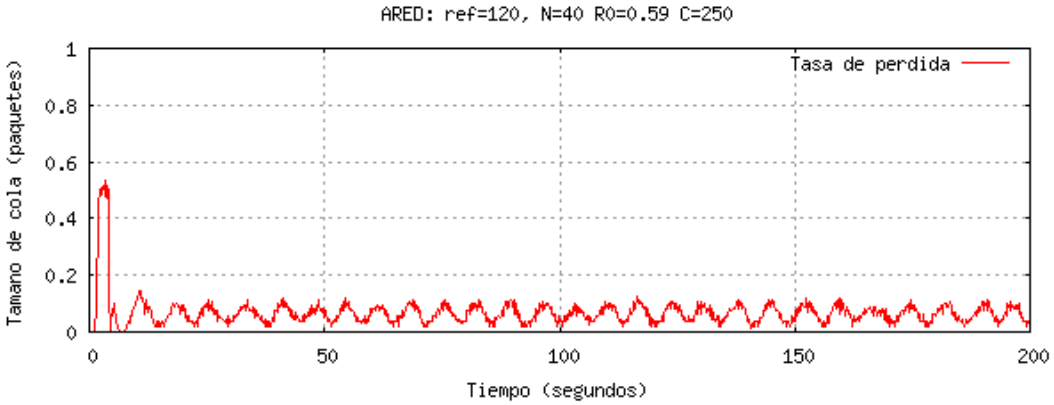


Figura 7.26. Tasa de pérdida ARED

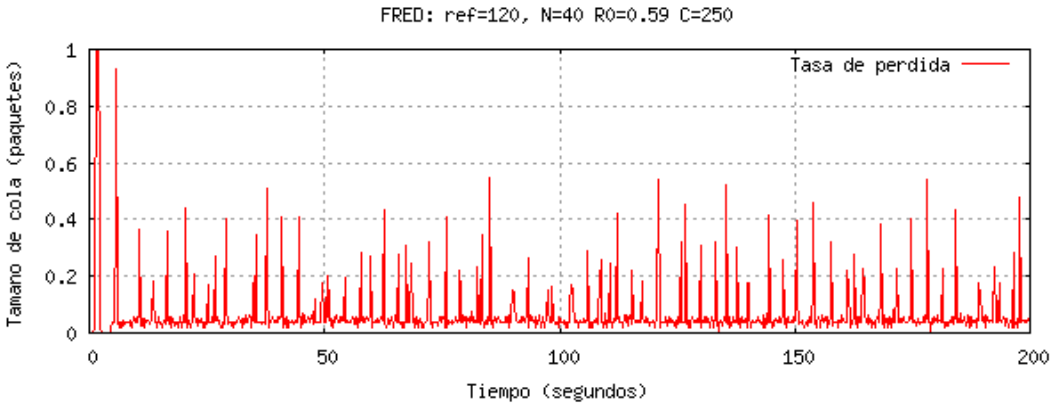


Figura 7.27. Tasa de pérdida FRED

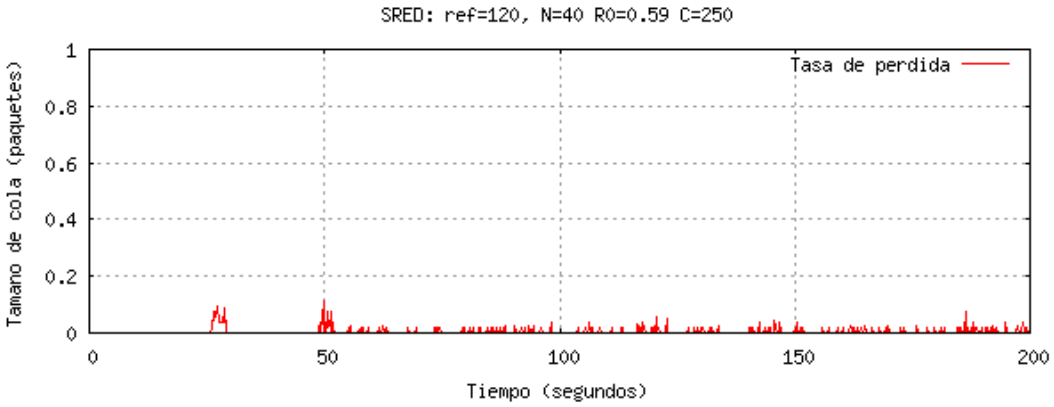


Figura 7.28. Tasa de pérdida SRED

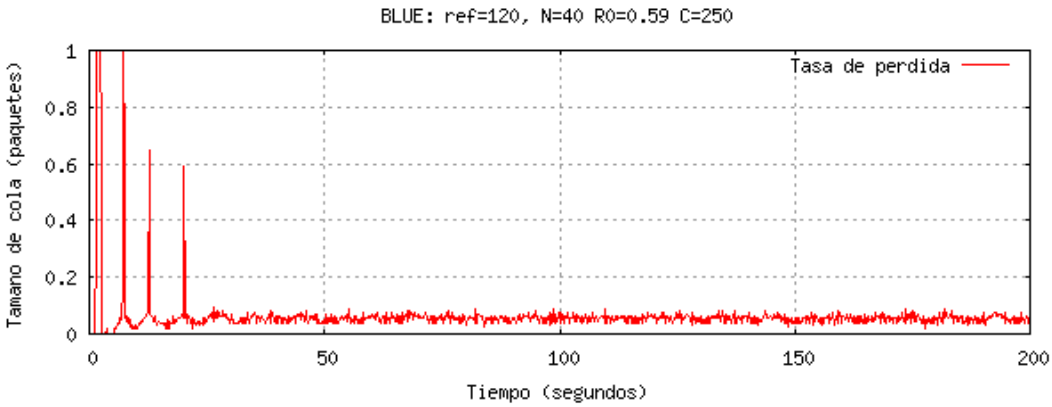


Figura 7.29. Tasa de pérdida BLUE

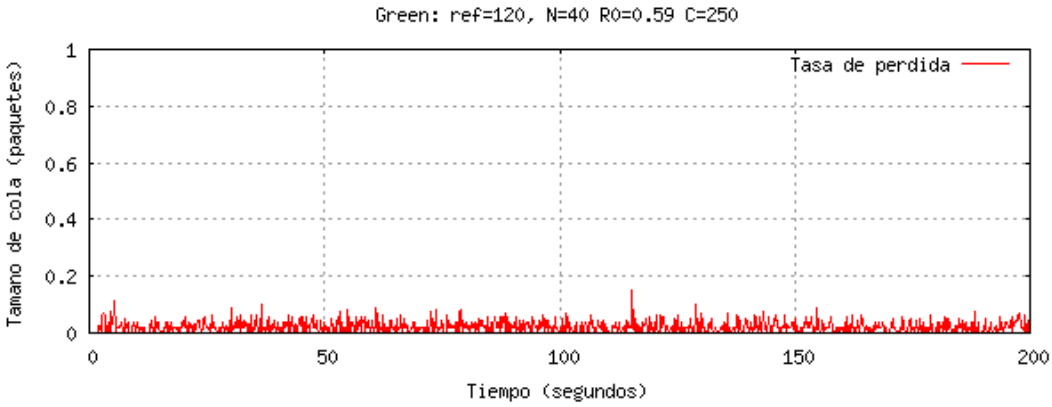


Figura 7.30. Tasa de pérdida GREEN

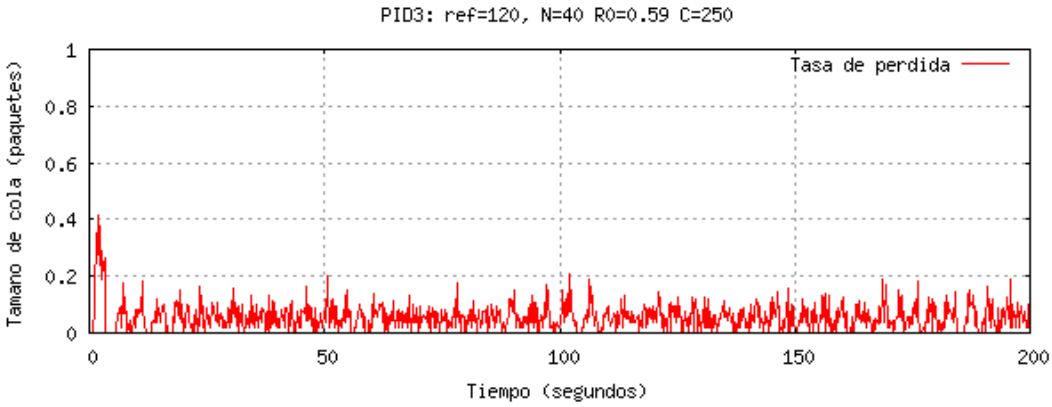


Figura 7.31. Tasa de pérdida PID

Como puede apreciarse en la **figura 7.24** para el algoritmo *DropTail* (y en menor medida para el algoritmo SRED, ya que al principio se comporta como *DropTail*) mientras

la cola no se llene, no se descartará ningún paquete y los agentes TCP no tendrán notificación de congestión alguna, por lo que aumentarán progresivamente el tamaño de su ventana en cada RTT. El problema surge porque *Drop Tail* solamente es capaz de notificar la congestión cuando la cola se ha llenado. Cuando la cola ha llegado a su tope, se produce una oleada de descartes que causa que todos los emisores TCP aprecien una congestión grave de la red y reduzcan drásticamente el tamaño de su ventana, por lo que después de cada periodo de congestión se entra en un periodo de escaso tráfico, provocando grandes oscilaciones del tamaño de la cola, como ya vimos con anterioridad en la **figura 7.1**.

En cuanto a los algoritmos RED y ARED, presentan un comportamiento muy similar, pero ARED tiene un mayor número de oscilaciones.

FRED y BLUE tienen una tasa de pérdidas similar en cuanto a los picos que se introducen, aunque cuando BLUE llega a estabilizarse deja de presentar esos picos, al contrario que FRED.

PID y GREEN presentan una tasa de pérdidas regulares, acercándose en muchos puntos a valores nulos.

Análisis detallado

Algoritmo	TamMediaCola	DesvCola	ProbDescarte	UtilizacionEnlace
DropTail	556.952	39.2464	-	0.99999
RED	111.03	7.0194	0.08119	1.0000
ARED	97.9609	17.3267	0.092439	0.99999
FRED	117.083	5.78037	0.0929788	0.99999
SRED	552.56	37.7012	-	0.99999
BLUE	110.657	8.08696	0.0807785	0.99999
GREEN	251.211	24.2707	-	0.99999
PID	120.115	9.65368	0.0556443	0.99999

Tabla 7.1. Valores significativos Experimento I

Como se había comentando anteriormente, el valor de la probabilidad de descarte de un paquete es mejor en el algoritmo PID, aunque tuviera un mayor número de oscilaciones, muchos valores se acercaban al nulo, logrando una probabilidad de descarte menor que en el resto de algoritmos.

El tamaño medio de la cola cuando se emplea dicho algoritmo, coincide con el valor de referencia que se le pasa en la simulación. También se puede observar cómo la desviación de la cola es bastante baja en este algoritmo, aunque en FRED es menor, la tasa de probabilidad de descarte de FRED es mucho más elevada que en el algoritmo PID. Con todo esto, se puede considerar que el algoritmo de control PID es bastante aconsejable.

Aunque se han estado observando anteriormente unos datos similares en los algoritmos RED y ARED, aquí se puede constatar cómo la desviación del algoritmo ARED es superior a la de RED, indicando que hay mayores oscilaciones en el tamaño de la cola. La probabilidad de descarte alcanzada en esta simulación es bastante similar para ambos, siendo ligeramente superior en ARED.

Comparación de resultados

Una vez analizados los resultados, se pretende comparar las gráficas obtenidas en la simulación con las mostradas en el artículo y así confirmar los resultados conseguidos.

- La **figura 7.32** muestra la cola instantánea y la tasa de descarte del algoritmo *DropTail*.

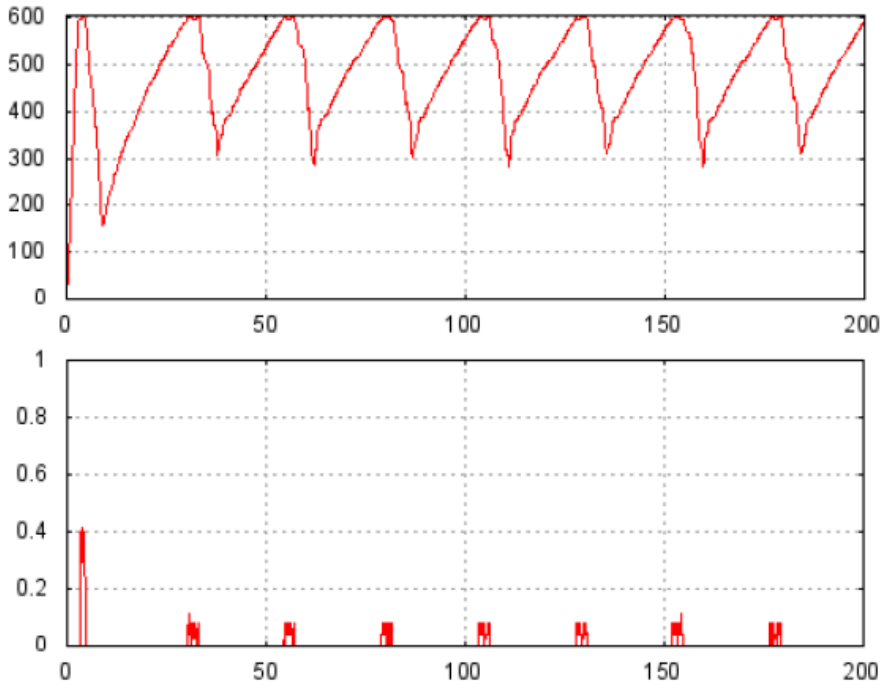


Figura 7.32. *DropTail:* cola instantánea y tasa de descarte.

Como se puede apreciar, la gráfica de la parte superior tiene la forma de dientes de sierra como la **figura 7.1** mostrada al inicio de este capítulo. La figura de la parte inferior, que muestra la tasa de descarte, tiene unos datos similares a los mostrados en la **figura 7.24**.

- La **figura 7.33** muestra la cola instantánea y la probabilidad de descarte del algoritmo PID.

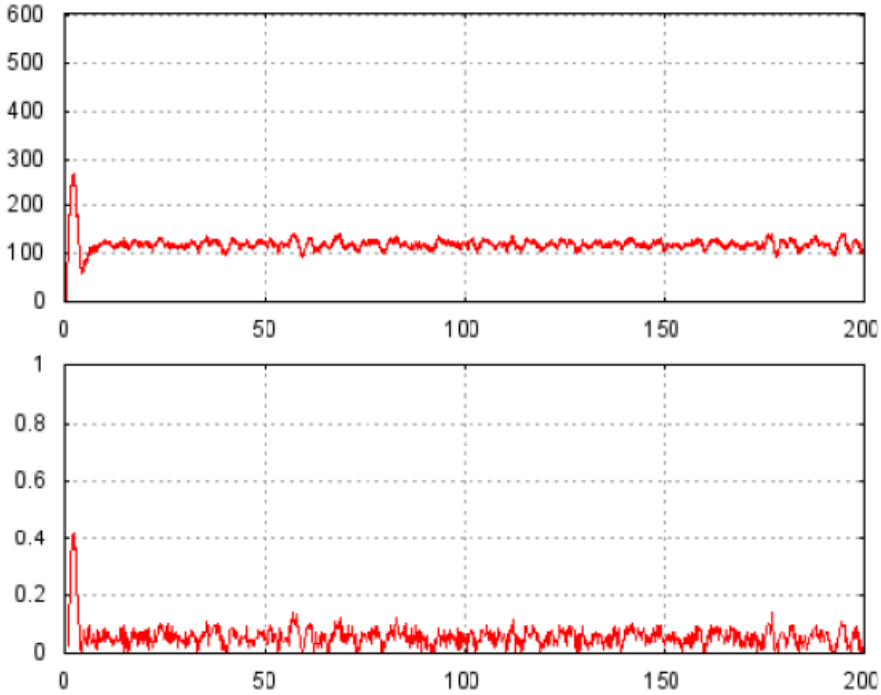


Figura 7.33. PID: cola instantánea y probabilidad de descarte.

Como se puede observar en la gráfica de la parte superior, la cola se mantiene en torno al valor de referencia 120, lo cual también se puede apreciar en la **figura 7.8** de esta simulación. La **figura 7.22** muestra unos valores de probabilidad de descarte ligeramente superiores a los que se observan en la figura anterior, aunque con valores cercanos.

