Autor	Gómez Martínez, Javier
E-mail del Autor	javigomezmartinez@hotmail.com
Director(es)	Pablo Pavón Mariño
E-mail del Director	Pablo.Pavon@upct.es
Título del PFC	Diseño e implementación de herramientas docentes de conmutación en el entorno Omnet

El objetivo fundamental de este proyecto es introducir al lector en la problemática de la evaluación de prestaciones de conmutadores de paquetes, desde un punto de vista didáctico, centrándose en los conmutadores con colas a la salida.

Primero se evaluarán individualmente y después en una red en tres etapas de conmutación de paquetes, donde estos conmutadores serán los nodos de interconexión.

La evaluación se desarrollará con la herramienta de simulación Omnet++.

Titulación	Ingeniería Técnica de Telecomunicaciones, especialidad en Telemática
Departamento	Departamento de Tecnologías de la Información y Comunicaciones
Fecha de Presentación	Diciembre- 2003

Índice

Capítulo 1
INTRODUCCIÓN

El buen rendimiento de las redes de conmutación de paquetes depende en gran medida del diseño de los conmutadores. El objetivo de este proyecto es iniciar el estudio de estos dispositivos desde un punto de vista didáctico.

Lo que se pretende es realizar un estudio de estos conmutadores para concienciar al lector de cual es su funcionamiento básico, de observar cual su arquitectura y percatarse de cómo afectan una serie de parámetros modificables en el rendimiento de la red.

Nosotros nos centraremos en los conmutadores con colas a la salida. Se estudiará cual es su estructura interna (cuales son los elementos que lo forman), explicar cómo realiza su trabajo, y se realizará una evaluación sobre como afecta a estos conmutadores la carga de entrada y el número de posiciones de memoria interna a la probabilidad de pérdida de paquete y al retardo medio de los paquetes, que son dos parámetros significativos a la hora de medir las prestaciones de estos dispositivos.

Debido a los propósitos docentes del proyecto, esta evaluación será realizada en forma de prácticas, de manera que sea el lector quien realice todas las pruebas, tomando conciencia de las principales características de estos dispositivos a partir de su experiencia.

Para terminar la evaluación de estos conmutadores, se realizará un estudio de una red de conmutación formada por la interconexión de estos dispositivos, y se comprobará ahora como afecta el uso de estos elementos de conmutación al rendimiento de la red. La red que se implementará será una red de conmutación en tres etapas con topología de interconexión de red de Clos.

Esta evaluación también será realizada en forma de prácticas, con el mismo objetivo que antes, que sea el lector el que realice todas las pruebas y estudie los resultados obtenidos de forma que sea él mismo el que tenga que sacar conclusiones y vea la importancia que tienen estos dispositivos a la hora de diseñar una red, y cómo afectan los parámetros modificables en el rendimiento al utilizar como elementos de interconexión estos dispositivos.

Estos estudios no se realizarán de forma exhaustiva. Nos limitaremos únicamente a realizar una introducción a los problemas principales que se presentan a la hora de estudiar estos dispositivos de conmutación.

El desarrollo de las prácticas que se realizan para la evaluación de los conmutadores implementados serán implementadas usando la herramienta de simulación *Omnet++* [5] [6]. Es una herramienta de propósito general pero utilizada sobre todo para diseñar redes y sistemas de telecomunicación. *Omnet++* podría tener una influencia importante en el campo de la docencia, pues el alumno podría desarrollar cualquier tipo de sistema y realizar su posterior evaluación utilizando dicha herramienta.

Además *Omnet*++ es una vía para adquirir conocimientos en el lenguaje C++ [11], en el que se basa la programación de simulaciones en *Omnet*++.

Capítulo 2
DESCRIPCIÓN DEL ENTORNO

2.1 Conmutadores de Paquetes

En torno a 1970 se ideó una nueva forma de arquitectura para comunicaciones de datos digitales de larga distancia: la conmutación de paquetes. Aunque la tecnología de esta técnica de conmutación ha evolucionado mucho desde entonces, se han de resaltar dos cosas: que la tecnología básica en conmutación de paquetes es esencialmente la misma en la actualidad que las redes de principios de los años 70, y que la conmutación de paquetes continúa siendo una de las pocas tecnologías efectivas para comunicaciones de datos a larga distancia.

La técnica de conmutación de paquetes se diseñó para ofrecer un servicio más eficiente que el proporcionado por la conmutación de circuitos. En la conmutación de paquetes, una estación realiza la transmisión de los datos en base a pequeños bloques llamados paquetes [1].

Cada paquete contiene la dirección de origen, la dirección de su destino, e información acerca de cómo volver a unirse con otros paquetes emparentados, entre otros campos.

Pues bien, los conmutadores de paquetes tienen la tarea de encaminar esos paquetes que le llegan a las líneas de entradas a cualquiera de sus líneas de salida, sabiendo que la información de encaminamiento utilizada para establecer caminos entre las entradas y salidas se encuentra en la cabecera de los paquetes [1] [9].

Decir que las arquitecturas en las que se centrará el estudio trabajan con paquetes de longitud fija y será así a lo largo de todo el proyecto.

Las partes fundamentales de un conmutador son el núcleo de conmutación y la memoria interna del conmutador. Dependiendo de la localización de la memoria se pueden tener distintos conmutadores. A continuación se examina el funcionamiento de diferentes estructuras de conmutación dependiendo de la localización de su memoria.

2.1.1 Conmutador con Colas a la Entrada

En estos se sitúa un *buffer* de memoria independiente para cada entrada del conmutador (Fig.2.1.).

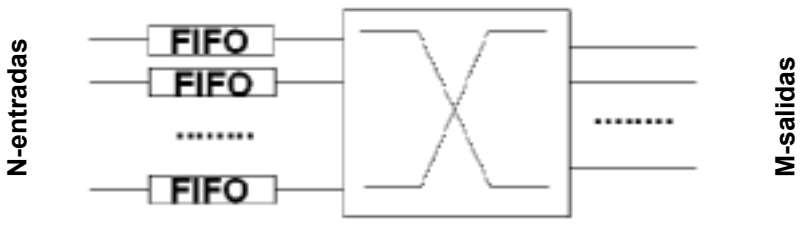

Fig. 2.1 Conmutador con colas a la entrada

Cada paquete que llega se sitúa en el *buffer* del puerto de entrada al que llegó y esperará su turno en la cola para atravesar el conmutador. Decir que los *buffers* siguen una disciplina FIFO y que sólo al comienzo del *slot* se lee la cabeza de la cola, pudiéndose así transmitir un paquete.

El problema surge cuando dos o más paquetes desean dirigirse al mismo puerto de salida, y es que en un mismo *slot* temporal, por un puerto de salida sólo puede transmitirse un paquete. La consecuencia de este suceso es que sólo un paquete se transmitiría quedando los restantes paquetes en cola. Estos paquetes esperarán al siguiente *slot* para "luchar" por su transmisión. El problema se agravaría mucho más si uno de esos paquetes que no se ha podido transmitir queda bloqueado, produciendo retardos importantes en los paquetes que están almacenados en esa cola. Esta situación tiene como consecuencia la obtención de caudales bajos.

Una solución por la que se opta para mejorar este problema es "relajando" la condición FIFO de las colas de entrada. Esto es que cada entrada continua enviando un paquete al conmutador por *slot* temporal, pero ahora no será necesariamente el paquete que ocupe la cabeza de la cola. En cada *slot*, el primer paquete de la cabecera de la cola compite con el segundo por su transmisión. Este proceso se repite sólo un determinado número de veces.

La ventaja que ofrece esta arquitectura es que las colas pueden realizar una lectura y una escritura en un *time slot* .

2.1.2 Conmutadores con Colas a la Salida

Este tipo de conmutadores, tal y como se muestra en la figura 2.2, tienen una cola por cada puerto de salida del conmutador. Dichas colas siguen, al igual que en un conmutador con colas a la entrada, una disciplina FIFO.