- La **figura 7.34**. muestra la cola instantánea y la probabilidad de descarte del algoritmo RED

Figura 7.34. RED: Cola instantánea y probabilidad de descarte

La cola instantánea obtenida en la simulación del algoritmo RED (**Figura 7.2**) y la de la gráfica de la parte superior de la **figura 7.34** presenta una forma muy similar, con una probabilidad de descarte ligeramente superior, como se puede apreciar en la **figura 7.18**.

Como conclusión, se puede afirmar que los resultados obtenidos son satisfactorios, ya que son bastante aproximados a los datos presentados en el artículo, lo cual indica que la simulación realizada es correcta.

7.2 Experimento II

Dado que este experimento es una continuación del anterior, puesto que el número de flujos ahora es el doble, para evitar una extensión demasiado grande de la memoria, se muestran los datos de la tabla sin incluir las gráficas de los resultados que podrán ser consultadas en el CD adjunto a esta memoria (ver Anexo C).

Algoritmo	TamMediaCola	DesvCola	ProbDescarte	UtilizacionEnlace
DropTail	570.453	30.2851	-	0.99999
RED	131.013	9.62664	0.181262	1.0000
ARED	97.1812	10.0437	0.211672	0.99999
FRED	119.355	7.96677	0.0968421	0.99999
SRED	563.847	30.2104	-	0.99999
BLUE	118.96	8.58606	0.0970922	0.99999
GREEN	548.112	20.2718	-	0.99999
PID	120.072	4.99419	0.141961	0.99999

Tabla 7.2. Valores significativos Experimento II

Como se podía esperar, el algoritmo PID sigue manteniendo el tamaño de la cola según el valor de referencia que se emplea en la simulación. En este caso, sí que tiene el menor valor de desviación de la cola, lo que indica un buen comportamiento cuando el numero de flujos es incrementado.

Cabe destacar en este momento, el buen comportamiento que siguen manteniendo los algoritmos FRED y BLUE, ya que disponen de un tamaño de cola similar, su desviación es baja (lo que indica un tamaño de cola bastante estable) y una probabilidad de descarte similar a la que tenían cuando el número de flujos era la mitad.

Como se puede observar en ambos experimentos, la utilización del canal por parte de los algoritmos es total.

7.3 Experimento III

Como se indicó previamente en el apartado 6.3.2, el número de flujos con el que se iba a realizar el experimento iban a ser 40, 60, y 80 empleando los algoritmos RED y ARED. Para no saturar la documentación con múltiples gráficas, se mostrarán sólo las gráficas para N=80 y el muestreo de 0.001 ms. En la documentación del proyecto están incluidas las gráficas restantes, realizadas con el tiempo de muestreo 0,001 ms (tiempo de muestreo indicado en el artículo "PURPLE: *Predictive Active Queue Management Utilizing Congestion Information*" [PLE1]) y 0,2013 ms.

Se presentan a continuación las cuatro gráficas correspondientes al tamaño de la cola, utilización del enlace, probabilidad de descarte y tasa de pérdidas para cada controlador empleado. Se comentan los resultados obtenidos.

7.3.1 RED

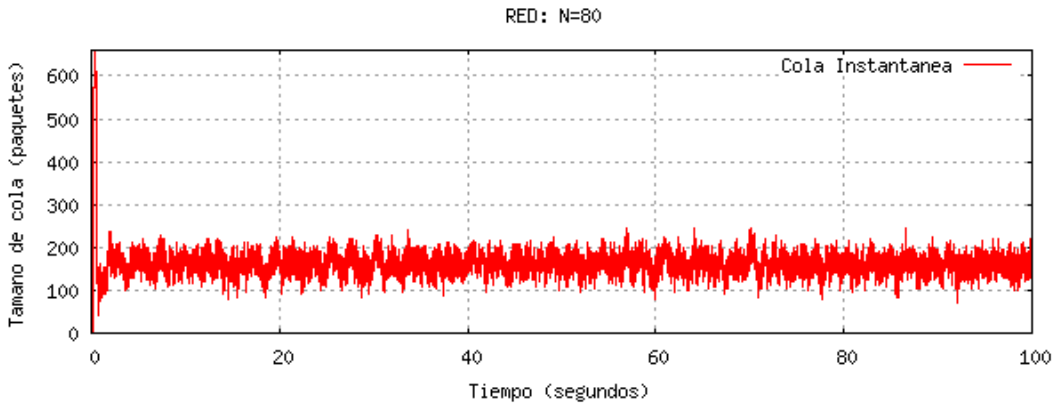


Figura 7.35. Cola instantánea RED

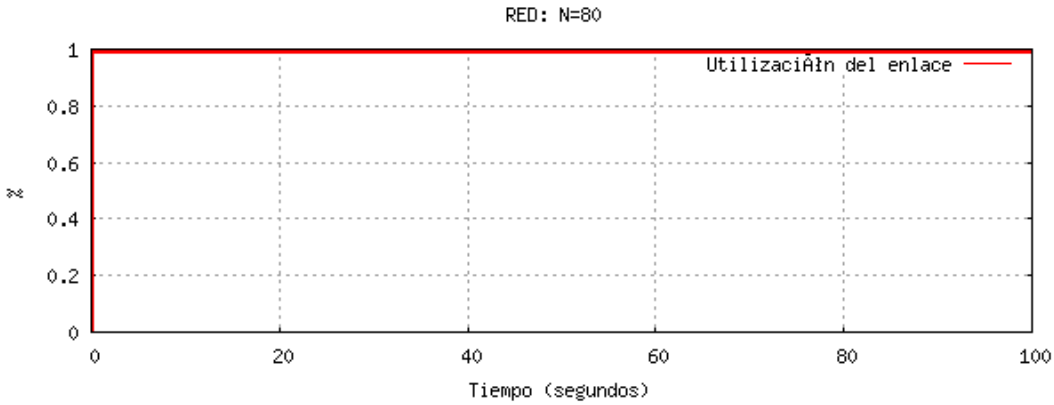


Figura 7.36. Utilización del enlace RED

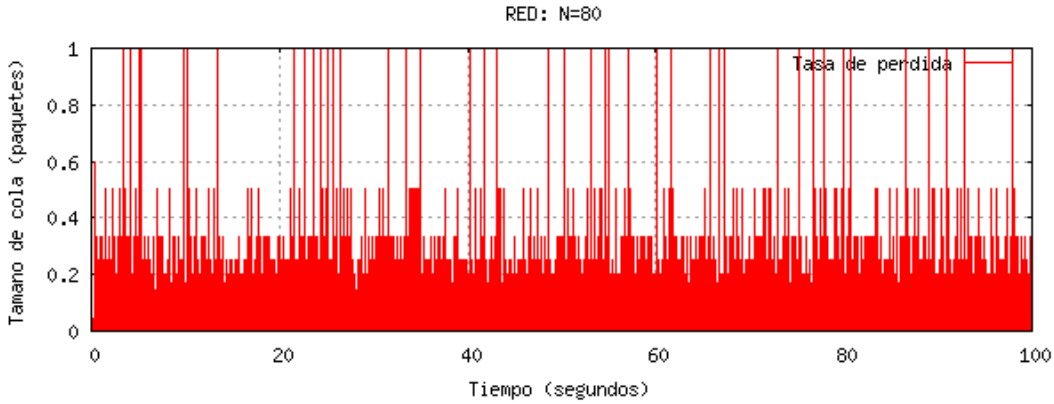


Figura 7.37. Tasa de pérdida RED

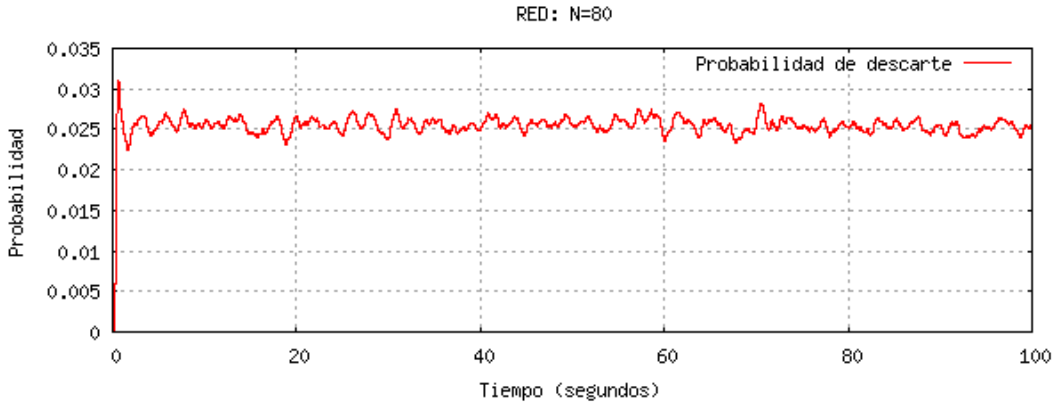


Figura 7.38. Probabilidad de descarte RED

7.3.2 ARED

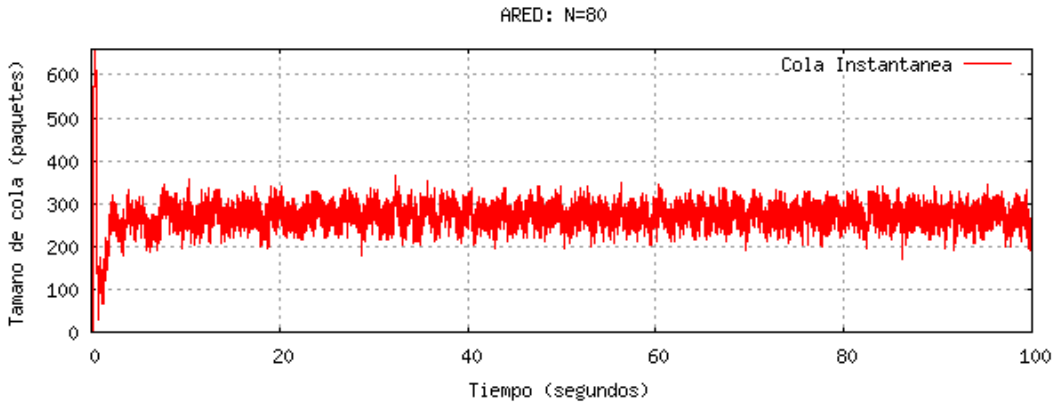


Figura 7.39. Cola instantánea ARED

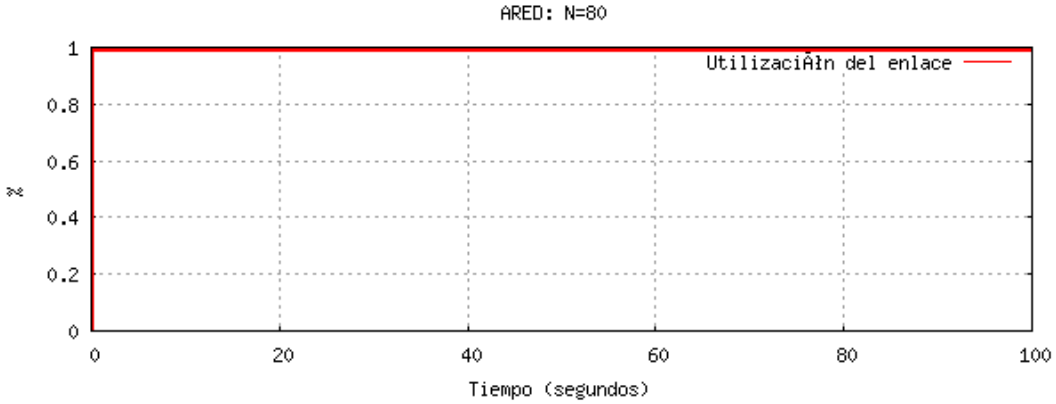


Figura 7.40. Utilización del enlace ARED

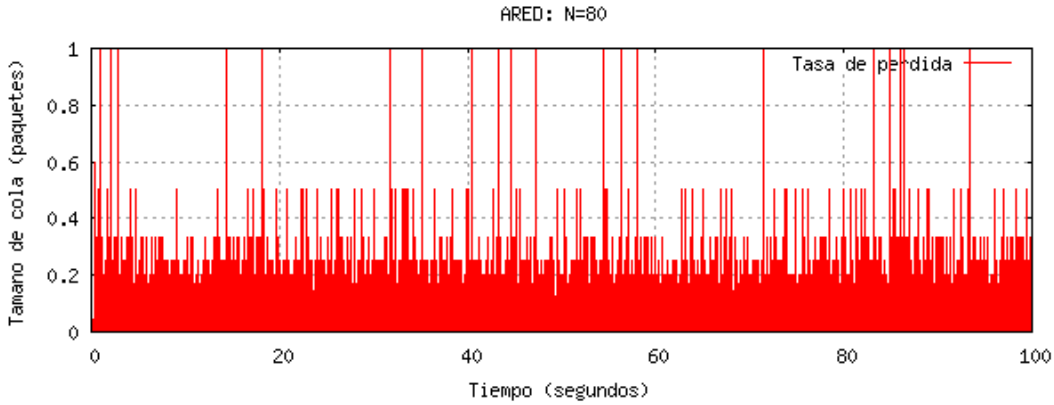
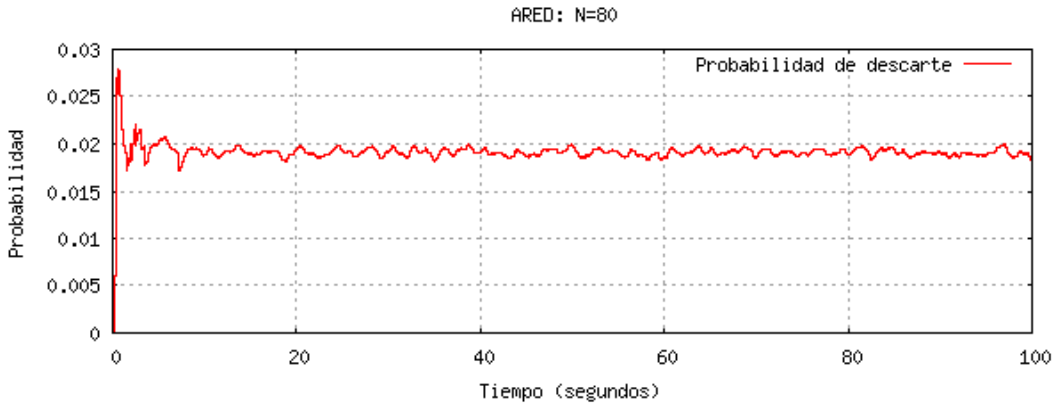


Figura 7.41. Tasa de pérdida AREDFigura 7.42. Probabilidad de descarte



A simple vista, comparando las gráficas, puede verse como AREDtiene una probabilidad de descarte y una tasa de pérdidas menor que las obtenidas con el algoritmo RED.

7.3.3 Análisis detallado.

Algoritmo	RED	ARED
Tamaño Media Cola	161.062	271.982
Tasa perdidos	0.0171741	0.0127522
Desviación Perdidos	0.0526747	0.044507
Probabilidad Descarte	0.0255245	0.0190923
Desviación de probabilidad	0.000768047	0.000342297
Utilización del enlace	0.987313	0.987313

Tabla 7.3. Valores significativos Experimento III

La tabla anterior recoge los valores más significativos de las medias calculadas después de realizar las simulaciones.

En la **tabla 7.3** se puede apreciar como:

- Ambos hacen uso del canal de igual forma, moviéndose en valores similares a los del artículo.
- La tasa de paquetes perdidos y la desviación del mismo, son inferiores para el algoritmo ARED.
- La probabilidad de descarte y la desviación de la probabilidad son también inferiores en el algoritmo ARED.

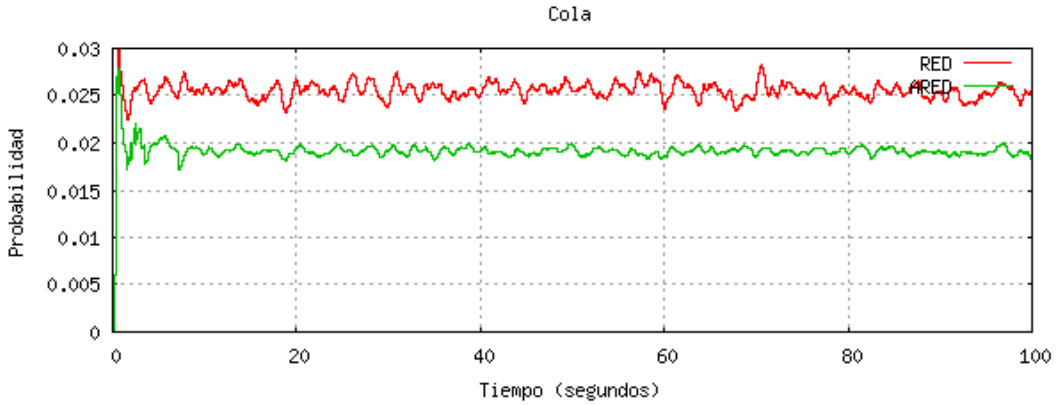


Figura 7.43. Probabilidad de descarte

- Para resaltar esto, se contrastan con las gráficas del artículo publicado, como puede observarse a continuación.

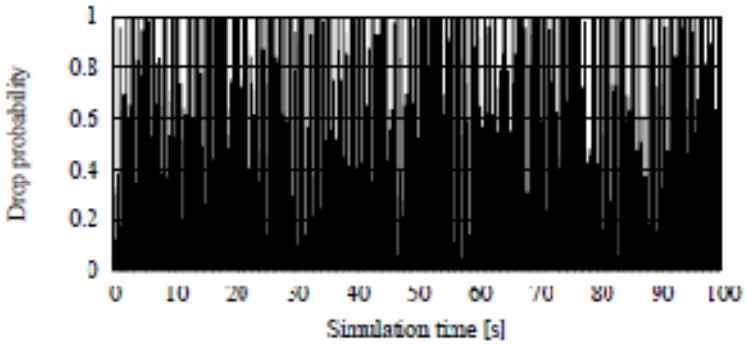


Figura 7.44. Probabilidad de descarte del articulo-RED

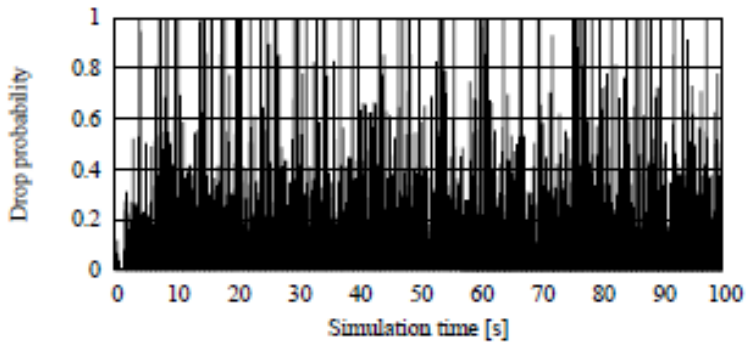


Figura 7.45. Probabilidad de descarte articulo-ARED

Como puede observarse, los datos de la **figura 7.38** y la **figura 7.42** son similares a los de las figuras anteriores, indicando que la probabilidad de descarte es inferior en el algoritmo ARED.

7.4. Experimento IV

Con este experimento lo que se intenta observar es el comportamiento de los algoritmos RED, REM y PI cuando se emplea una topología con múltiples cuellos de botella, la cual se acerca más a la realidad, como se puede apreciar en la **Figura 6.4**, no como las topologías simples antes descritas.

Para no saturar la documentación de gráficas, se mostrarán las gráficas de las colas 1 y 2 de cada algoritmo, encontrándose en la documentación de soporte digital del proyecto las gráficas restantes.