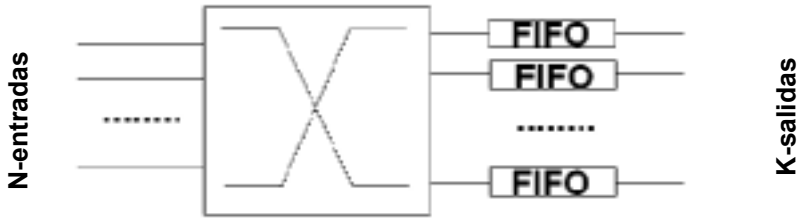

Fig. 2.2 Conmutador con colas a la salida

La ventaja que ofrece este tipo de conmutadores es que en un mismo *slot* temporal pueden procesarse todos los paquetes recibidos por los puertos de entrada y dirigirlos a cualquiera de las salidas, sin importar que varios paquetes deseen ir al mismo puerto de salida. Al llegar a los puertos de salida, los paquetes quedarán almacenados en las colas correspondientes. En cuanto a la transmisión de estos paquetes, sólo uno de ellos podrá ser transmitido a su enlace de salida en ese *slot* temporal (que será el paquete que ocupe la cabeza de la cola), quedando el resto almacenados hasta el siguiente *slot* temporal.

Como es de imaginar, con conmutadores con colas a la salida se consiguen tiempos de espera y caudales óptimos.

La desventaja que ofrece esta arquitectura es que las colas que emplea están diseñadas para poder realizar *N* escrituras + 1 lectura en un *time slot,* lo que hace que sean más caras, o incluso irrealizables para *slots* muy cortos.

Decir que en el capítulo 3 se volverán a explicar los conmutadores con colas a la salida de forma más detallada.

2.1.3 Conmutadores de Memoria Compartida

Esta arquitectura sigue siendo una estructura con colas a la salida. Tiene un *buffer* de memoria compartida para todos los puertos de salida, en lugar de tener una memoria individual para cada puerto (Fig.2.3).

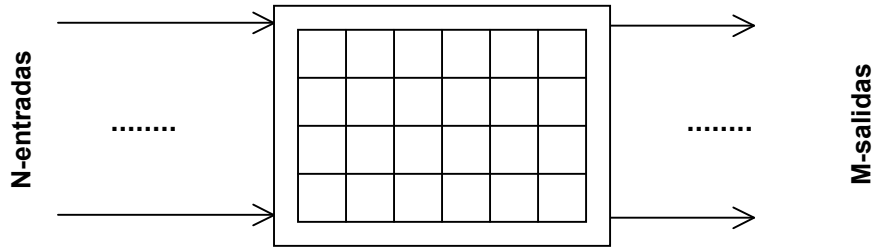

Fig.2.3 Conmutador con memoria compartida

A pesar de que se forma una cola separada para cada salida del conmutador, físicamente, todos los paquetes comparten el mismo espacio de memoria. Esta compartición supone una reducción de la cantidad total de memoria necesaria y una mayor eficiencia en la compartición de recursos de memoria. Con esto se consigue que el número total de *buffers* requerido es menor que en el caso de asignación de la memoria individualmente. A pesar de crecer los puertos de salida, la memoria no crece de igual manera la memoria requerida.

Además, como se trata de una arquitectura de conmutadores con colas a la salida, se siguen teniendo caudales y tiempos de espera óptimos.

El inconveniente es que la memoria puede ser adjudicada inconvenientemente. Si una salida está bastante cargada puede ocupar gran parte del *buffer* de memoria común, impidiéndose la entrada de paquetes con otros destinos. Además, presentan el mismo inconveniente que los conmutadores con colas a la salida: las colas internas pueden realizar hasta *N* lecturas + 1 escritura, lo que hace que sean más caras y a veces irrealizables, cuando el *slot* es muy corto.

2.2 Herramientas de Simulación de Conmutadores

A la hora de implementar un conmutador, antes de realizarlo físicamente, primero se le evalúa. Para realizar esta evaluación la técnica más utilizada es la simulación por software [10]. A la hora de elegir el software debe tenerse en cuenta que este software debe ser capaz de reflejar fielmente el diseño que se quiere implementar. Debe de proporcionar el equilibrio necesario entre la funcionalidad del sistema implementado y al mismo tiempo proporcionar flexibilidad necesaria para configurar una gran cantidad de parámetros de simulación por parte del usuario. Además debe de ser capaz de mostrar resultados concretos y correctos ante esos parámetros que se le introducen al empezar la simulación.

Estas herramientas de simulación se pueden clasificar en herramientas de simulación de propósito general o de propósito especifico. A continuación se detallarán

algunas de estas herramientas, teniendo en cuenta de que serán empleadas para la docencia.

2.2.1 Herramientas de Propósito General

Este tipo de herramientas son aquellas que proporcionan un entorno en el cual poder desarrollar cualquier tipo de implementación. La ventaja suele ser que este tipo de herramientas son de libre distribución. Algunas de estas herramientas son:

2.2.1.1 Parsec

PÁRSEC [3] es un lenguaje de simulación muy eficiente, con un soporte bastante fuerte en simulaciones paralelas. PÁRSEC es utilizado en numerosos proyectos de simulación sobre todo en aplicaciones de *mobile radio networks* en entornos militares.

PARSEC (*for PARallel Simulation Environment for Complex programs*) es un lenguaje de simulación discreto basado en eventos utilizando el lenguaje C. Un objeto (también referido a un proceso físico) o un grupo de objetos en el sistema real, son representados por un proceso lógico (un *thread*). Las interacciones entre los procesos físicos (eventos) son modelados por el intercambio de mensajes entre los correspondientes procesos lógicos.

Una de las características importantes que distingue a PÁRSEC es la capacidad de ejecutar un modelo de simulación discreto basado en eventos usando diferentes protocolos de simulación asíncronos en una variedad de arquitecturas en paralelo.

El lenguaje PÁRSEC es derivado a partir del lenguaje *Maisie* pero con dos diferencias importantes: la sintaxis del lenguaje y el medio de ejecución donde se desarrollan las simulaciones.

Uno de los aspectos importantes a tener en cuenta a la hora de trabajar con PÁRSEC es que se puede utilizar en cualquier plataforma: se puede usar bajo Linux, Windows, Macintosh....

2.2.1.2 Maisie

Maisie [4] es un lenguaje de simulación discreto basado en eventos que utiliza el lenguaje C. Un objeto (también denominado por PP debido a las palabras *physical process*) o un grupo de objetos de la simulación son representados por procesos lógicos o LP (por *logical process*). La interacción entre PPs (eventos) son modelados por mensajes en tiempo de ejecución intercambiados entre los distintos LPs.

La característica más importante es la misma que la de la herramienta PÁRSEC descrita anteriormente, ya que como se ha dicho PARSEC es una derivación de Maisie. Esta característica es la capacidad de ejecutar un modelo de simulación discreto basado en eventos usando diferentes protocolos de simulación asíncronos en una variedad de arquitecturas en paralelo.

Un programa en Maisie es una colección de funciones de C y definiciones de entidades. Una entidad describe una clase de un objeto. Después de tener la entidad creada se puede crear una instancia de ella para crear el modelado de los objetos del sistema físico. Por ejemplo, se puede crear una instancia de tipo *server* que puede ser definida para modelar objetos servidores. Si después se crea una instancia especifica de esta entidad para modelar objetos servidores del sistema real.

2.2.1.3 Mirsim

Esta herramienta [2] es un simulador de procesos en paralelo. Ayuda a programas, por ejemplo creados en Maisie, que constan de numerosos procesos en paralelo a poder realizarlos de forma eficiente y rápida.

Para las ejecuciones primero se debe dividirse el modelo en una serie de particiones, las cuales potencialmente se podrán ejecutar en un procesador distinto. El objetivo del particionamiento es generar particiones balanceadas de forma que el número de particiones sea el mínimo así se minimiza el número de mensajes intercambiados entre las distintas partes (hay que recordar que este tipo de programas se basan en la simulación por paso de mensajes, que serán los eventos).