7.4.1 RED

Cola 1

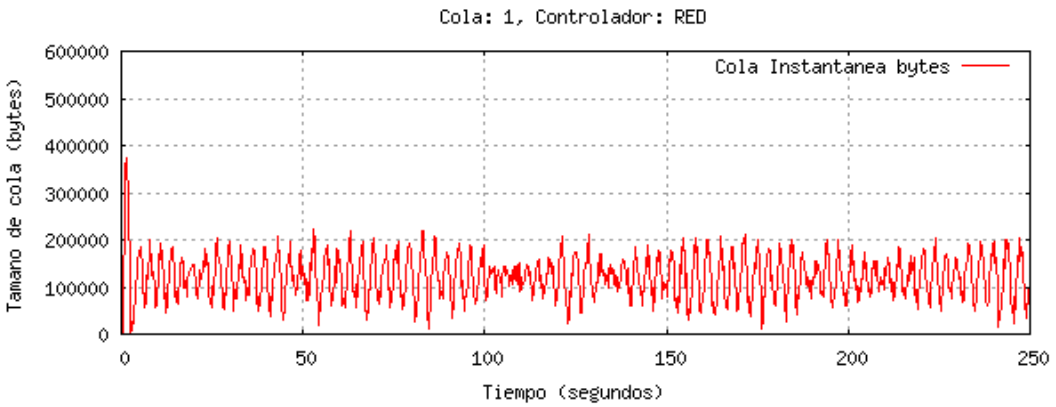


Figura 7.46. Cola instantánea cola 1-RED

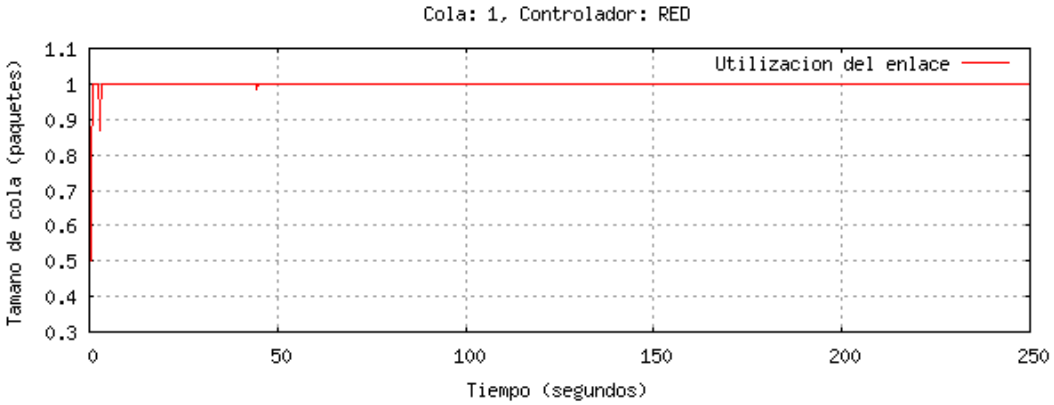


Figura 7.47. Utilización del enlace cola 1-RED

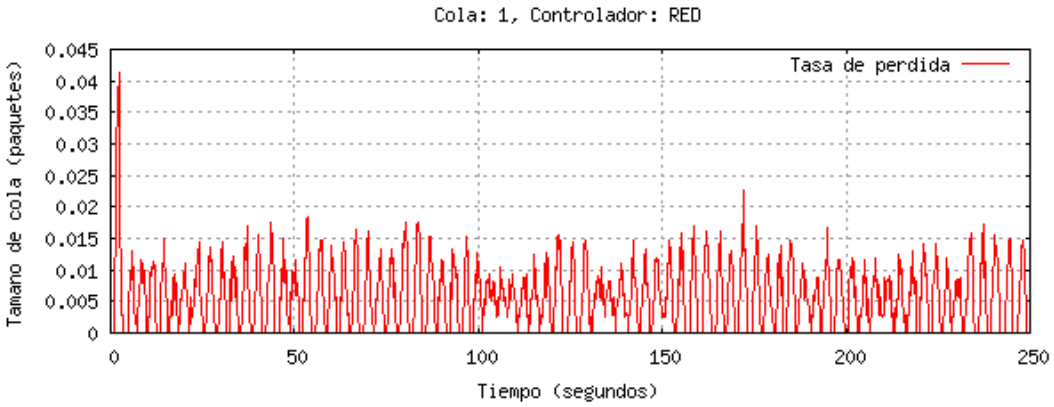


Figura 7.48. Tasa de perdida cola 1 –RED

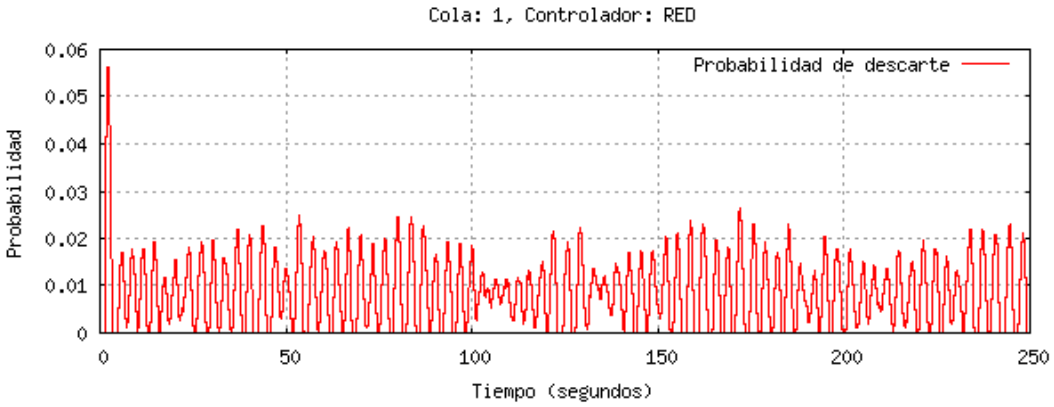


Figura 7.49. Probabilidad de descarte cola 1-RED

Cola 2

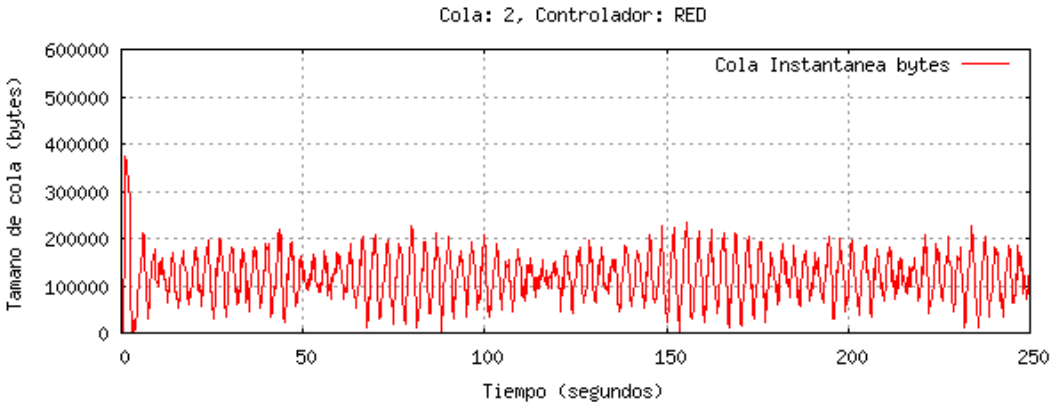


Figura 7.50. Cola instantánea cola 2-RED

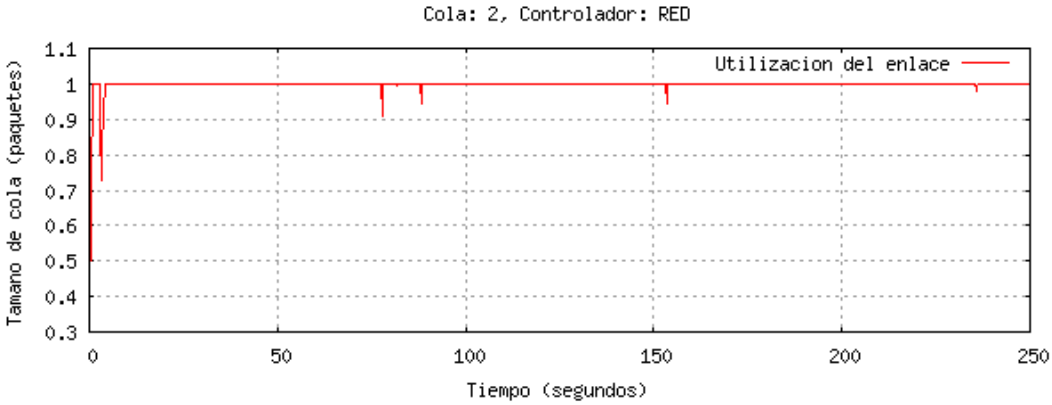


Figura 7.51. Utilización del enlace cola 2-RED

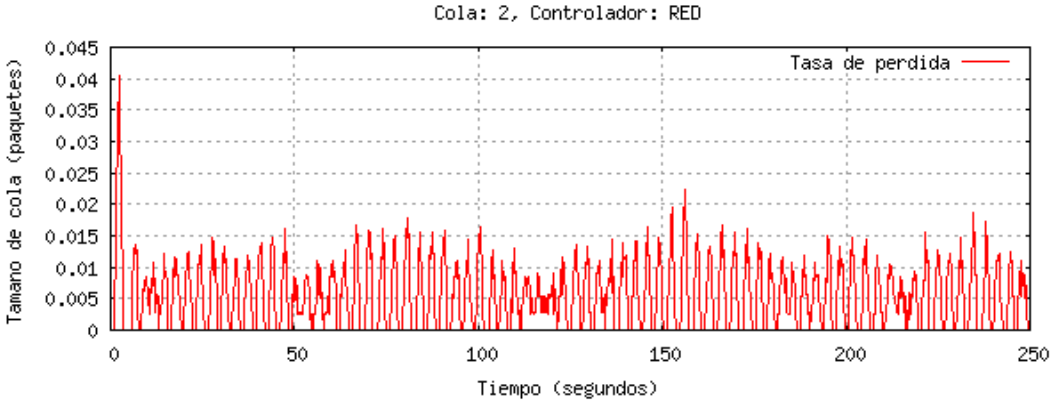


Figura 7.52. Tasa de pérdida cola 2-RED

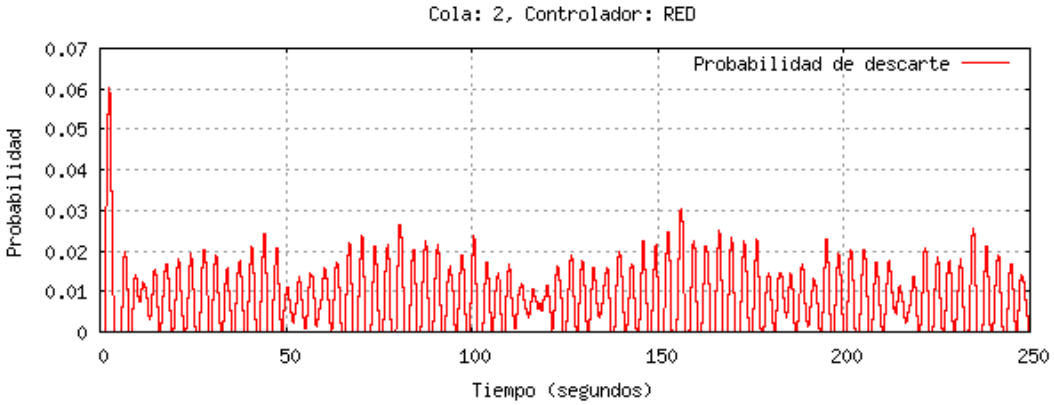


Figura 7.53. Probabilidad de descarte cola 2-RED

7.4.2 REM

Cola 1

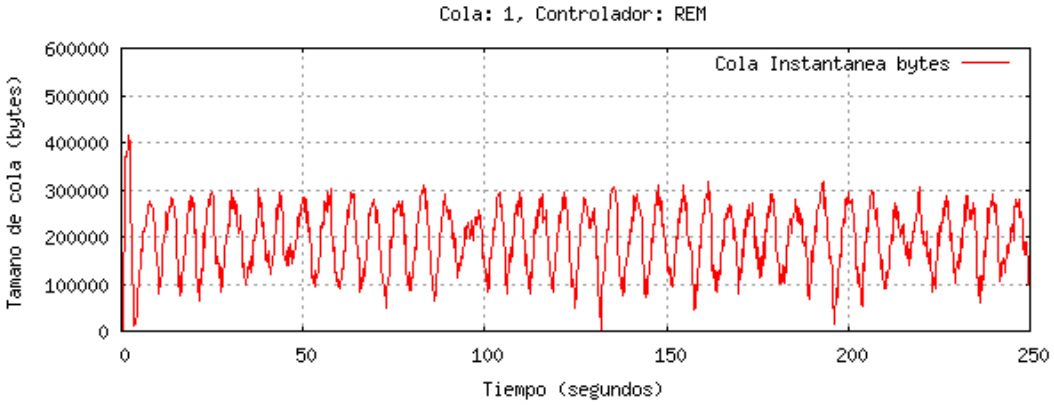


Figura 7.54. Cola instantánea cola 1-REM

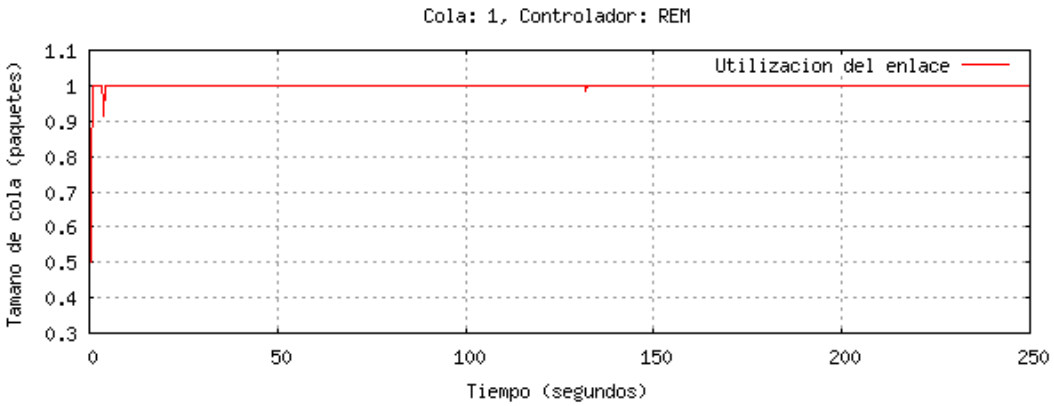


Figura 7.55. Utilización del enlace cola 1-REM

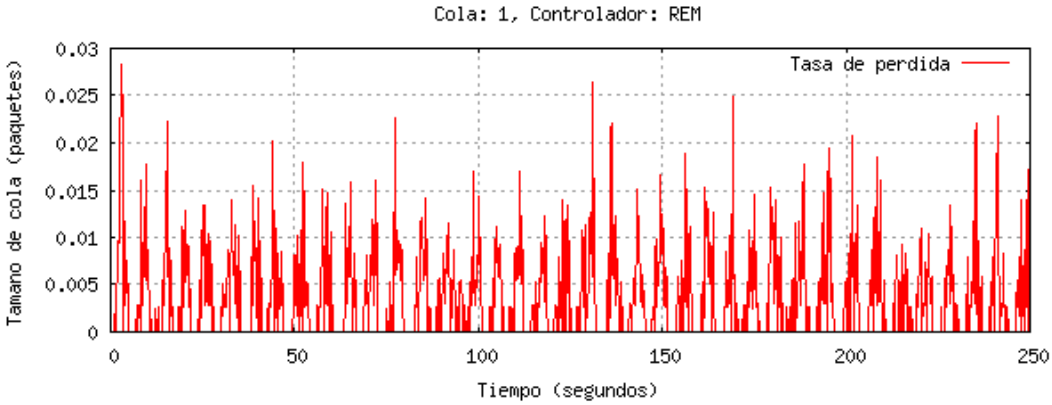


Figura 7.56. Tasa de pérdida cola 1-REM

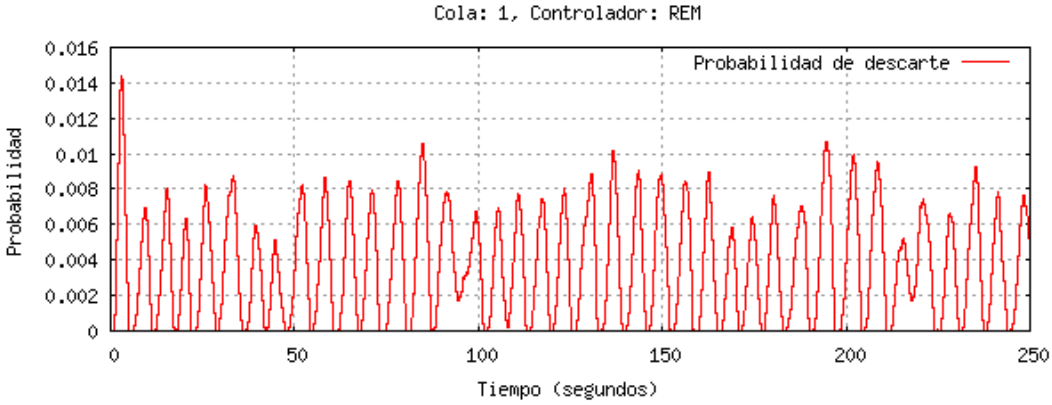


Figura 7.57. Probabilidad de descarte cola 1-REM

Cola 2

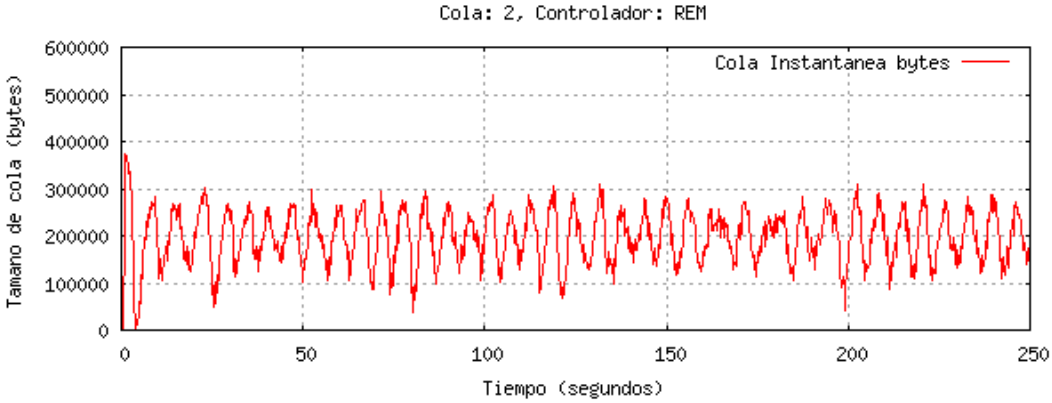


Figura 7.58. Cola instantánea cola 2-REM

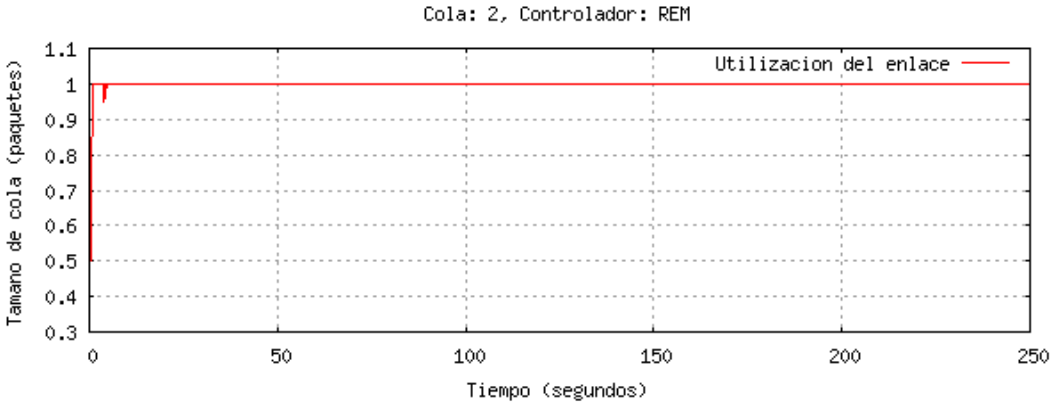


Figura 7.59. Utilización del enlace cola 2-REM

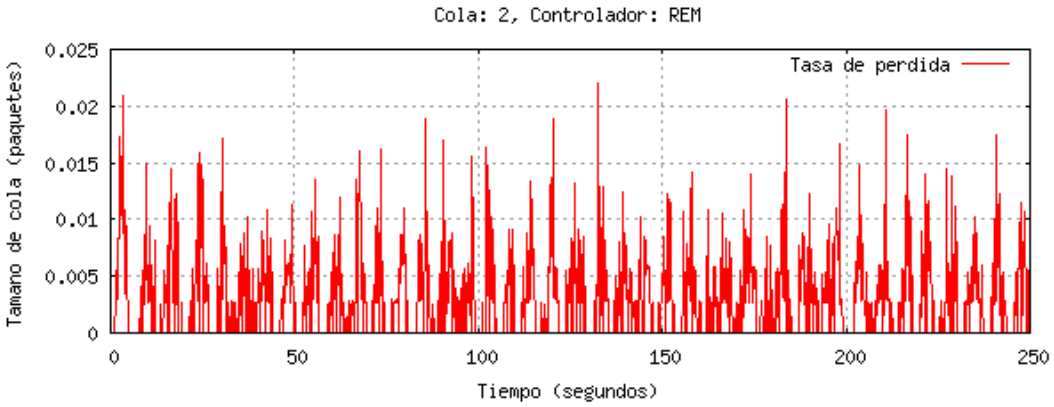


Figura 7.60. Tasa de pérdida cola 2-REM

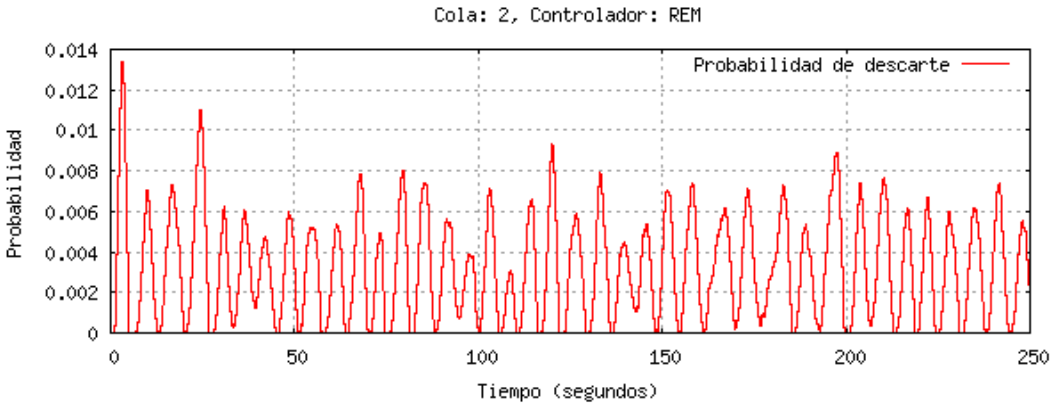


Figura 7.61. Probabilidad de descarte cola 2-REM

7.4.3 PI

Cola 1

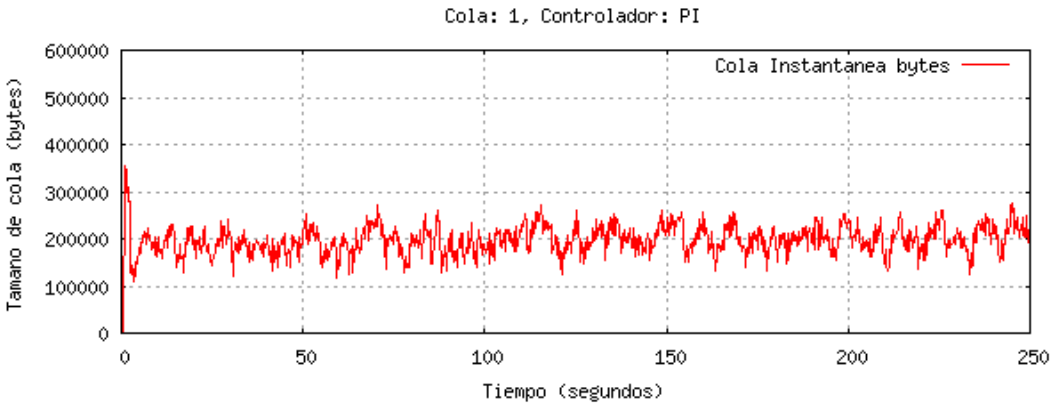


Figura 7.62. Cola instantánea cola 1-PI

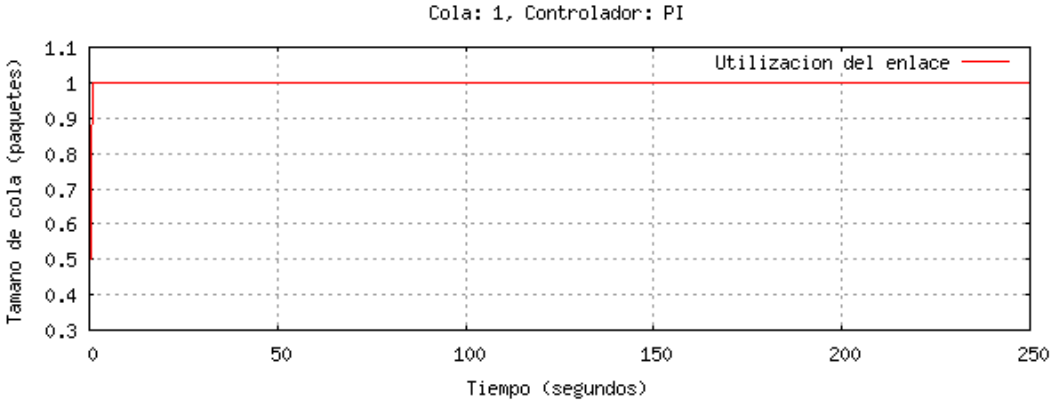


Figura 7.63. Utilización del enlace cola 1-PI

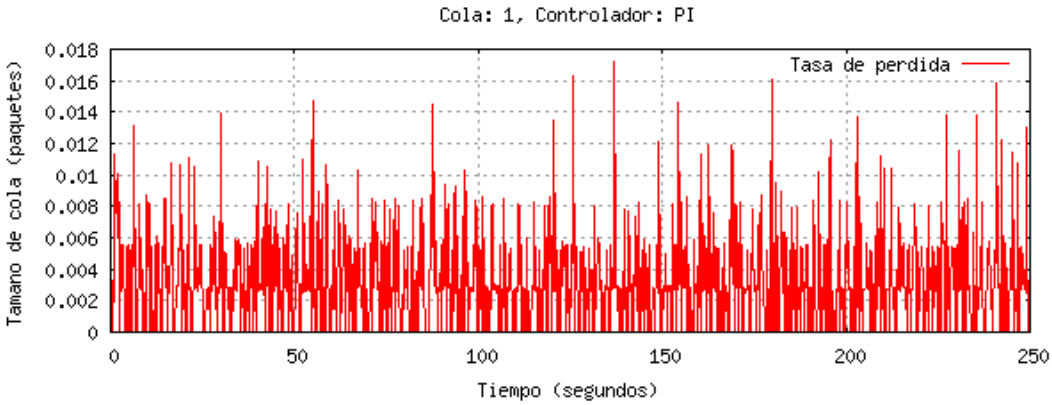


Figura 7.64. Tasa de pérdida cola 1-PI

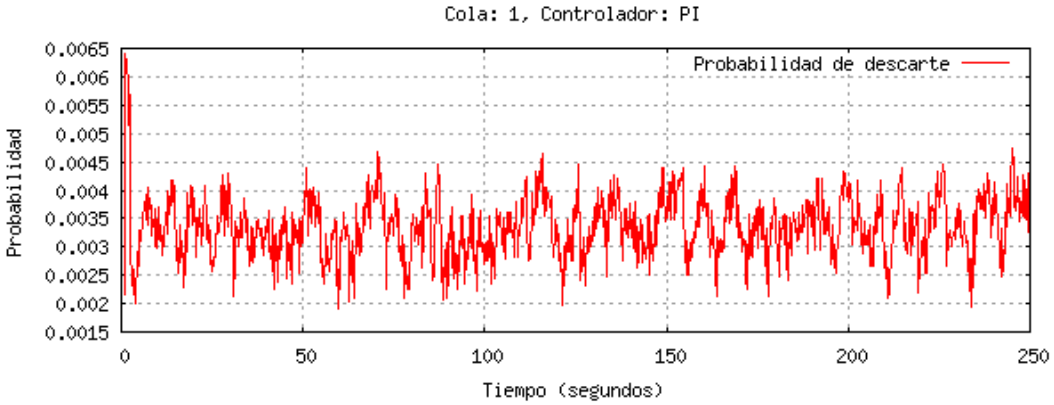


Figura 7.65. Probabilidad de descarte cola 1-PI

Cola 2

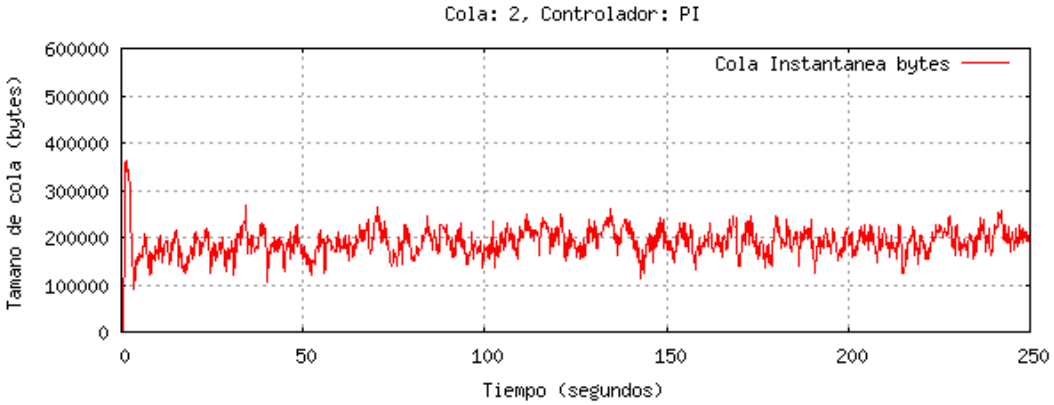


Figura 7.66. Cola instantánea cola 2-PI

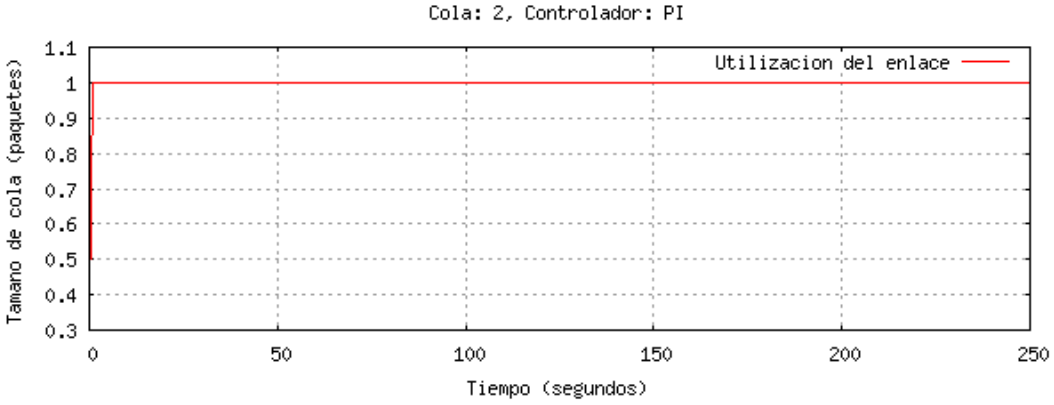


Figura 7.67. Utilización del enlace cola 2-PI

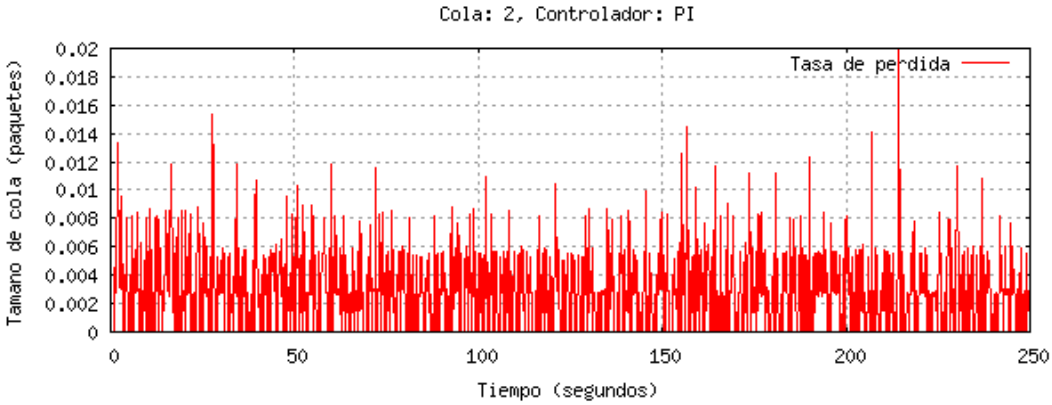


Figura 7.68. Tasa de pérdida cola 2-PI

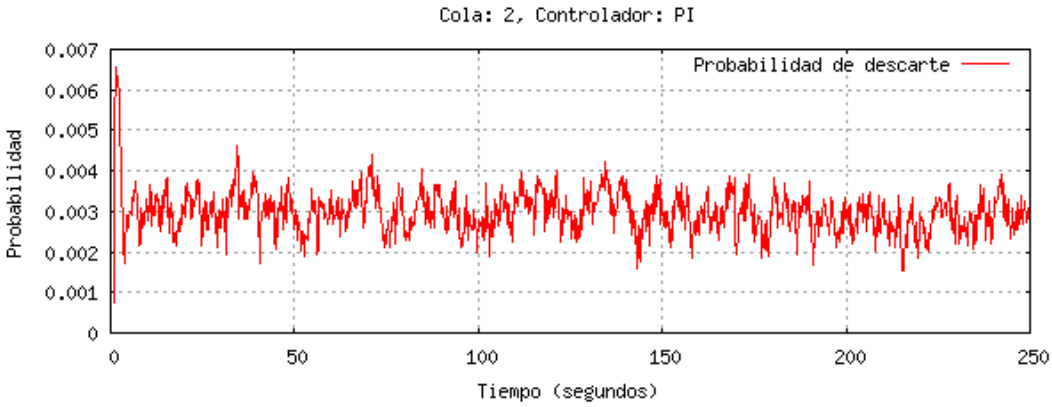


Figura 7.69. Probabilidad de descarte cola 2-PI

7.4.3 Análisis detallado

Para el análisis de los datos se recogen los resultados correspondientes a la cola 4 y se presentan en la tabla mostrada a continuación.

Algoritmo	RED	REM	PI
Tamaño Media Cola	119.604	197.954	194.822
Tasa perdidos	0.00566408	0.0033023	0.00317154
Desviación Perdidos	0.00483853	0.00391606	0.00293889
Probabilidad Descarte	0.0103319	0.00366363	0.00326748
Desviación de probabilidad	0.00625133	0.002482	0.000497031
Utilización del enlace	0.99976	0.999999	0.999999

Tabla 7.4. Valores significativos Experimento IV

En la tabla se puede apreciar como:

- Todos hacen uso del canal de forma similar, aprovechando al máximo el canal disponible.
- La tasa de paquetes perdidos y la desviación del mismo son inferiores para el algoritmo PI.
- La probabilidad de descarte y la desviación de la probabilidad son también inferiores en el algoritmo PI.

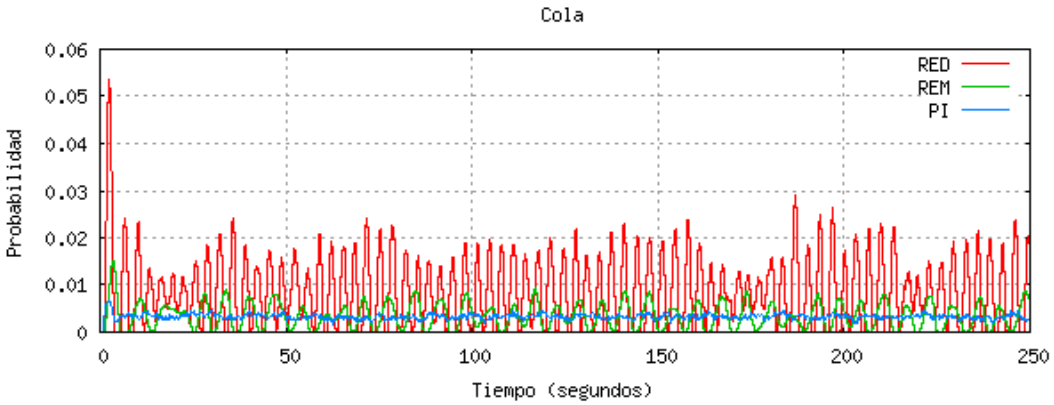


Figura 7.70. Probabilidad de descarte

Según los datos anteriores se puede llegar a la conclusión de que el algoritmo PI es el que mejor comportamiento tiene.

Comparación de resultados

A continuación se muestran las gráficas, con la evolución de la cola, para los algoritmos RED, REM y PI.

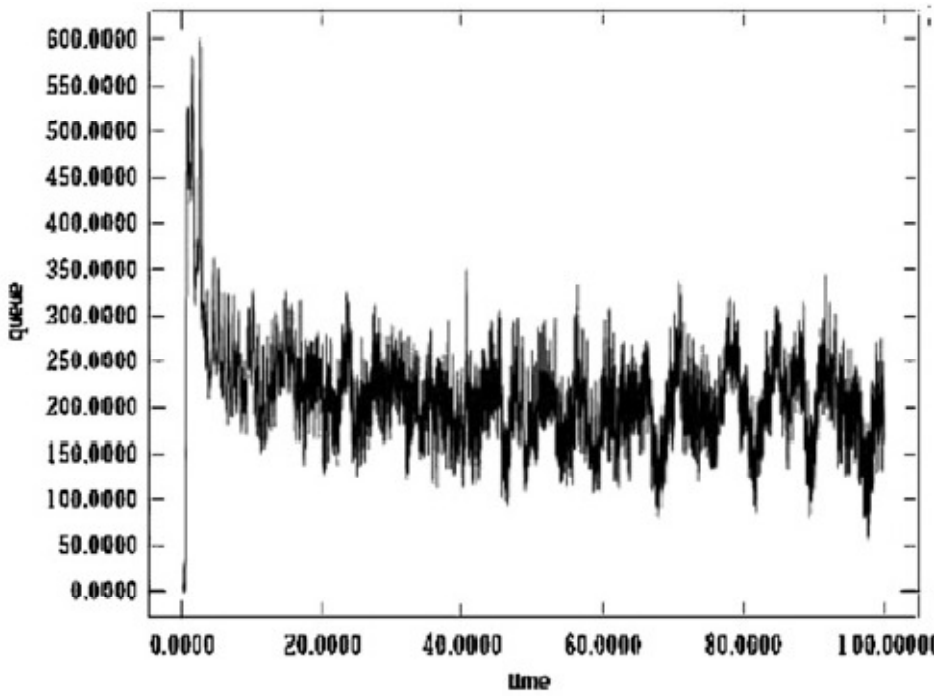


Figura 7.71. Cola instantánea-PI

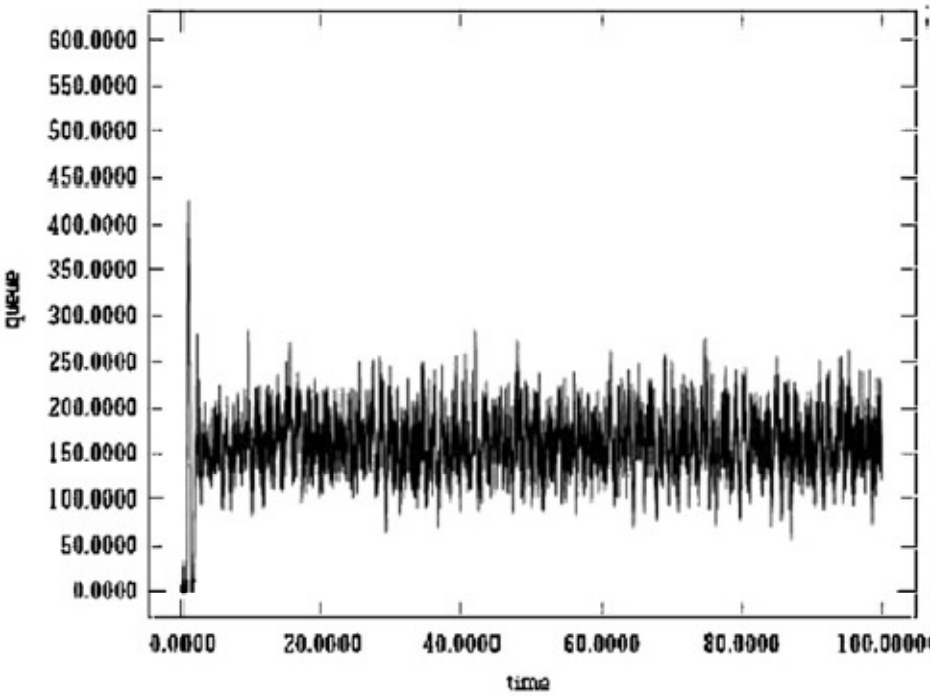


Figura 7.72. Cola instantánea RED

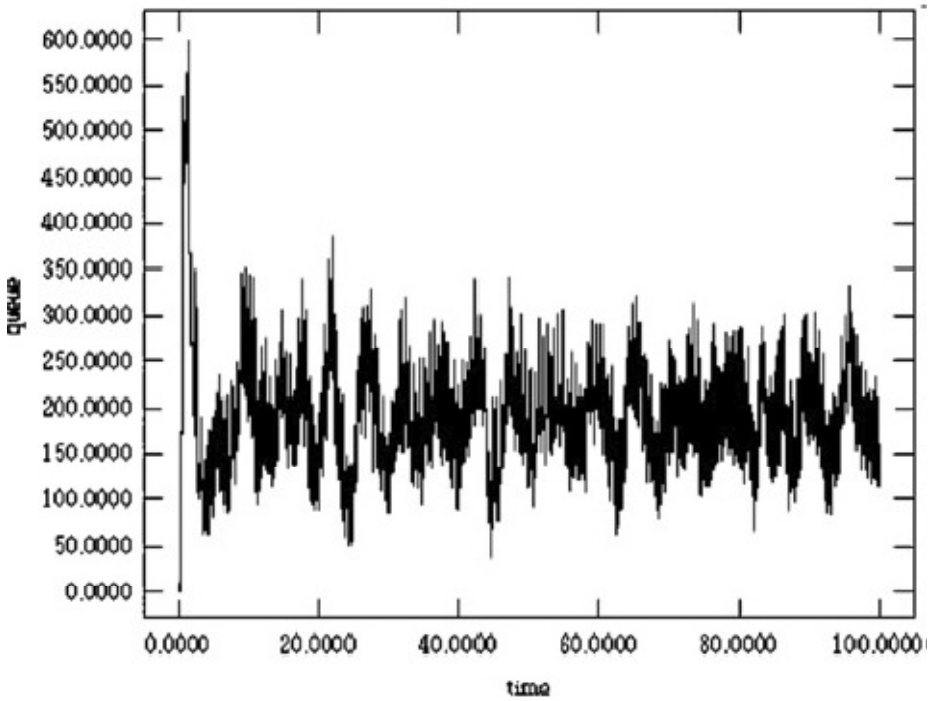


Figura 7.73. Cola instantánea REM

Estas gráficas muestran la evolución del tamaño de la cola a lo largo de la simulación. Como se puede observar, los datos del artículo son muy similares a la **figura 7.46**, la **figura 7.54** y la **figura 7.62**, ratificándose unos correctos resultados.

También se incluyen las gráficas que contienen la media de la cola y la desviación de cada cola analizada.

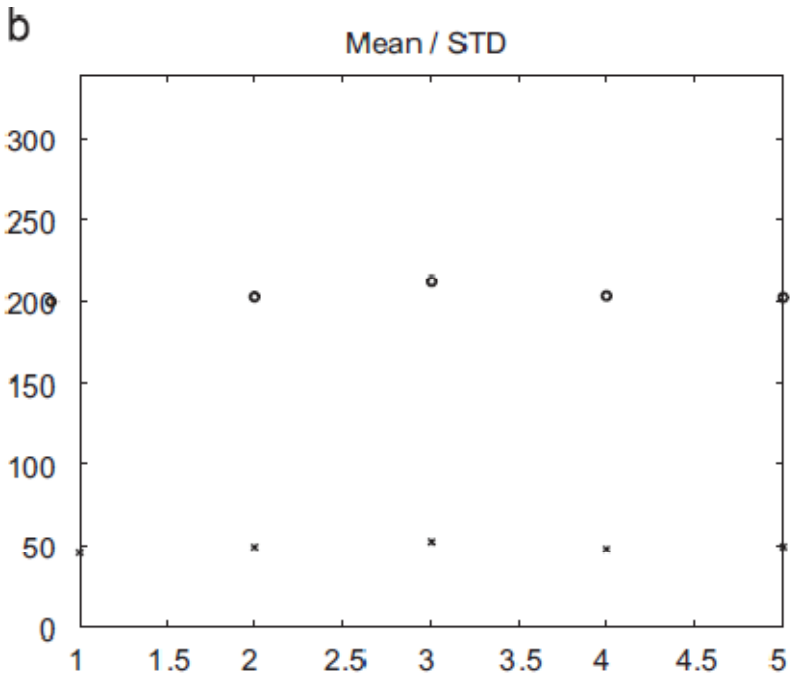


Figura 7.74. Media/Desviación PI

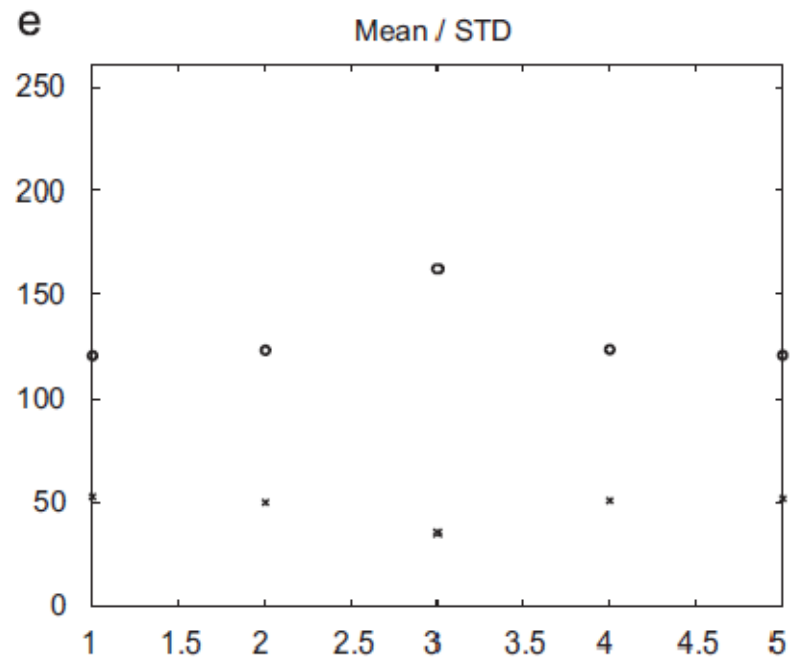


Figura 7.75. Media/Desviación RED

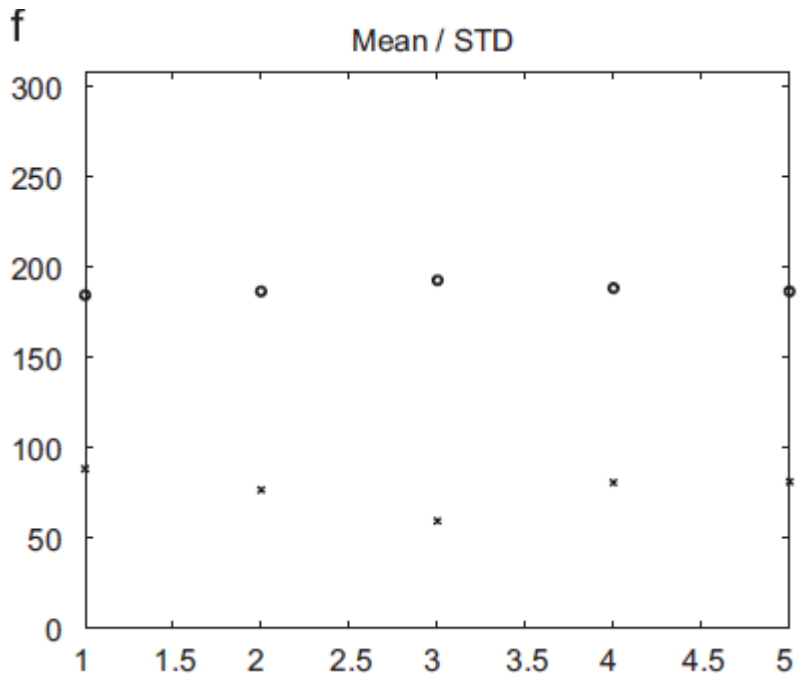


Figura 7.76. Media/Desviación REM

A continuación se muestran los datos de las medias y desviaciones de las colas para los algoritmos simulados:

	RED	REM	PI
Cola1			
MediaCola: DesvCola:	120.241 46.1312	194.861 64.3919	197.51 27.7936
Cola 2			
Mediacola: DesvCola:	118.539 42.3276	198.43 52.0967	191.139 25.1635
Cola 3			
MediaCola: DesvCola:	120.008 41.4326	198.823 52.2343	196.659 27.199
Cola 4			
MediaCola: DesvCola:	119.604 44.498	197.954 54.876	194.822 27.3362
Cola 5			
MediaCola: DesvCola:	119.405 46.711	196.486 54.7165	193.752 25.5151

Tabla 7.5. Valores significativos comparación

Como puede apreciarse, las medias de las colas mantienen la misma tendencia ascendente hasta la cola 3, y descendente, a partir de ella.

La desviación del algoritmo PI mantiene más o menos constantes sus valores, aunque inferiores a los obtenidos, mientras que las desviaciones de RED y REM mantienen la misma tendencia descendente hasta la cola 3 y ascendente a partir de la misma, como puede apreciarse en los datos.

7.5. Experimento V

En este experimento se han empleado los algoritmos RED,ARED,PID,FRED y BLUE, ya que estos han mostrado unos buenos resultados como se ha visto previamente.

Para no saturar la documentación de gráficas, sólo se mostrarán las gráficas de la cola instantánea, encontrándose en la documentación de soporte digital incluida en el proyecto, las gráficas restantes.

Cola instantánea

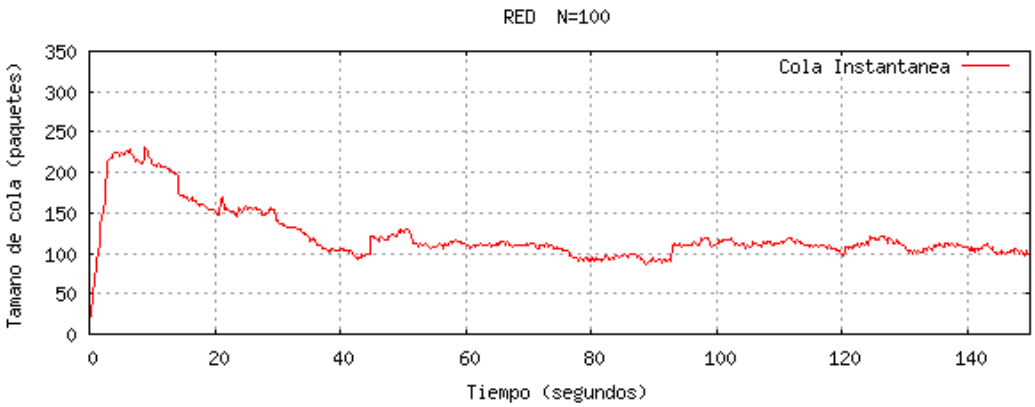


Figura 7.77. Cola instantánea RED

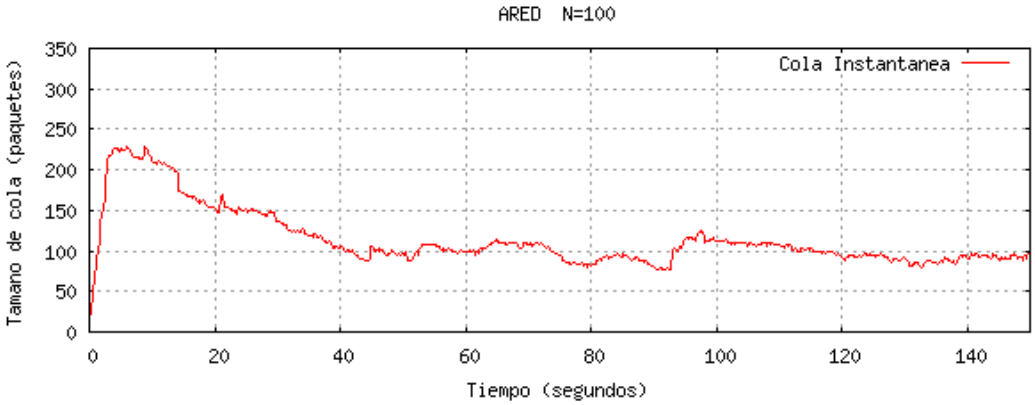


Figura 7.78. Cola instantánea ARED

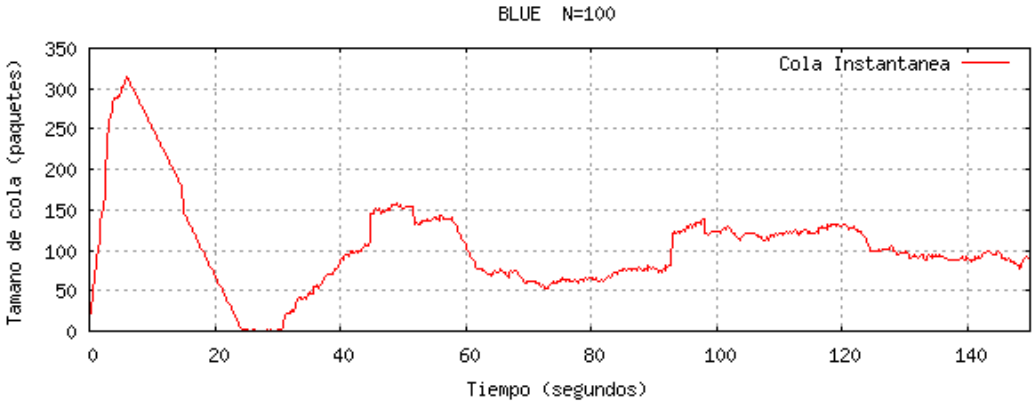


Figura 7.79. Cola instantánea BLUE

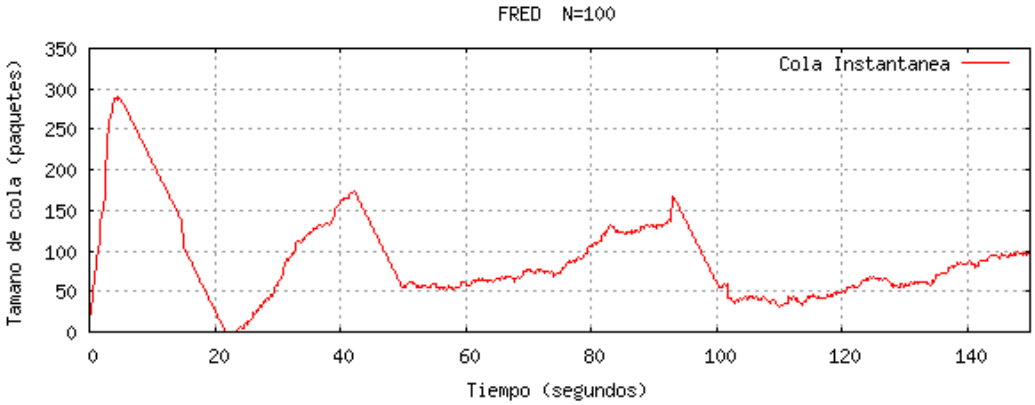


Figura 7.80. Cola instantánea FRED

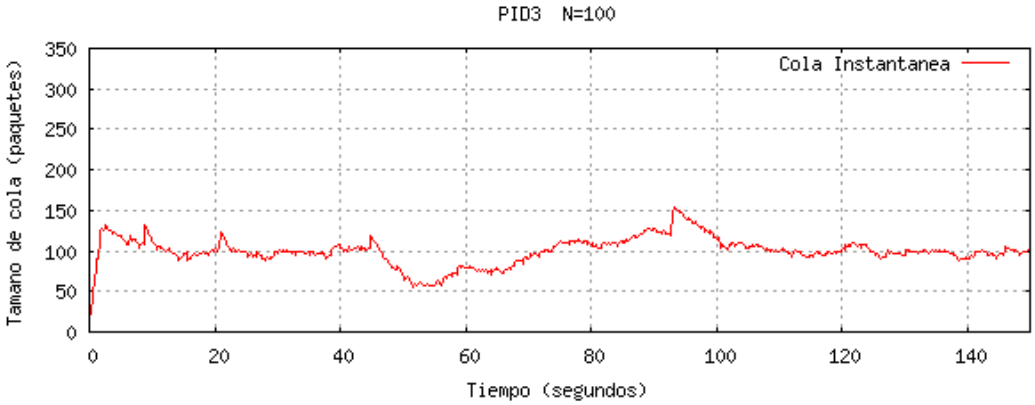


Figura 7.81. Cola instantánea PID

Como puede apreciarse, los algoritmos RED, ARED y PID son los que mejores resultados ofrecen. Mientras, el comportamiento de FRED y BLUE no es tan bueno como en los experimentos anteriores.

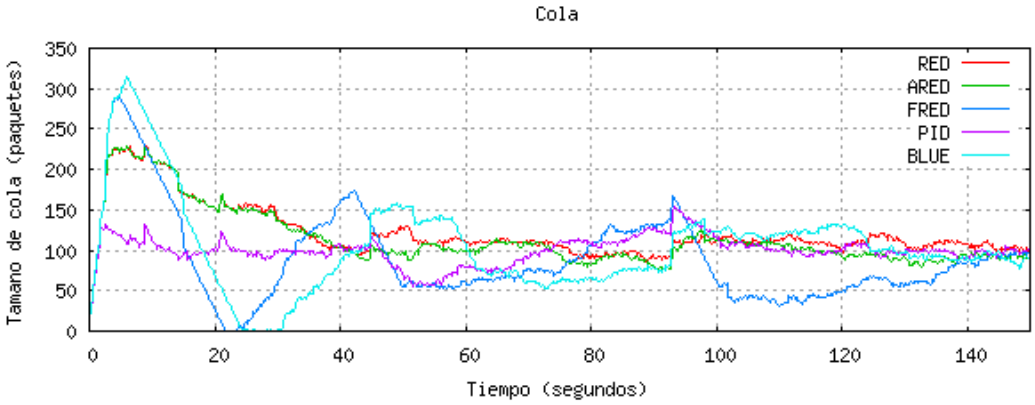


Figura 7.82. Colas instantáneas

Análisis detallado

Se comparan a continuación los valores más significativos de los algoritmos RED, ARED y PID, ya que son los que mejor se han comportado.

Algoritmo	RED	ARED	PID
Tamaño Media Cola	112.042	103.345	99.0511
Tasa perdidos	0.0988031	0.0935469	0.0760574
Desviación Perdidos	0.195199	0.174747	0.217751
Probabilidad Descarte	0.149621	0.184012	0.249419
Desviación de probabilidad	0.134859	0.148631	0.206371
Utilización del enlace	1	1	1

Tabla 7.6. Valores significativos Experimento V

- Las colas se mantienen entre los valores establecidos para RED y ARED (90 y 110), mientras PID se establece en torno al valor de referencia establecido.
- La tasa de perdidos y su desviación, tiene valores bastante parejos.
- La probabilidad de descarte es la mejor en el algoritmo RED, siendo superior para el algoritmo PID.
- La utilización del enlace es plena para los tres algoritmos empleados.

El algoritmo que mejor comportamiento presenta es RED, aunque ARED y PID tienen valores muy cercanos.

Parte IV

CAPITULO 8

8. Conclusiones

La congestión es uno de los problemas más importantes de las redes de ordenadores, puesto que cuando se produce un desbordamiento del *buffer,* el rendimiento del sistema se degrada, aumentando el retardo y la pérdida de paquetes. A fin de controlar este fenómeno, TCP cambia su tasa de transmisión de acuerdo con el nivel de congestión.

Por otro lado, el rápido crecimiento tanto de Internet como de las diversas y heterogéneas aplicaciones que se construyen alrededor de él, hacen que los antiguos mecanismos que se venían empleando para controlarlos vayan quedando poco a poco obsoletos. Tal es el caso de las estrategias de control de congestión, entre las cuales, el mecanismo tradicional de control extremo a extremo de TCP unido a la estrategia de planificación de la cola *Drop Tail,* se puede considerar insuficiente para las necesidades actuales.

Partiendo de esta situación, surge el planteamiento de cambiar la forma en la que los paquetes se descartan en los nodos con el fin de mejorar el rendimiento de TCP, naciendo entonces los algoritmos conocidos como AQM o *Active Queue Management.* En su inicio, estos algoritmos fueron creados para resolver, a medida, los problemas que se iban planteando con una observación directa del problema.

Los algoritmos AQM son implementados en los *routers* y permiten poder controlar el tamaño de sus colas teniendo en cuenta el número de paquetes que hay en sus *buffers* y, según ese número, descartar el paquete entrante o admitirlo. Uno de estos algoritmos AQM, que además es el único que ha sido propuesto como estándar por la IETF, es RED. Su rendimiento no ha demostrado ser el mejor y, en consecuencia, han proliferado desde la comunidad científica multitud de nuevos algoritmos para el control de las colas.

En este trabajo se ha tratado de analizar y comparar el funcionamiento de algunos de estos algoritmos AQM, así como de los métodos tradicionales *Drop Tail* y RED. Los algoritmos elegidos para su estudio han sido: ARED, GREEN, BLUE, FRED, SRED, REM, PI y PID.

En los dos primeros experimentos realizados se ha podido observar como los algoritmos BLUE, FRED, RED y ARED han tenido actuaciones muy parecidas aunque, con más detalle:

- BLUE y FRED mostraban un comportamiento muy similar, teniendo FRED un rendimiento ligeramente superior.
- RED y ARED han manifestado un funcionamiento similar, notándose en el experimentos 3, la mejora que supone ARED.
- PID ha tenido un buen comportamiento en ambos experimentos, como se indicó en el análisis detallado.

El experimento 3 permitió ver un mejor comportamiento en ARED, lo cual fue confirmado por los datos resultantes.

En el experimento 4 pudo verse como los algoritmos RED, REM y PI mantienen un buen rendimiento con un sistema multicuello de botella, sistema que se acerca más a la realidad, siendo el algoritmo PI el que tenía un mejor comportamiento.

Como punto final, en el experimento 5 se introducen nodos inalámbricos (aunque sin ningún patrón de movimiento) para estudiar su funcionamiento con los algoritmos FRED,BLUE,RED,ARED y PID. En esta ocasión, FRED y BLUE no tuvieron el comportamiento mostrado con anterioridad, mientras que RED, ARED y PID presentaron un rendimiento aceptable.