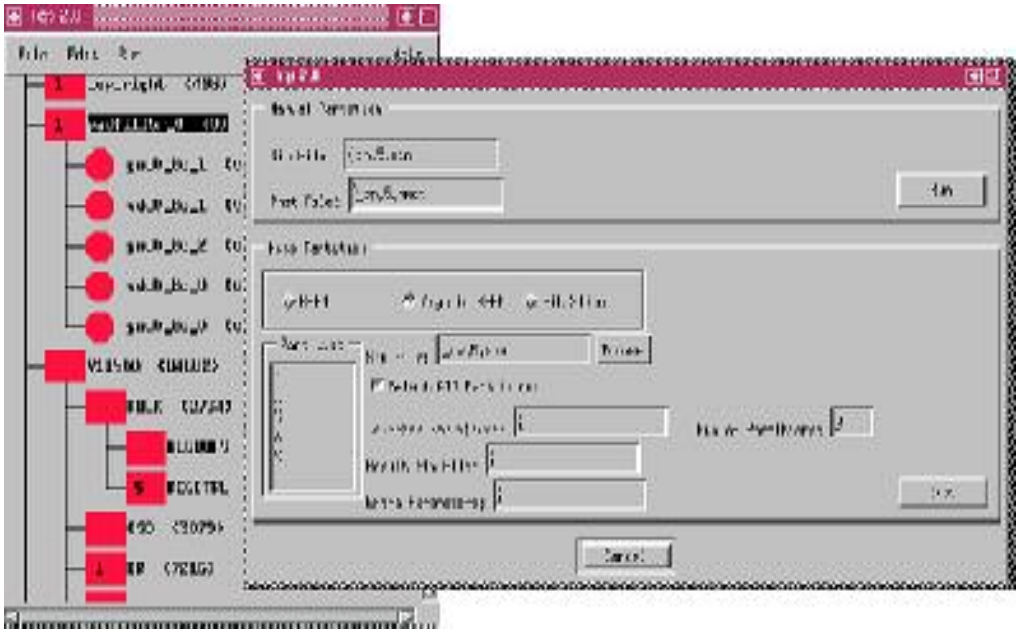

Fig.2.4 Entorno gráfico del MIRSIM

Decir por último que MIRSIM, para simular los eventos en orden provee los siguientes algoritmos de sincronización:

- *Global Event List algorithms*, o secuenciales
- *Accelerated null message protocol* (ANP), basado en tres algoritmos en paralelo, que usan respectivamente, *null messages*, eventos condicionales y una mezcla de los dos anteriores
- *Parallel optimistic algorithm*, basado en el espacio-tiempo de las simulaciones
- *Ideal Simulation Protocol* (or ISP)

2.2.1.4 OpNet

OpNet [5] [12] es un programa de simulación comercial que se utiliza para el modelado de sistemas de comunicación. Esta herramienta está diseñada para que se pueda implementar cualquier tipo de sistemas, posee herramientas que ayudan a implementar cualquier tipo de protocolo (aunque no sea estándar) o comportamiento.

OpNet presenta como características principales:

- Una avanzada interfaz de usuario que soporta multi-ventana, ofrece iconos y menús desplegables y trabaja bajo *X-Windows*. Además puede

trabajar bajo estaciones de trabajo muy extendidas como Sun, DEC, HP y Silicon Graphics. (Existe una versión para trabajar bajo Windows NT).

- Editor gráfico orientado a objetos que permite definir topologías y arquitecturas totalmente paralelas a los sistemas reales, permitiendo un mapeado intuitivo entre el modelo de la simulación y el sistema. Además, la jerarquía de OpNet permite simplificar la representación de modelos extensos.
- El editor de procesos provee un potente y flexible lenguaje denominado *Proto-C*, que combina diagramas de estados de transición y la potencia del lenguaje C.
- OpNet permite representar los resultados obtenido utilizando gráficas tales como funciones de distribución, histogramas, etc. Estadísticas estándar e intervalos de confianza son fácilmente creados con esta herramienta.
- También provee una avanzada capacidad de visualización de los eventos simulados, de forma que estos se interpreten de forma intuitiva.
- Por último, OpNet también puede ser utilizado para utilizar otras herramientas de otros sistemas, como puede ser un entorno gráfico, utilizar librerías de terceras partes, acceder a bases de datos, etc.

2.2.1.5 Omnet++

Esta es la herramienta [5] [6] sobre la que se centra el proyecto. Con ella se desarrollarán y estudiarán las distintas arquitecturas que se quieren implementar. En este apartado se hablará de la herramienta superficialmente porque en la sección 2.3 se describirán detalladamente todas sus utilidades.

Omnet++ es una herramienta de libre distribución. Está basada en eventos discretos, orientada a objetos y usa la potencia del lenguaje C++ [11] (de ahí las siglas ++).

El simulador se puede utilizar para describir:

- Protocolos de comunicación
- Redes de ordenadores y modelos de tráfico
- Multi-procesos y sistemas distribuidos
- Sistemas administrativos
- ... y cualquier tipo de sistema que esté basado en eventos

Un modelo en Omnet se basa en una jerarquía de módulos anidados. El nivel de anidamiento es ilimitado lo que permite a usuario reflejar fielmente el sistema mediante el modelo creado.

Los módulos se comunican entre ellos mediante el paso de mensajes, y estos mensajes pueden contener estructuras tan complejas como se deseen. Los mensajes se pueden transmitir entre los módulos directamente o a través de una ruta preestablecida, atravesando puertas y enlaces. Además, los mensajes pueden ser utilizados con tres propósitos fundamentales: modelar el comportamiento, crear topologías flexibles o modelar comunicaciones, como variables almacenadas.

Los módulos que están en el nivel más bajo de la jerarquía los diseña el programador y contienen los algoritmos del diseño. Estos módulos se denominan módulos simples (*simple modules*), y están programados en C++ [11]. Durante la ejecución de la simulación, estos módulos simples se ejecutan en paralelo.

Cuando una simulación termina se pueden obtener estadísticas tales como histogramas, funciones de probabilidad, cotas máximas, etc. Además Omnet viene acompañado de una herramienta que ayudará a representar estas estadísticas, denominada Plove [5].

Por último decir que, Omnet puede ser ejecutado en numerosas plataformas: Unix, DOS, Windows... junto a un compilador C++, lo que hace que esté muy extendido.

2.2.2 Herramientas de Propósito Específico

Son aquellas diseñadas para programar un sistema concreto. Son mucho más especializadas. A continuación se presentará una herramienta de libre distribución diseñada para simular redes ATM [8] denominada SIGLA.

2.2.2.1 Simulador Global para Redes ATM (SIGLA)

El problema del diseño

El modo de transferencia asíncrono (MTA) se ha propuesto como el principio de conmutación, multiplexación y transmisión de las futuras redes de servicios integrados de banda ancha. Dichas redes están siendo objeto de un amplio estudio e investigación. El hecho de que el tiempo de propagación sea comparable e incluso superior al tiempo de transmisión de los paquetes implica nuevos retos en su diseño. En particular, se puede afirmar que los métodos utilizados para diseñar y dimensionar las redes de telecomunicación convencionales han dejado de ser válidos.

Una forma común de diseñar los nodos MTA es por medio de redes de interconexión que responden a distintas arquitecturas (redes multietapa, memoria compartida...). En la red de comunicación, tanto los nodos de conmutación como los nodos de acceso estarán unidos de acuerdo a una topología determinada. En general la ubicación de estos dispositivos vendrá fijada previamente por otros condicionantes (población, etc.) y sólo será necesario explorar las diversas formas de interconexión posibles. En la conmutación rápida de paquetes, una de las principales cuestiones es la disposición de zonas de almacenamiento de dichos paquetes en la entrada, en la salida o de forma compartida entre la entrada y la salida. Los mecanismos de control que prevengan la utilización abusiva de estas zonas de almacenamiento y por supuesto la pérdida de posibles celdas es uno de los objetivos principales del diseño de estos dispositivos de conmutación.