Al poder disponer de artículos con los que comparar los resultados obtenidos, los experimentos I,III y IV reflejan un correcto funcionamiento, lo que asegura unos datos bastante fiables.

Líneas de trabajo futuras

Debido a la gran cantidad de frentes abiertos en este campo y a la amplitud del mismo, los posibles caminos a seguir aparecen como innumerables. Aquí se nombran sólo alguna de las posibilidades más directas:

- Ampliar el estudio comparativo de los algoritmos escogidos para este trabajo a un mayor rango de controladores de ambas clases, tales como YELLOW, SFB (*Stochastic Fair* BLUE), *fuzzy* o control robusto.
- Realizar un estudio más amplio bajo topologías WLAN.
- Los algoritmos AQM están optimizados para flujos TCP constantes. Se debería hacer un estudio sobre su comportamiento ante tráfico *web* o tráfico UDP (ya que sólo se generó cierta cantidad de este tráfico en el experimento V), el cual no reacciona ante los descartes tempranos de paquetes.

Bibliografía

[ALG] Algozino R., Alincastro N., Corteggiano F., Magnago H., Gioda M. RED, Algoritmo de Control de Congestión en Redes IP. Universidad Nacional de Río Cuarto. Asociación Argentina de Mecánica Computacional. Mecánica Computacional Vol XXVIII. 3-6 Noviembre 2009.

[ALT1] Altman E., Jiménez T. NS Simulator Course for Beginners. http://www-sop.inria.fr/mistral/personnel/Eitan.Altman/ns.htm (Fecha última consulta: 19/08/2012).

[ALT2] Altman E., Jiménez T. NS Simulator for beginners. Lecture notes, 2003. Univ. de Los Andes, Mérida, Venezuela and ESSI, Sophia-Antipolis, France. December 4, 2003.

[ALV1] Álvarez Flores E. P. Descarte Selectivo de Paquetes en Mecanismos de Gestión Activa de Colas. Universidad de Granada. Departamento de Teoría de la Señal, Telemática y Comunicaciones. Editorial de la Universidad de Granada. Granada, España. Abril, 2009.

[ALV2] Álvarez Teresa. Design of PID Controllers for TCP/AQM Wireless Networks. *Department of Engineering Science and Automatic Control, Escuela de Ingenierías Industriales (Sede Doctor Mergelina), Universidad de Valladolid, Valladolid, Spain.* 2012.

[ALV3] Álvarez Teresa, Salim Anuar, Maestre J.M. A control theoretical approach to congestion control of TCP/AQM networks. *Department of Engineering Science and Automatic Control, Escuela de Ingenierías Industriales (Sede Doctor Mergelina), Universidad de Valladolid, Valladolid, Spain.* 2012.

[BUR] Burri S. BLUE: Active Queue Management CS756 Project Report. May 5, 2004. http://www.thefengs.com/wuchang/work/blue/burri04blue.pdf (Fecha última consulta: 10/08/2012)

[CHA] Chang J., Hu N., Ren L. Evaluation of Queue Management Algorithms. Course Project Report for 15-744 Computer Networks. 30 April 2001.

http://www.cs.cmu.edu/~hnn/cs744_project/project.html
(Fecha última consulta: 10/08/2012).

[CHU] Chung J., Claypool M. NS by Example. http://nile.wpi.edu/NS/ (Fecha última consulta: 11/08/2012).

[FAH] Fahmy S., Kwon J. S. Active Queue Management. http://www.cs.purdue.edu/homes/fahmy/software/aqm/ (Fecha última consulta: 01/08/2012).

[FAL] Fall K. Congestion Control Metrics. TCP Congestion Control. EECS 122, Lecture 21. http://www.cs.berkeley.edu/~kfall/EE122/lec21/sld001.htm (Fecha última consulta: 01/08/2012).

[FEN] Feng W., Kandlur D. D., Saha D., Shin K. G. BLUE: A New Class of Active Queue Management Algorithms. U. Michigan CSE-TR-387-99, April 1999. http://www.cs.ust.hk/faculty/bli/660h/feng99blue.pdf (Fecha última consulta 01/08/2012)

[GNU] Gnuplot homepage. Página principal. http://www.gnuplot.info/ (Fecha última consulta: 02/07/2012).

[GRE] Greis Marck. Tutorial for the Network Simulator ns.Página principal. http://www.isi.edu/nsnam/ns/tutorial/index.html (Fecha última consulta: 02/07/2012).

[INS1] Installing ns2.34 on ubuntu 11.04. http://erl1.wordpress.com/2011/05/12/installing-ns-2-34-on-ubuntu-11-04/ (Fecha última consulta: 01/07/2012)

[INS2] Installing ns2.34 on ubuntu 11.10. http://erl1.wordpress.com/2011/10/14/installing-ns-2-34-on-ubuntu-11-10-oneiric-ocelot/ (Fecha última consulta: 01/07/2012)

[JIN] Jinsheng Sun, Moshe Zukerman and Marimuthu Palaniswami. A Stable Adaptive PI Controller for AQM. Department of Electrical and Electronic Engineering, The University of Melbourne, Victoria, 3010, Australia

[KAP] Kapadia A., Feng W., Campbell R. H. GREEN: A TCP Equation-Based Approach to Active Queue Management. Dept. of Computer Science University of Illinois at Urbana-Champaign 201 N. Goodwin Urbana, IL 61801. https://www.ideals.illinois.edu/bitstream/handle/2142/10780/GREEN%20-%20A%20TCP%20Equation-Based%20Approach%20to%20Active%20Queue%20Management.pdf?sequence=2 (Fecha última consulta: 02/07/2012)

[MAN] Manfredi Sabato, Mariodi Bernardo y Franco Garofalo. Design, validation and experimental testing of a robust AQM control. Faculty of Engineering, University of Naples Federico II, Via Claudio 21, Napoli 80100, Italy. 22 de Noviembre de 2008.

[MAR1] Martínez O., Palau C. Introducción a la Programación de Protocolos de Comunicaciones con Network Simulator 2. Editorial Club Universitario. San Vicente (Alicante), España.

[MAR2] Marami B., Bigdeli N., Haeri M. Active Queue Management of TCP/IP Networks Using Rule-Based Predictive Control. Advanced Control System Lab. Electrical Engineering Department, Sharif University of Technology Tehran, Iran.

[PLE1] Pletka Roman, Marcel Waldvogel y Soenke Mannal. PURPLE: Predictive Active Queue Management Utilizing Congestion Information. IBM Research,Zurich Research Laboratory. 20-24 Oct. 2003.

[PLE2] Pletka Roman, Andreas Kindy, Marcel Waldvogely and Soenke Mannal. Closed-Loop Congestion Control for Mixed Responsive and Non-Responsive Traffic.IBM Research,Zurich Research Laboratory. Globelcom 2003

[RYU1] Ryu S., Rump C., Qiao C. Advances in Active Queue Management (AQM) Based TCP Congestion Control. Telecommunication Systems, Volume 25, Numbers 3-4, March 2004, pp. 317-351(35).

[RYU2] Ryu S., Rump C., Qiao C. Advances In Internet Congestion Control. State University of New York at Buffalo. March 2004.

[SAL] Salim del Río, Anuar. Algoritmos AQM basados en teoría de control. Universidad de Valladolid. 2008

[THE] The Network Simulator NS-2, Home page. Página principal. http://www.isi.edu/nsnam/ns/ (Fecha última consulta: 01/07/2012).

[THI] Thiruchelvi G. y J.Raja. A Survey On Active Queue Management Mechanisms. IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.12, Diciembre 2008

[WAN] Lijun Wang, Lin Cai, Xinzhi Liu, Xuemin (Sherman) Shen, Junshan Zhang. Stability analysis of multiple-bottleneck networks. 1 de Noviembre de 2008.

Lista de abreviaturas

ACK	Acknowledgement
AQM	Active Queue Management
ARED	Adaptive RED
CBR	Constant Bit Rate
cwnd	Congestion Window
rwnd	Receiver Window
DT	Drop Tail
ECN	Explicit Congestion Notification
EWMA	Exponentially Weighted Moving Average
FIFO	First-In, First-Out
FRED	Flow Random Early Drop
GPC	Generalized Predictive Control
GRED	Gentle RED
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISN	Initial Sequence Number
JFI	Jain's Fairness Index
MSS	Maximun Segment Size
NACK	Negative Acknowledgement
NAM	Network Animator
NS	Network Simulator
OTCL	Object Tool Command Language
P	Proporcional
PDU	Protocol Data Unit
PI	Proporcional, Integral
PID	Proporcional, Integral, Derivativo
QACD	Quadratic Average of Control Deviation
RED	Random Early Detection
REM	Random Exponential Marking
RFC	Request for Comments
RIO	RED with In/Out
RTT	Round Trip Time
SFB	Stochastic Fair BLUE
SRED	Stabilized Random Early Drop
TCL	Tool Command Language
TCP	Transmission Control Protocol
TTL	Time to Live

Anexos

Anexo A

Instalación de ns-2 en Ubuntu 11.10

Las razón por la que se incluyó este anexo fue por las dificultades encontradas para instalar ns 2.34 en la versión Ubuntu 11.10, y así facilitar el trabajo para posteriores instalaciones.

La aplicación que conforma el simulador NS realmente está compuesta por varios componentes. Hay dos formas de descargar e instalar NS. Se puede hacer cada componente por separado, o mediante el paquete denominado *all-in-one*, con todos los componentes a la vez. La segunda opción es la más sencilla, ya que se hace en un sólo paso y evita problemas de dependencias entre componentes. Será ésta la que explicaremos aquí.

Supondremos que realizaremos la instalación como usuario en nuestro directorio personal, y que nuestro nombre de usuario es *miusuario*. Por tanto nuestro directorio personal estará situado en */home/miusuario*.

Si lo decidimos hacer como *root*, hay que tener en cuenta que los usuarios normales no podrán ejecutar correctamente NS a no ser que se le den los permisos adecuados. Una opción es cambiar con *chown* el propietario del directorio de instalación al usuario que lo usará.

Descarga de las fuentes

Actualmente se encuentra en su versión 2.35 (pero descargaremos la versión 2.34) y se encuentra alojado en la dirección http://sourceforge.net/projects/nsnam/files/. Supondremos que descargamos la versión 2.34.tar.gz e indicaremos los pasos a realizar con el nombre del archivo anterior

Instalación

Lo primero de todo es asegurarnos de que disponemos de los paquetes que necesita el script de instalación para poder compilar y configurar. Para instalar estos paquetes en Ubuntu escribiremos en el terminal:

```
$ sudo apt-get install build-essential autoconf automake libxmu-dev
```

Nos pedirá la contraseña de nuestro usuario, la escribimos y pulsamos intro. Después si disponemos de conexión a Internet se instalarán automáticamente los paquetes indicados.

Una vez hecho esto vamos a nuestro directorio personal (si no lo estamos ya) donde tenemos el fichero que nos hemos descargado con las fuentes de NS-2. El siguiente paso será descomprimirlo, para ello:

```
$ cd
```

```
$ tar −xzvf ns−allinone−2.34.tar.gz
```

Se creará un directorio llamado ns-allinone-2.34 en el que tendremos los archivos descomprimidos. Accedemos al nuevo directorio:

```
$ cd ns−allinone−2.34
```

Una vez dentro del directorio, podemos ver el contenido del fichero README. Antes de ejecutar el script de instalación debemos realizar una serie de cambios ya que nos encontramos en la versión 11.10 de Ubuntu.

Instalar los archivos de desarrollo para Windows X más el compilador g++:

```
$ sudo apt-get install xorg-dev g++ xgraph
```

Se debe corregir el error en la vinculación de *otcl*, para ello se debe editar la línea 6304 del fichero *otcl-1.13/configure* para que se tenga

SHLIB_LD=”gcc -shared”

en lugar de

SHLIB_LD=”ld -shared”

A continuación se debe editar el archivo *ns-2.34/tools/ranvar.cc* y cambiar la línea 219

*return GammaRandomVariable::GammaRandomVariable(1.0 + alpha_, beta_).value() * pow (u, 1.0 / alpha_);*

a

*return GammaRandomVariable(1.0 + alpha_, beta_).value() * pow (u, 1.0 / alpha_);*

Después se debe cambiar las líneas 183 y 185 en el archivo *ns-2.34/mobile/nakagami.cc* resultando lo siguiente:

resultPower = ErlangRandomVariable(Pr/m, int_m).value();

y

resultPower = GammaRandomVariable(m, Pr/m).value();

Añadir una línea después de la línea 64 en *ns-2.34/mac/mac-802_11Ext.h*:

#include <stddef.h>

Ahora se puede ejecutar el script *./install*. Sin embargo puede no ser capaz de ejecutar el archivo ejecutable ns. Si se recibe un error indicando que ha habido un desbordamiento de búffer **** buffer overflow detected *** ./ns terminated* se debe realizar lo siguiente:

Instalar gcc-4.4 y g++-4.4 , incluidas las dependencias, ejecutando el siguiente código.

```
$sudo apt-get install gcc-4.4 g++-4.4
```

Después es necesario cambiar la línea 270 en *tcl8.4.18/unix/Makefile*.in que tiene:

CC = @CC@

por esta otra linea

CC = gcc-4.4

Una vez hecho esto ya se puede volver a ejecutar *./install* desde el directorio anterior sin ningún problema.

```
$ ./install
```

Una vez hecho esto, la instalación se habrá completado y se mostrará por terminal información sobre el estado de la instalación. Entre los datos que se muestran, encontramos una lista de las versiones de los componentes de NS y de sus rutas de instalación, será necesario anotar estos datos.

También se indica que habrá que modificar las variables de entorno. En el siguiente apartado explicamos cómo hacerlo.

Configuración de las variables de entorno

Para una correcta ejecución del simulador es necesario actualizar las variables de entorno. Para hacerlo añadiremos las líneas necesarias al fichero *.bashrc* de nuestro directorio personal para que las variables de entorno se actualicen cada vez que hagamos un login.

Primero nos aseguraremos de que estamos en nuestro directorio personal:

```
$ cd
```

Y abrimos el fichero *.bashrc* con nuestro editor de textos preferido, hemos elegido el gedit, pero se podrá hacer igualmente con vi, emacs, nano, etc.:

```
$ gedit . bashrc
```

Al final de este fichero deberemos añadir las siguientes líneas, pero sustituyendo *miusuario*, por el nombre de usuario que tengamos en la máquina

```
# LD_LIBRARY_PATH
OTCL_LIB =/ home / miusuario / ns - allinone -2.34/ otcl -1.13
NS2_LIB =/ home / miusuario / ns - allinone -2.34/ lib
X11_LIB =/ usr / X11R6 / lib
USR_LOCAL_LIB =/ usr / local / lib
export LD_LIBRARY_PATH = $LD_LIBRARY_PATH : $OTCL_LIB :$NS2_LIB : $X11_LIB :
$USR_LOCAL_LIB

# TCL_LIBRARY
TCL_LIB =/ home / miusuario / ns - allinone -2.34/ tcl8 .4.18/library
USR_LIB =/ usr / lib
export TCL_LIBRARY = $TCL_LIB : $USR_LIB

# PATH

XGRAPH =/ home / miusuario / ns - allinone -2.34/ bin :/ home /miusuario / ns - allinone
-2.34/ tcl8 .4.18/ unix :/ home /miusuario / ns - allinone -2.34/ tk8 .4.18/ unix
NS =/ home / miusuario / ns - allinone -2.34/ ns -2.34/
NAM =/ home / miusuario / ns - allinone -2.34/ nam -1.14/
PATH = $PATH : $XGRAPH : $NS : $NAM
export PATH
```

NOTA IMPORTANTE: Estos valores para las variables son válidos en la versión 2.34. En el caso de usar otra versión habrá que fijarse en si las versiones de los componentes de las líneas anteriores coinciden con las de la versión descargada (se puede comprobar

observando el nombre de los o subdirectorios incluidos dentro de ns-allinone-2.xx). En cualquier caso, los valores que deben tener estas variables se muestran al final de la ejecución del script de instalación *install*, como hemos mencionado antes.

Validación

Hay un script en la distribución de NS que permite comprobar si la instalación ha sido correcta y el simulador funciona bien. Se basa en la realización de simulaciones y comparativa con los resultados esperados. Se puede ejecutar asegurarnos del correcto funcionamiento aunque puede durar bastante tiempo:

```
$ cd /home/miusuario/ns−allinone−2.34/ns−2.34

$ ./validate
```

Anexo B

Extensión del simulador NS

Inclusión de una nueva cola

Primero copiaremos los ficheros .h y .cc al directorio en el que se alojan las colas en NS, esto es, en *ns-allinone-2.34/ns-2.34/queue*, y nos aseguramos de que tienen los permisos correctos. Estando situados en el directorio que contiene los ficheros que queremos incluir, escribimos en la terminal:

```
$ cp pid3.h pid3.cc /home/miusuario/ns-allinone2.34/ns2.34/queue
```

```
$ chmod 644 /home/miusuario/ns-allinone2.34/ns2.34/queue/pid3.*
```

Una vez copiados tenemos que añadir las variables (las variables que se añaden son únicamente las que están ligadas a Tcl) con sus valores por defecto al fichero *ns-default.tcl*. Abrimos el fichero con un editor de texto (gedit):

```
$ gedit/home/miusuario/ns-allinone2.34/ns2.34/tcl/lib/ns-default.tcl
```

Y añadimos las siguientes líneas al final del fichero:

```
Queue/PID3 set bytes_ false
Queue/PID3 set queue_in_bytes_ false
Queue/PID3 set typepid_ 3
Queue/PID3 set kp_ 2.8
Queue/PID3 set ki_ 2.54
Queue/PID3 set kd_ 0.84
Queue/PID3 set ti_ 1.10
Queue/PID3 set td_ 0.3
Queue/PID3 set zn_ false
Queue/PID3 set w_ 170
Queue/PID3 set qref_ 50
Queue/PID3 set mean_pktsize_ 500
Queue/PID3 set setbit_ false
Queue/PID3 set prob_ 0
Queue/PID3 set curq_ 0
Queue/PID3 set aveq_ 0
Queue/PID3 set earlydrops_ 0
Queue/PID3 set forceddrops_ 0
Queue/PID3 set newpackets_ 0
Queue/PID3 set totalarrivals_ 0
```

```
Queue/PID3 set with_ave_ 0
Queue/PID3 set ewma_weight_ 0 .002
Queue/PID3 set type_d_b_ 1.0
Queue/PID3 set type_d_n_ 8.0
```

El siguiente paso consiste en modificar el *Makefile* para indicar que hay que compilar los nuevos ficheros que hemos añadido. Para ello abrimos el *Makefile* del componente principal de NS con el editor de textos:

```
$ cd/home/miusuario/ns-allinone2.34/ns2.34

$ gedit Makefile
```

Y buscamos la zona del fichero donde se le asigna el valor a la variable OBJ CC, que indica dependencias de compilación. Como vemos esta asignación se prolonga durante decenas de líneas, continuando a la siguiente con la barra invertida (\). abajo podemos ver resumido como es la disposición de esa parte del fichero en la versión 2.34 de NS:

```
OBJ_CC = \
        tools / random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-
        handler .o \
        ...
        ...
        apps /pbc.o \
        $( OBJ_STL )
```

Una vez localizado buscamos la última línea de la asignación *($(OBJ STL))*. Y antes de esta línea indicamos la ruta del fichero que acabamos de añadir, esta vez acabado en *.o*, haciendo referencia al fichero objeto (*pid3.o*). El fichero modificado quedaría de la siguiente forma:

```
OBJ_CC = \
        tools / random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-
        handler .o \
        ...
        ...
        apps /pbc.o \
        queue/pid3.o \
        $(OBJ_STL)
```

Para analizar nos aseguramos de que estamos en el directorio correcto (donde se encuentra el *Makefile* que acabamos de editar) y ejecutamos el comando *make*, que realizará la compilación:

```
$ cd /home/miusuario/ns-allinone-2.34/ns-2.34

$ make
```

Es posible que en este punto nos muestre un error de conversión (si no es así ya hemos terminado y podemos omitir este párrafo). Esto es producido porque las nuevas versiones del compilador de linux (GCC) muestran como errores algo que en las versiones anteriores eran sólo avisos (*Warnings*). Para solucionarlo debemos volver a editar el *Makefile* y buscar la siguiente línea:

```
CCOPT = -Wall -Wno -write – strings
```

Una vez localizada debemos añadir el flag *-fpermissive*, que evitará que el compilador de ese error. De tal forma que la línea quedaría de la siguiente forma:

```
CCOPT = -Wall -Wno -write - strings –fpermissive
```

Una vez hecho esto deberíamos volver a ejecutar el comando *make* de la misma forma que hemos visto antes, comprobando que en este caso ya no se producen errores de *compilación*.

Anexo C

Contenido del CD

El CD que acompaña a esta memoria tiene la siguiente estructura:

- Memoria: En este directorio se encuentra la memoria del proyecto en formato pdf.
- Fuentes:
  - *ns-allinone-2.34.tar.gz*: Código fuente de la versión de NS-2 sobre la que se trabajó durante el desarrollo de este proyecto.
- Extensión: Este directorio contiene los archivos que han sido necesarios modificar o que se han añadido al simulador.
  - *Makefile*: Usado en la compilación del NS-2 después de la modificación.
  - *ns-default.tcl*: Fichero con los valores por defecto de las clases OTcl del simulador. En este fichero se han añadido los de nuestras nuevas clases.
  - Colas: Dentro de este directorio se encuentran los algoritmos AQM necesarios para extender el simulador. Para acceder a dichos ficheros sólo tenemos que colocarnos en la carpeta del algoritmo que nos interese (por ejemplo, extensión/colas/fred). Cada algoritmo incluye:
    - *Green:*
      - *green.cc*, *green.h*: Clases C++ relativas al algoritmo GREEN.
      - n*s-default-green.tcl*: Valores por defecto de los atributos de Tcl de GREEN.
    - Blue:
      - *blue.cc*, *blue.h*: Clases C++ relativas al algoritmo BLUE.
      - *ns-default-blue.tcl*: Valores por defecto de los atributos de Tcl de BLUE.
    - Fred:
      - *fred.cc*, *fred.h*: Clases C++ relativas al algoritmo FRED.
      - *ns-default-fred.tcl*: Valores por defecto de los atributos de Tcl de FRED.
    - Sred:
      - *sred.cc*, *sred.h*: Clases C++ relativas al algoritmo SRED.

- *ns-default-sred.tcl*: Valores por defecto de los atributos de Tcl de SRED.

- Pid3:
  - *pid3.cc*, *pid3.h*: Clases C++ relativas al controlador PID3.
  - *ns-default-pid3.tcl*: Valores por defecto de los atributos de Tcl del PID3.

- Simulaciones: Este directorio contiene los datos extraídos de las simulaciones mostradas en este trabajo así como los archivos necesarios para la simulación. La carpeta está dividida en 4 subcarpetas, cada una relativa a cada experimento:
  - EXP1: contiene los datos del primer experimento. A su vez contiene:
    - DATOS: contiene una carpeta por cada algoritmo con el que se ha desarrollado el experimento, por ejemplo vemos el caso de ARED, dentro de ARED están los ficheros (las trazas y el fichero medias con los cálculos realizados) resultados de la simulación y la carpeta img, que contiene las graficas de dicha simulación.
    - Simulación.tcl: el fichero tcl que define la simulación para el NS-2.
    - Preparadatos: La función de este script es la de calcular índices y separar en ficheros distintos los datos de una simulación dejándolos preparados para mostrarlos en una gráfica o su uso para comparaciones u otros medios.
  - EXP2: misma estructura que el anterior.
  - EXP3: misma estructura que EXP1, salvo que la carpeta DATOS contiene dos carpetas respectivas a los muestreos 0.001 y 0.2013. Cada una en su interior contiene 6 carpetas, correspondientes a las simulaciones con N=40,60 y 80 y los algoritmos RED y ARED. Cada una de estas 6 carpetas contienen los ficheros resultados de las simulaciones así como las gráficas, todo ello ya comentado.
  - EXP4: contiene los ficheros simulaciont.tcl y preparadatos, explicados anteriormente. La carpeta DATOS contiene 3 carpetas para los algoritmos RED, ARED y REM para las simulaciones con igual a 30. Su interior estará organizado de la siguiente forma:
    - Colax: carpeta que contiene los ficheros resultados de la simulación. La x corresponde al número de cola, en nuestro caso serán 4 colas.
    - img: carpeta con las gráficas de la simulación.

        - 4 ficheros de trazas correspondientes a cada cola congestionada.
    - EXP5: misma estructura que EXP1
- Artículos: en este directorio se encuentran los artículos empleados en los experimentos.

Anexo D

Técnicas

Lenguaje Unificado de Modelado (UML)

El lenguaje unificado de modelado (UML, Unified Modeling Language) es un lenguaje de modelado visual para sistemas. Aunque UML está más asociado con modelar sistemas de software orientado a objetos, tiene una aplicación mucho más amplia que esto debido a sus mecanismos incorporados de extensibilidad

UML es un estándar orientado a objetos que marca las pautas para modelar los diferentes aspectos del análisis y diseño de un proyecto software. Es un lenguaje que se utiliza para representar los modelos que se obtienen a partir de la aplicación de cualquier metodología orientada a objetos.

Se entiende por modelo una simplificación de la realidad, creada para comprender mejor el sistema que se está creando. Los diagramas UML son legibles por las personas y los ordenadores pueden mostrarlos fácilmente. Es importante apreciar que UML no nos proporciona ningún tipo de metodología de modelado (el proceso unificado si es una metodología que emplea UML como su sintaxis de modelo visual).

Para modelar, en UML existen tres bloques de construcción: elementos, relaciones y diagramas.

Elementos

Existen 4 tipos:

- Elementos estructurales: son las partes estáticas de un modelo y representan cosas conceptuales o materiales. Se distinguen siete elementos estructurales básicos, aunque existen variaciones de los mismos:

    - ***Clase*:** descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Comparten estructura y comportamiento. Una clase implementa una o más interfaces.

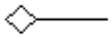
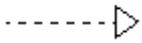
- ***Interfaz*:** colección de operaciones que especifican un servicio de una clase o componente. Define un conjunto de especificaciones de operaciones, pero nunca un conjunto de implementaciones.
- ***Colaboración*:** define una interacción, es una sociedad de roles y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Tiene una dimensión tanto estructural como de comportamiento.
- ***Caso de Uso*:** descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular. Se emplea para estructurar los aspectos del comportamiento en un modelo. Es realizado por una colaboración.
- ***Clase Activa*:** clase cuyos objetos tienen uno o más procesos o hilos de ejecución, y por tanto, pueden dar origen a actividades de control.
- ***Componente*:** parte física y reemplazable de un sistema que conforma con un conjunto de interfaces y proporciona la implementación de dicho conjunto.
- ***Nodo*:** elemento físico que existe en tiempo de ejecución y representa un recurso computacional; por lo general dispone de algo de memoria, y con frecuencia, de capacidad de procesamiento.

- Elementos de comportamiento: son las partes dinámicas de los modelos UML. Son los verbos de un modelo y representan comportamiento en el tiempo y el espacio. Son dos:
  - ***Interacción:*** comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos.
  - ***Máquina de estados:*** *comportamiento que especifica las secuencias de estados por las que pasa un objeto o una interacción durante su vida en respuesta a eventos, junto con sus reacciones a estos eventos*

- Elementos de agrupación: son las partes organizativas de los modelos UML. El paquete, que se utiliza para agrupar elementos de modelado semánticamente relacionados en unidades cohesivas.
- Elementos de anotación: La nota, que se pude anexar al modelo. Son las partes explicativas de los modelos UML.

Relaciones

Existen cuatro tipos de relaciones en UML: dependencia, asociación, generalización y realización.

Relación	Notación	Descripción
Dependencia	---------->	El elemento origen depende del elemento destino y se puede ver afectado por cambios en éste.
Asociación	————	La descripción de un conjunto de vínculos entre objetos
Agregación		El elemento destino es parte del elemento origen
Composición		Una forma de agregación fuerte (más restringida)
Generalización		El elemento origen es una especialización del elemento destino más general y se puede sustituir por éste
Realización		El elemento origen garantiza llevar a cabo el contrato especificado por el elemento destino

Diagramas

Un diagrama es la representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones).

UML incluye nueve diagramas:

- ***Diagrama de clases:*** representa gráficamente la vista estática, que muestra una colección de elementos declarativos del modelo como: clases, interfaces y colaboraciones, así como sus relaciones.
- ***Diagrama de objetos:*** muestra un conjunto de objetos y sus relaciones en un determinado instante de tiempo. Representan instantáneas de las instancias de los elementos existentes en los diagramas de clases.
- ***Diagrama de casos de uso:*** muestra las relaciones existentes entre actores y casos de uso dentro de un sistema.
- ***Diagrama de secuencia:*** muestra las interacciones entre objetos organizadas en una secuencia temporal. Resalta la ordenación temporal de los mensajes.
- ***Diagrama de colaboración:*** muestra las interacciones organizadas alrededor de los roles. Resalta la organización estructural de los objetos que envían y reciben mensajes.
- ***Diagrama de estados:*** muestra una máquina de estados que consta de estados simples, transiciones, estados compuestos anidados, eventos y actividades.

- ***Diagrama de actividades:*** es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema. Resaltan el flujo de control entre objetos.
- ***Diagrama de componentes:*** muestra la organización y las dependencias entre un conjunto de componentes.
- ***Diagrama de despliegue:*** muestra la configuración de los nodos de procesamiento en tiempo de ejecución y las instancias de los componentes y objetos que residen en ellos.

Programación Orientada a Objetos (OOP)

La programación orientada a objetos es una evolución de la programación estructurada que no sólo modifica la metodología de programación, sino que también supone un cambio en la metodología de diseño.

En la programación orientada a objetos se definen objetos que conforman una aplicación. Dichos objetos están formados por una serie de características y operaciones que se pueden realizar sobre los mismos. No se encuentran aislados en la aplicación, sino que se comunican entre ellos. Para entender mejor el concepto , definiremos objeto y clase.

Objeto: una entidad autónoma con una funcionalidad concreta y bien definida

Clase: especificación de las características de un conjunto de objetos. Un objeto es una instancia de una clase.

Un lenguaje orientado a objetos debe tener las siguientes propiedades:

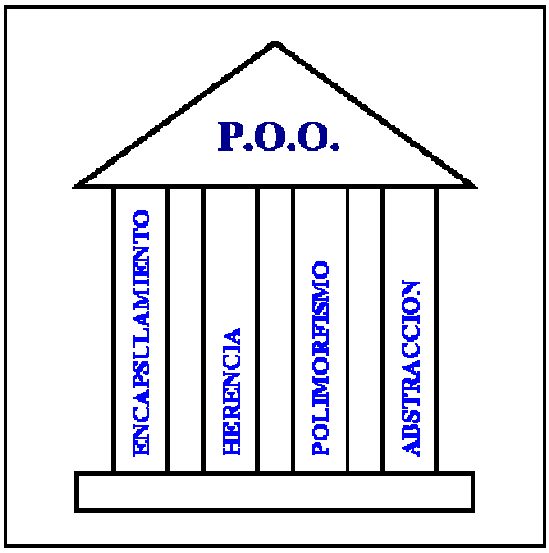


Figura 1. Pilares de la POO.

Encapsulamiento

El encapsulamiento es la propiedad que tienen los objetos de ocultar sus atributos y métodos a otras partes del programa u otros objetos. Dentro de un objeto se pueden definir atributos que sean accesibles desde fuera de esa clase o atributos a los que sólo se pueda acceder desde dentro de esa clase.

Herencia

La herencia permite definir clases descendientes de otras, de forma que la nueva clase herede de la clase antecesora todos sus atributos y métodos. Además puede definir nuevos atributos y métodos o incluso puede redefinir atributos y métodos ya existentes. Esta característica permite la reutilización de código.

Polimorfismo

Permite que un mismo mensaje enviado a objetos de clases distintas haga que éstos se comporten también de forma distinta (objetos distintos pueden tener métodos con el mismo nombre o incluso un mismo objeto puede tener nombres de métodos idénticos pero con distintos parámetros).

Abstracción

La abstracción consiste en captar las características esenciales de un objeto, así como su comportamiento. En los lenguajes de programación orientada a objetos, el concepto de Clase es la representación y el mecanismo por el cual se gestionan las abstracciones.

Simulación de Redes

La simulación es el proceso de diseñar un modelo de un sistema real y llevar a término experimentos con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias para su funcionamiento.

Debido al desarrollo de nuevas tecnologías, algoritmos y protocolos para Internet, la simulación de redes se convierte en una práctica habitual de modelado que permite comprobar y realimentar el proceso de diseño antes de proceder a una costosa implementación práctica.

Los simuladores de red pueden clasificarse en varios tipos atendiendo a distintos criterios (protocolo, tecnología, método de procesamiento…), pero la categorización más general es en base al método de simulación.

Existen dos métodos típicos de simulación:

- **Eventos discretos**: Producen predicciones en la red a bajo nivel (paquete por paquete), siendo precisos pero lentos al momento de generar los resultados.
- **Simulación analítica**: Utiliza modelos matemáticos para producir los resultados, siendo más rápidos que los discretos pero menos precisos.

El uso más habitual es combinar ambas metodologías para formar un simulador híbrido con el fin de proveer un desempeño aceptable en términos de velocidad, pero manteniendo la precisión en áreas críticas.