Las figuras de mérito que normalmente se evalúan para estos dispositivos son, entre otras, el caudal o flujo de celdas, retraso medio debido al proceso de conmutación, probabilidad de pérdida de celdas, etc. No obstante, en una red existen usuarios que solicitan los servicios propios ofrecidos por ésta, con lo que será necesario establecer otros criterios de prestaciones de la red en base a las medidas (simulaciones) obtenidas para cada uno de los elementos que configuran la red bajo estudio. Entre éstas podemos incluir: el retraso medio extremo-extremo y su variación (*jitter*), caudal estimado (una vez que se ha considerado la capacidad de los enlaces), pérdidas, control de tráfico, puntos de congestión, tipos de encaminamiento, etc..

El diseñador de estos sistemas podría estar más convencido de su diseño si además fuera capaz de obtener resultados concretos sobre el comportamiento de la red al introducir uno de los servicios especificados, por ejemplo: servicios multimedia, viendo cómo se puede variar su calidad con la modificación de algunos de los parámetros previamente definidos en la red. También se podrían incorporar otros para establecer de forma inequívoca a qué parámetros físicos de diseño de la red es más sensible la calidad del servicio ofrecido y proponer las medidas oportunas para su no degradación.

El proyecto SIGLA

Como objetivo general, se quiere desarrollar un simulador global para redes MTA [7] utilizando programación orientada a objeto y con una ejecución distribuida en varios sistemas mono y multiprocesadores que estén conectados por medio de una red local. Se pretenderá que el interfaz de usuario sea lo más amigable posible existiendo versiones modulares simplificadas del programa global que funcionen en otras plataformas, como por ejemplo en PC.

El simulador está organizado en tres niveles, por lo que el estudio, planificación y dimensionado de la red se puede efectuar en cada uno de ellos reutilizando los resultados obtenidos en cada nivel en el inmediato superior. Estos tres niveles se han denominado: nivel de dispositivo, nivel de red y nivel de usuario o servicio.

En el nivel de dispositivo se desarrollan los módulos de cada uno de los dispositivos que intervienen en una red MTA como son: nodos de conmutación y acceso, dispositivos generadores/colectores de tráfico y dispositivos controladores y monitorizadores de tráfico. Los resultados obtenidos en este primer nivel podrán ser utilizados en el nivel de red para establecer el dimensionado y planificación de la misma mediante la generación de topologías y el establecimiento de las técnicas de encaminamiento y control de congestión apropiadas. Por último, en el tercer nivel se evalúan los diferentes servicios que integran las redes MTA a través de la cuantificación de diferentes parámetros de calidad y mediante el sistema perceptual humano, en el caso de los servicios que incorporen información de vídeo o voz.

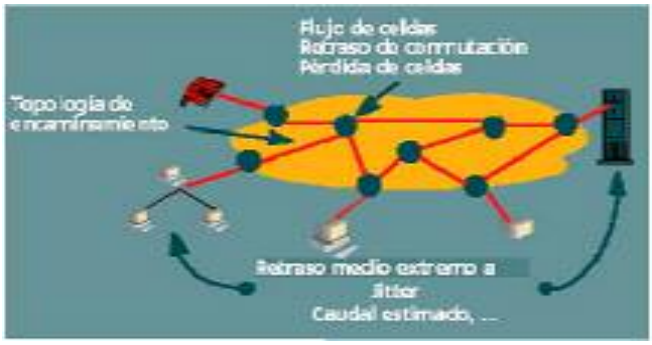

Fig.2.5 Infraestructura Red de Conmutación

Este objetivo general pretende cubrir varias actividades. Entre éstas, y como más importantes, se pueden destacar:

- Planificación de infraestructuras, capaces de ofrecer servicios telemáticos (comunicación multimedia interactiva y no interactiva) y determinar el grado de servicio de los mismos, lo que podría ser de utilidad como indicativo para el establecimiento de la futura tarificación asociada a estos servicios.
- Desarrollo y experimentación de nuevas plataformas de trabajo para las tecnologías emergentes que permitan el estudio de nuevas arquitecturas y el análisis y evaluación de diferentes técnicas y prestaciones de las mismas.
- Apoyo a la docencia mediante la realización de versiones simplificadas del programa general que podrán ser difundidas por las redes existentes para su utilización en puntos distantes. Cuando su uso así lo exija, se podrán establecer los mecanismos de consulta y resolución de

aclaraciones, bien mediante la utilización de un servidor para tal fin o la realización de seminarios y cursos de interés específico en este área.

2.3 Descripcion del Entorno Omnet

A continuación se hará un estudio detallado de todo el entorno Omnet [5], diciendo qué es, como y para qué se utilizada, etc.

2.3.1 Introducción

2.3.1.1 Qué es Omnet

E una herramienta de simulación orientada a objetos, que utiliza el lenguaje C++ [11]. Usado para modelar:

- Protocolos de comunicación
- Redes de ordenadores y modelado de tráfico
- Multiprocesadores y sistemas distribuidos
- .....

El modelado en Omnet se basa en módulos jerárquicamente anidados. El anidamientos es ilimitado y los módulos se comunican por el intercambio de mensajes. Estos mensajes se pueden mandar directamente o a través de *gateway* y conexiones. Los módulos pueden tener parámetros usados con tres fines distintos: modelar el comportamiento del sistema, crear flexibles topologías o modelar la comunicación.

Los módulos de más bajo nivel en la jerarquía son proveídos por el usuario y contienen los algoritmos del modelo, escritos en C++.

2.3.1.2 Conceptos de Modelado

Omnet proporciona al usuario eficientes herramientas para la descripción de la estructura del sistema analizado

- Modelos jerárquicamente anidados
- Los módulos son instancias de módulos simples
- La comunicación de los módulos se realiza a través de canales
- La introducción de los parámetros de los módulos es flexible
- Tiene un lenguaje de descripción de la topología propio

Jerarquía de los Módulos

Los modelos construidos en Omnet, casi siempre, son referidos a redes. Para diseñar estos modelos, como se ha mencionado antes, se realiza de forma jerárquica, anidando entre sí los módulos, de forma que el nivel más alto es el denominado *system* y a partir de este se van encontrado módulos compuestos de otros módulos (denominados submódulos), y así hasta el nivel más bajo, formado por los módulos simples (Fig. 2.6). Estos módulos simples se implementan en C++ [11], usando la librería de Omnet *simulation class library.*

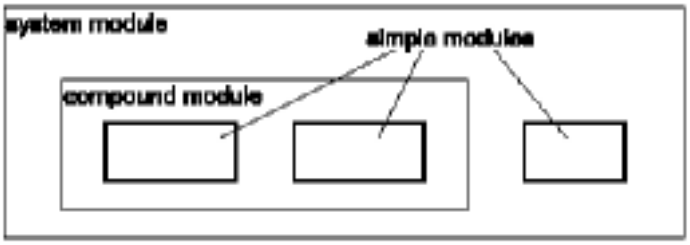

Fig.2.6 *Simple* y *Compound Modules*

Tipos de Módulos

Los módulos simples y compuestos son instancias de *module types.* Mientras se describe el modelo el usuario define el tipo de módulo; instancia de ese *module type*, que sirve como componente para un módulo más complejo. Finalmente, el usuario crea un *system module* como una instancia del tipo de módulo anteriormente creado.

Cuando se una un tipo de módulo para construir un bloque no hay distinción si es simple o compuesto. Esto permite dividir un módulo simple en más módulos simples insertados en un módulo compuesto, o viceversa, agregas la funcionalidad de un módulo compuesto a uno simple, sin afectar nada.

Mensajes, *Links* y *Gates*

Los mensajes se representan en tramas o paquetes en una red, trabajos o clientes en una cola u otros tipos de una entidad móvil. Los mensajes pueden contener cualquier tipo de estructura.

Los *gates* son las entradas o salidas de las interfaces de los módulos. Los mensajes se mandan por las *output gates* y se reciben por las *input gates.*

Cada conexión (también llamada *link*) se crea dentro de un mismo nivel de jerarquía: dentro de un módulo compuesto, uno puede conectar entre sí dos *gates* de dos módulos simples, o conectar un *gate* de un módulo simple con un *gate* de un módulo compuesto. (Fig.2.7)

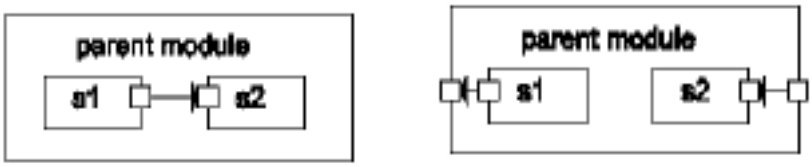

Fig.2.7 Conexiones posibles en un nivel de jerarquía

Debido a la jerarquía del modelo, los mensajes comienzan en un módulo simple y terminan en un módulo simple, atravesando por su camino distintas conexiones.

Características de los enlaces

Las conexiones pueden ser modeladas con tres parámetros interesantes que pueden ser útiles a la hora de diseñar los sistemas:

- Retardo de propagación
- BER (*bit error rate*)
- *Data rate*

Cada uno es opcional. Uno puede especificar los parámetros de los enlace individualmente o definir un modelo de conexiones una vez y utilizarlo cada vez que se necesite.

El retardo de propagación es el tiempo que tarda en llegar el mensaje al destino en relación a lo que debería tardar en función del canal atravesado.

El BER tiene influencia en la transmisión de mensajes a través del canal. Es la probabilidad de que un *bit* sea transmitido incorrectamente. P(n)= (1-ber)^n, donde P(n) mide la probabilidad de que un mensaje sea transmitido correctamente y n es el número de *bits* que contiene el mensaje.

El paquete tiene un *flag* que activa cuando se transmite el mensaje con errores.

El *data rate* es utilizado para el cálculo del retardo de transmisión.

Parámetros

En Omnet los parámetros se usan con tres labores fundamentalmente: parametrizar la topología del modelo, modelar el comportamiento de un módulo simple o para la comunicación entre módulos.

Los parámetros pueden ser *strings,* números, punteros, valores numéricos incluyendo expresiones o llamadas a funciones en C, variables aleatorias para ciertas distribuciones y valores de entrada para los usuarios.

Los valores numéricos son usados para construir topologías de forma flexible. Dentro de un módulo compuesto, los parámetros pueden definir el número de submódulos, el número de conexiones y el camino interno de las conexiones.

Los módulos compuestos pasan parámetros a los submódulos (por valor o referencia) durante la simulación. Si un puntero cambia el valor de un parámetro pasado por referencia, este cambio se propaga al resto de módulos.

Método de descripción de la topología

Mediante el lenguaje de descripción NED.

2.3.1.3 Programación de los Algoritmos

Los módulos simples de un modelo contienen los algoritmos como funciones de C++.

Omnet++ soporta un método de descripción del estilo de proceso para describir las actividades. Durante la ejecución, los módulos se arrancan por separado (en paralelo). Esta técnica se diseño por la facilidad que da para implementar después otros métodos.

Omnet++ tiene un diseño que consiste en la programación orientada a objetos. Se puede usar todas las herramientas que esto ofrece: herencia, polimorfismo... para extender la funcionalidad del simulador.

Los elementos de la simulación son representados como objetos. Esas clases son parte del simulador en la librería de clases:

- Módulos, *gates, conexiones...*
- Parámetros
- Mensajes
- Contenedores de clases (*array*, colas)
- Clases de recolección de datos
- Clases para estimaciones de distribución y estadísticas
- ...

Creación de Módulos Simples

Cada módulo simple es implementado en C++. Es derivado de la clase base de módulos simples redefiniendo los métodos virtuales que contiene. El usuario puede añadir otros métodos a la clase para simplificar un algoritmo complejo.

Es posible derivar también nuevos módulos simples a partir de otros existentes. Por ejemplo en un protocolo de transporte se quiere experimentar con un esquema de retransmisión y entonces derivar una familia de clases que implemente esquemas concretos.

Derivar nuevas clases

En la mayoría de casos, con la funcionalidad que ofrece Omnet++ es suficiente, pero se es necesario, se pueden derivar nuevas clases de las ya existentes, o crear clases completamente nuevas. Por la flexibilidad, muchas de las funciones son declaradas virtuales. Cuando creamos clases nuevas son necesarias ciertas reglas para guardarlo y que luego pueda actuar junto a otros ya existentes.

2.3.1.4 Utilización de Omnet++

***Building & run* de las simulaciones**

Un modelo en Omnet++ consiste en la siguientes partes:

- Archivos .NED. Estos son proporcionados por el lenguaje NED de descripción de la topología. Este describe la estructura del módulo con los parámetros, conexiones...
- Código fuente de los módulos simples. Son archivos C++ (.h / .cc)

El sistema de simulación nos da los siguientes componenete:

- Kernel que contiene el código que dirige la simulación
- Interfaces de usuario, usadas para la ejecución de la simulación, y que facilitan la ejecución de la simulación.

Los programas con construidos con las siguientes partes: primero el archivo .NED es compilado dentro de un código fuente C++ [11], usando el NEDC *compiler* (es parte de Omnet++). Después todas las fuentes son compiladas y juntadas con la simulación del Kernel y el interfaz de usuario para formar una simulación ejecutable.

El programa ejecutable es un único programa, así que puede ser arrancado en distintas máquinas con Omnet++. Cuando arranca se lee un fichero de configuración (omnetpp.ini), que contiene secciones para controlas muchas características de la simulación, así como ajustar los parámetros de los módulos o del modelo.

Cuando termina la simulación, si se han utilizado objetos de Omnet++ que sirven para realizar estadísticas, tales como *output vector,* quedará información grabada en estos objetos, que posteriormente podrá ser interpretada con una GUI que provee

Omnet++ llamada Plove. Esta herramienta sirve para dibujar gráficas de los contenidos del *output vector file.* Decir por último que estos archivos *output vector* son archivos de texto que pueden ser procesados con programas como Matlab, Excel, awk (para realizar filtros)...

2.3.2 Lenguaje NED

2.3.2.1 Introducción

Este lenguaje ofrece el poder describir la topología de cualquier sistema. Soporta una descripción modular de la red. Esto es que la red consiste en una serie de componentes descritas (canales y módulos simples o compuestos). Los módulos simples y compuestos y los canales de una red pueden ser empleados en otras descripciones de otras redes. Esto establece la posibilidad de tener una librería con nuestras propias redes.

Los archivos que contienen las descripciones de estas redes son archivos .ned. No son usados directamente. Primero se compilan con el compilador NEDC, transformando el archivo en uno con código C++. Este después se compila con todos los restantes archivos C++ (que contienen la descripción de los módulos simples) y se forma el ejecutable.

Componentes de una descripción NED

- Importar archivos
- Definición de los canales
- Declaración de los módulos (simples o compuestos)
- Declaración de *system module.*

2.3.2.2 Importar Archivos. Cláusula i*mport*

Esta declaración es usada para importar otros archivos de descripción (*.ned*). Después de importarlos se pueden usar como si se hubiesen creado en el mismo *.ned* que estamos diseñando. Un ejemplo de esta sentencia es la siguiente: *import "tkn_module".*

Lo que se importa es sólo la descripción del módulo, pero no se importa nada de código C++.

El usuario también puede especificar el nombre de archivos con o sin red *.ned,* es decir, importar únicamente módulos.

Cuando importamos lo que se tiene que hacer es volver a compilar y linkar todas las redes descritas, no sólo las de más alto nivel. A la hora de compilar el usuario puede indicar en la línea de comandos donde se encuentra el directorio del cual se importa el archivo.

2.3.2.3 Definición de los Canales

Especifica el tipo de conexión que se desea realizar, a partir de las características reales del sistema. El nombre del canal puede ser usado después en la descripción NED para crear conexiones con esas características. Un ejemplo es el siguiente:

```
channel DialUpConnection
delay normal (0.004, 0.0018)
error 0.00001
datarate 14400
endchannel
```

Cualquiera de los parámetros son opcionales y pueden aparecer en cualquier orden. Los valores son expresiones NED: constantes, valores aleatorios de una distribución dada, etc.

2.3.2.4 Definición de Módulos Simples

Son los bloques básicos que utilizan otros módulos. Son declarados indicándoles los parámetros y las puertas de las que consta, por ejemplo:

```
simple SomeNameForModule
parameters:
//...
gates:
//...
endsimple
```

Los parámetros de los módulos simples son variables que pertenecen a un módulo. Pueden ser requeridos y usados por algoritmos de módulos simples, por ejemplo: *num_of_msg* que puede ser requerido por un módulo llamado mensaje para contabilizar cuantos mensajes han sido generados.

Los parámetros son declarados listando sus nombres en la sección ***parameters:***

```
parameters:
interarrival_time,
num_of_messages : const,
address : string;
```

Si los tipos son omitidos, se asumen numéricos.

Cuando el usuario escribe *const* después del parámetro, este es convertido a constante, que es reemplazado para su evaluación. Esto es importante cuando el valor es un número aleatorio o una expresión. Hay que llevar cuidado al pasar una constante por referencia, puesto que cambiar el valor de esa variable por referencia cambiaría el estado de todos los módulos que la utilizasen.

Convertir el parámetro a constante puede afectar al resto de módulos y causar errores difíciles de encontrar.

Las puertas son puntos de conexión del módulo con otros módulos.

Hay dos clases de puertas: *input* y *output.* Los mensajes llegan al módulo por las puertas *input* y salen del módulo por las *output.*

Las puertas son identificadas por sus nombres. Además pueden ser vectores, conteniendo cada posición del vector una de las salidas del módulo. El tamaño de estos vectores se indica cuando se diseña la puerta del módulo o después, cuando se realiza el módulo compuesto y se obtiene el número de puertas que debe tener.

Por último, las puertas se declaran listando los nombres en la sección de ***gates.*** Un corchete vacío denota una puerta como vector. Además, al crear una puerta se inicializa a cero.

Un ejemplo de declaración de puertas:

```
gates:
in: output[];
out: input[];
```

2.3.2.5 Definición de Módulos Compuestos

Son módulos compuestos por otros módulos. Al igual que los módulos simples, pueden tener parámetros y puertas, por lo que tienen una definición similar a la de un módulo simple, excepto que también tienen secciones para especificar los submódulos y conexiones dentro del módulo. Los submódulos pueden ser a su vez simples o compuestos. Un ejemplo de declaración es el siguiente:

```
module SomeNameForCompoundModule
parameters:
//...
gates:
//...
submodules:
//...
connections:
//...
endmodule
```

Todos los campos son opcionales

Los parámetros son declarados de la misma forma que en un módulo simple (ver los ejemplos anteriores).

Pueden ser usados de dos formas:

- En expresiones para pasarles valores a los parámetros de los submódulos
- Usados en la definición interna de la topología de la red.

Las puertas vuelven a tener la misma definición que en un módulo simple. Para ello mirar los ejemplos mencionados en la parte de módulos simples donde se habla de las puertas.

Para cada submódulo, hay secciones para definir el valor actual de sus parámetros y el tamaño de sus puertas a partir de los valores de los parámetros y puertas, respectivamente, del módulo compuesto. Por ejemplo:

```
module NameForCompoundModule
parameters: //...
gates: //...
submodules:
SubModuleName: TypeOfSubModule
parameters:
//...
gatesizes:
//...
SecondSubModuleName: TypeOfSecondSubModule
//...
connections: //...
endmodule
```

En la definición de un submódulo, el usuario tiene que suministrar el nombre de un módulo previamente definido, así como el nombre y tipo del módulo. Esta descripción puede hacerse en el mismo archivo NED o en un archivo NED importado.

Es posible crear un vector de submódulos. Esto se consigue con expresiones entre paréntesis.

Los parámetros de los submódulos se pasan de la siguiente manera:

```
module ManyParameters
parameters:
par1, par2, switch;
submodules:
Submod1: Node
parameters:
p1 = 10,
p2 = par1+par2,
p3 = switch==0 ? par1 : par2;
//...
endmodule
```

Al igual que en los módulos simples, los parámetros pueden ser pasados por valor o por referencia con la clausula *ref ancestro.* Ejemplo:

```
parameters:
s_s_par1 = ref s_par,
```

El tamaño de las puertas de los submódulos son determinadas con la palabra reservada *gatesizes.* El valor puede ser una constante, una expresión o un parámetro. Ej:

```
Submod1: SimpleType
gatesizes:
inputs[10], outputs[num];
```

2.3.2.6 Conexiones

En la definición de un módulo compuesto, hay que decir que las puertas del módulo compuesto están conectadas a sus submódulos. El lenguaje no soporta dos módulos puedan comunicarse si no es a través de *gates.* Es por esto que la única conexión que soporta el lenguaje es punto-punto.

Las conexiones se especifican en el apartado de *conections:*

```
module SomeCompound:
parameters: //...
gates: //...
submodules: //...
connections:
node1.output --> node2.input;
node1.input <-- node2.output;
//...
endmodule
```

Cada conexión puede ser directa, conexión simple o múltiple, conexión condicional o incondicional.

Para realizar conexiones simples lo único que hay que hacer es indicar la puerta *input* de que módulo se conecta con la puerta *output* de que módulo. La sentencia se realiza de la siguiente manera: *`Sender.outgate --> Receiver.ingate.`* El usuario también puede llamar a la conexión por el nombre que le haya dada (si tiene alguna conexión definida con ciertos parámetros), o pasarle directamente los parámetros.

Si usamos un submódulo o un vector de puertas es cuando se pueden usar conexiones múltiples. Estas conexiones se suelen realizar de la siguiente manera:

```
for i=0..4 do
Sender.outgate[i] --> Receiver[i].ingate
endfor
```

El resultado de esta operación se muestra en la figura 2.8. Se pueden colocar muchas conexiones en el cuerpo del *for* separadas por *;*. Además, se puede utilizar más de un índice para referirnos a una conexión, teniendo como resultado un *for* anidado. Por ejemplo:

```
for i=0..3, j=i+1..4 do
//...
endfor
```

dando como resultado la siguiente secuencia:

(0,1) (0,2) (0,3) (0,4) (1,2) (1,3) (1,4) (2,3) (2,4) (3,4)

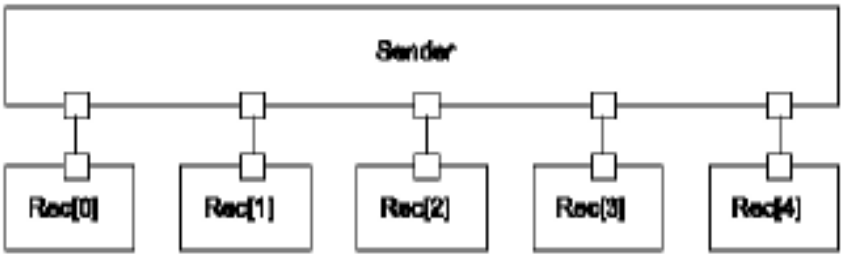

Fig.2.8 Representación Gráfica de una conexión múltiple

Otra característica que ofrecen la conexiones es que se pueden tener conexiones condicionales, poniendo dentro del for una sentencia como la siguiente:

```
Sender.outgate[i] --> Receiver.ingate[I] if I%2==0
```

2.3.2.7 Definición de una Red

La definición de la red es, por así decirlo, darle un nombre al *module simple*. La sintaxis es muy similar a la de un submódulo. Se comienza con la palabra reservada *network* y se termina con *endnetwork*. Por ejemplo:

```
network modelledNetwork: SomeModule
parameters:
par1=10,
par2=normal(100,20);
endnetwork
```

Aquí *SomeModule* es el nombre del módulo compuesto o simple.

Pueden haber múltiples definiciones de sistemas en la descripción de la red, cada una define una red diferente.

El programa construye una simulación a partir de una de red.

2.3.2.8 Expresiones

En el lenguaje NED hay una serie de sitios donde las expresiones están supuestas.

Cuando el compilador encuentra una expresión la compila y será evaluada en tiempo de ejecución.

Son similares a las que se encuentran en el lenguaje C. Están construidas con operadores matemáticos usuales, pueden utilizar parámetros, llamar a funciones...

Uso de parámetros en las expresiones

Las expresiones pueden usar los parámetros de los módulos diseñados. Pueden pasarse por valor o por referencia. Por defecto se coge por valor. Para pasar un valor

por referencia hay que utilizar el modificador *ref.* Normalmente, un parámetro se pasa por referencia cuando se desea que al cambiar el valor de la variable, este cambio se propague por toda la red. Un ejemplo de esto es el siguiente:

```
parameters:
par1 = ref nnn / 2,
```

Expresión *sizeof()* y el operador *index*

La función *sizeof()* lo que nos devuelve es la longitud de un vector.

El operador *index* lo que devuelve es el índice que ocupa el submódulo actual en el vector de módulos.

Constantes de tiempo

En cualquier lugar donde se quiera poner constantes numéricas que signifiquen un tiempo real, se puede especificar el tiempo en unidades tales como milisegundos, minutos u horas, con las siguientes nomenclaturas: ns, us (microsegundos), ms, s, m, h, d.

Números aleatorios

En Omnet++ se pueden obtener números aleatorios en función a alguna distribución dada. Por ejemplo una distribución uniforme.

Las distribuciones implementadas son:

- Uniforme
- Exponencial
- Normal, o normal truncada

Todas las funciones generan los números a partir del número generador cero. Con el uso de *genK_*-prefijo podemos indicar el número a utilizar como generador, por ejemplo

```
genk_normal(2,100,5)
```

Valores de entrada

Para introducir valores de entrada a la simulación, la sintaxis es la siguiente:

```
input( 10, "Number of processors:" )
```

Nota: se puede omitir también el *prompt.*

Los valores se pueden introducir en el archivo de configuración, o si no se especifican en ningún sitio, el simulador nos pedirá un valor para esas variables al arrancar la simulación.

Funciones

En el lenguaje NED se pueden utilizar las siguientes expresiones matemáticas:

- La mayoría de funciones que se usan en el lenguaje C de la librería *math.h*
- Funciones que generan números aleatorios
- Definir otras nuevas

Para realizar nuevas funciones se deben codificar en C++ [11]. Debe coger 0,1,2 o 3 argumentos de tipo *double* y devolver un *double*. La función debe ser registrada en un archivo C++ con el macro *Define_Function().*Ejemplo:

```
#include <omnetpp.h>
```

```
double average(double a, double b)
{
return (a+b)/2;
}
Define_Function(average, 2);
```

Esta función realiza la media de dos números (a y b) *double* y devuelve un *double.* Para llamar a la función en el archivo NED sería de la siguiente manera:

```
module Compound
parameter: a,b;
submodules:
proc: Processor
parameters: av = average(a,b);
endmodule
```

Expresiones que modelan la visualización gráfica

Se denomina también *display strings.*

Especifican la apariencia que tendrán los módulos en la simulación.

El formato de estas expresiones viene dado por etiquetas separadas por ";", cada etiqueta referida a un aspecto distinto del componente que se esté modelando.

Las etiquetas que se usan para modelar un submódulo son las siguientes:

- *p* que designa la posición del submódulo en la pantalla. Se puede especificar que un vector de submódulos se agrupen en una fila, en una columna, en forma matricial o en elipse; utilizando las siguientes palabras claves: *row, column, matriz ring* . Pe: *p=xpos,ypos,column,deltax.*
- *b* que designa la forma gráfica del módulo. Puede elegirse una forma rectangular (*rect*) o forma de elipse (*oval*).
- NED proporciona una serie de iconos para elementos determinados, como puede ser el icono de un computador. En ese caso, en la expresión se indica con la etiqueta *i.*
- La etiqueta *o* corresponde con el color del submódulo en el formato r, g, b con la siguiente sintaxis: #rrggbb.

Ejemplos:

```
"p=100,60;i=workstation"
"p=100,60;b=30,30,rect;o=4"
```

Para las conexiones, las etiquetas que las modelan son:

- *M* que indica la posición exacta del emplazamiento de la conexión en el submódulo. Puede tener los siguientes valores: auto (automática), north (norte), south (sur), east (este), west (oeste). Además permite una configuración manual, mediante la cláusula *manual* que tiene la siguiente sintaxis: **m=manual**,*srcpx,srcpy,destpx,destpy,* donde *srcpx,srcpy* indican la posición de la conexión en el módulo origen y *destpx,destpy* indican la posición de la conexión en el módulo destino.
- La etiqueta *o* tiene la función de elegir el color de la conexión, mediante el formato r, g, b con la sintaxis: #rrggbb.

Ejemplo:

```
"m=a;o=blue,3"
```

2.3.2.9 Editor Gráfico GNED

Permite diseñar los módulos gráficamente. Trabajo con los archivos *.ned* por lo que ahorra estar escribiendo el código de diseño de los módulos y submódulos (entre otras muchas cosas). Puedes cargar cualquier archivo *.ned* existente, editar los módulos compuestos de forma gráfica y salvar el archivo.

Además permite realizar todas operaciones que se han descrito anteriormente: diseñar las conexiones simples, importar módulos, diseñar los *display string...*

Lo más importante es que GNED es una herramienta que permite editar gráficamente y coger ese gráfico y editarlo textualmente, o al revés. Los cambios en el archivo fuente serán actualizados rápidamente en el gráfico.

La mejor forma de entender el GNED es abriéndolo e investigar todos sus rincones debido a que no tiene un manual específico, aunque basándose en un ejemplo se puede entender toda su funcionalidad fácilmente.

2.3.3 Los Módulos Simples

Las actividades realizadas por el módulo simple son implementadas por el usuario. Los algoritmos son codificados en C++ [11], usando la *Omnet++ class library*. Las siguientes secciones contienen una pequeña introducción a la simulación en eventos discretos en general, como se implementan esos conceptos y dar una pequeña visión práctica de cómo diseñar y codificar estos módulos simples.

2.3.3.1 Conceptos de Simulación

Simulación de eventos discretos (DES)

Es un sistema donde los cambios de estado (eventos) ocurren en instantes discretos de tiempo. Se asume que no ocurre ningún evento entre dos eventos consecutivos. Los sistemas que se pueden modelar como sistemas de eventos discretos se simulan mediante DES. Por ejemplo, en una red de ordenadores, normalmente se ven como sistemas de eventos discretos. Algunos de estos eventos serían el comienzo de transmisión del paquete, el fin de transmisión del paquete y la expiración del tiempo de retransmisión.

En este ejemplo implica que entre el comienzo de transmisión del paquete y del fin de transmisión no ocurre nada interesante. El estado del sistema quedaría en transmisión. Notar que este modelado de DES siempre queda en manos del programador. Si se deseara transmitir bits, los eventos más significativos serían: *start* y *end* de transmisión de los bits.

El tiempo en el que ocurre un evento se suele denominar *event timestamp* pero en Omnet++ la palabra está reservada, por lo que se denomina *arrival time.*

Evento *loop*

Las simulaciones de eventos discretos mantienen una tabla de eventos futuros en una estructura denominada FES (*future event set*). La mayoría de simuladores trabajan con un pseudocódigo similar a este:

```
initialize -- this includes building the model and
inserting initial events to FES
while (FES not empty and simulation not yet complete)
{
retrieve first event from FES
t:= timestamp of this event
process event
(processing may insert new events in FES or delete
existing ones)
}
```

```
finish simulation (write statistical results, etc.)
```

Lo primero, en el paso de inicialización, se contruye la estructura de datos representando el modelo de simulación, introduciendo los eventos iniciales para arrancar el sistema en la FES.

El siguiente bucle procesa todos los eventos de la FES. Son procesados en cada *timestamp* y según el orden establecido. En el bucle se pueden eliminar eventos del FES, por ejemplo cuando cancelamos *timeouts.*

La simulación llega a su fin cuando dentro de la FES no queda ningún evento para procesar, o cuando se alcanza el tiempo de CPU.

Eventos en Omnet++

Omnet++ usa mensajes para representar eventos. Cada evento es una instancia de la clase *cMessage* o una de sus subclases. Los mensajes se mandan de un módulo a otro, a través de las conexiones establecidas.

Los eventos como *timeouts* son implementados en los módulos que envían los mensajes a sí mismos, mientras que lo normal es que cuando se mande un mensaje sea a otro módulo, así cuando el módulo destino reciba el mensaje equivaldrá a que ese módulo sufre un evento.

El tiempo de simulación en Omnet++ es almacenado en una variable de C++ denominada *simtime_t* que es un *typedef* de tipo doble.

Los eventos son procesados por la lista de eventos futuros en el mismo orden en que llegaron. Mas específicamente, dados dos mensajes, se aplican las siguientes reglas:

- El primer mensaje en llegar es el primero en ejecutarse. Si llegan en el mismo tiempo =>
- El que mayor prioridad tenga se ejecutará. Si tienen la misma prioridad =>
- El que se programó primero o el que deba ser enviado antes se ejecutará.

Nota: la prioridad es un atributo que establece el usuario

El guardar el tiempo de simulación en una variable de tipo *double* puede ocasionar incoherencias a la hora de comparar dos tiempos, debido a la precisión que da la máquina, puesto que dos números flotantes, calculados con distintos algoritmos a veces no son iguales, aunque teóricamente sí lo sean. Por eso hay que llevar cuidado a la hora de calcular tiempos, y calcularlos de la misma manera.

Implementación de la FES en Omnet++

La correcta implementación de la FES es un aspecto crucial en el desarrollo de un DES. En Oment++ la FES se implementa con un *binary heap. Heap* es el mejor algoritmo conocido aunque otras estructuras de datos más exóticas como *skiplist* pueden ser más útiles en algunos casos que *heap.* En caso de estar interesado en saber como está implementada la FES, esta se puede ver en la clase *cMessageHeap* pero esto no es un aspecto importante a la hora de trabajar con ella.

2.3.3.2 Definición de Tipo Módulo Simple

Descripción

La implementación en C++ de un módulo simple es:

- Declaración de la clase de módulo. La clase hereda de *cSimpleModule*
- Registrar el tipo de módulo (*Define_Module* o *Define_Module_Like macro*)
- Implementación de la clase

A continuación se muestra un ejemplo, que corresponde a una posible implementación del protocolo de ventana deslizante:

```
// file: swp.cc
#include <omnetpp.h>
// module class declaration:
class SlidingWindow : public cSimpleModule
{
Module_Class_Members(SlidingWindow,cSimpleModule,81
92)
virtual void activity();
};
// module type registration:
Define_Module( SlidingWindow );
// implementation of the module class:
void SlidingWindow::activity()
{
int window_size = par("window_size");
...
}
```

Para ser capaz de utilizar el módulo en el archivo .ned debemos realizar una declaración similar a esta:

```
simple SlidingWindow
parameters:
window_size: numeric const;
gates:
in: from_net, from_user;
out: to_net, to_user;
endsimple
```

2.3.3.3 Declaración de un Módulo

Pasos:

- Especificar que se va a usar la clase como un módulo simple
- Asociar la clase con una interfaz declarada en NED

Existen dos formas distintas de declarar un módulo:

```
Define_Module(classname);
Define_Module_Like(classname, neddeclname);
```

Con la primera forma asociamos la clase con el módulo simple declarado en NED de mismo nombre.

La segunda forma asocia un módulo simple declarado en NED, de distinto nombre.

Para los módulos compuesto es el compilador *NEDC* el que genera la declaración del módulo, puesto que todo módulo necesita una declaración. Sin embargo es responsabilidad del programador poner esa línea en el archivo C++ para cada módulo simple.

2.3.3.4 Declaración de la Clase

Los módulos simples derivan de la clase *cModuleSimple*. Además de sobrescribir cuatro funciones *initialize(), activity(), handleMessage(cMessage *msg), finish()*, se debe implementar un constructor y algunas funciones más.

Implementación del constructor usando un macro

Se debe usar el macro:

```
Module_Class_Members( classname, baseclass, stacksize);
```

Los dos primeros parámetros so obvios pero el *stacksize* necesita una pequeña explicación. Si se implementa el método *activity(),* el código del módulo corre como si se tratase de un *thread,* por lo que habrá que separar la pila (*stack*). Ejemplo:

```
class SlidingWindow : public cSimpleModule
{
Module_Class_Members( SlidingWindow,cSimpleModule,8192)
...
};
```

Expresión completa del constructor

Se pueden implementar dos tipos:

- Un constructor con una lista de argumentos:

  (const char *name, cModule *parentmodule, unsigned stacksize= *stacksize*)

- Un constructor donde sólo se indica el nombre de la clase y devuelve el nombre de la clase como un *char.*

La ventaja de utilizar estas formas de constructor es que hay total control sobre todos los parámetros, pudiendo inicializar cualquiera de ellos desde aquí. No se debe cambiar el número o tipo de los argumentos del constructor porque el código de Omnet++ llama a ese constructor cuando genera el código. También hay que tener en cuenta que hay que sobrescribir el constructor *className()*

Decisión tamaño de pila para los métodos *activity/handleMessage*

Si se tiene que utilizar un tamaño de pila cero el método que se debe implementar es el *handleMessage*, mientras que si se debe usar un tamaño de pila distinto de cero el método que debería utilizarse es el *activity.*

2.3.3.5 Añadir Funcionalidad a los Módulos

Lo que queremos es implementar los métodos: *initialize(), activity(), handle Message(cMessage *msg) y finish().*

Activity()

Con este método se puede codificar como un *thread.* Se puede hacer que espere un mensaje entrante en algún punto del código y suspender la ejecución por algún tiempo. Cuando termina de funcionar este método, termina la función del módulo. Las funciones más importante que pueden utilizarse en el cuerpo del método son las siguientes:

- *Recevie()* -> (o toda la familia de funciones de este tipo) Para recibir un mensaje
- Wait()-> suspende una ejecución
- Familia de funciones *send()* -> enviar un mensaje

- *ScheduleAt()*-> para programar un evento
- *End()* -> Fin de ejecución

Cuando se llama a la familia de funciones *receive()* se forma un bucle infinito.

Cuando se diseña un módulo con *activity()* se necesita un mensaje para activar el método. Estos mensajes son insertados dentro del FES automáticamente por Omnet++ en el inicio de la simulación, justo antes de que *initialize()* sea llamada.

El campo de aplicación de este método es en situaciones donde hay muchos estados pero transiciones limitadas, por ejemplo, cuando se programa una red que usa una única conexión de red local.

Métodos initialize() y finish()

Ya que las variables locales de *activity()* son preservadas a través de los eventos, se puede almacenar cualquier cosa en ellas. Lo único que necesitan antes de ser utilizadas es que se las inicialice. Para esto sirve el método *initialize()*.

Sin embargo se necesita *finish()* para escribir las estadísticas y terminar la simulación. Ya que este método no tiene acceso a las variables de *activity()* las variables deben de ser variables de clase para poder tener acceso a ellas.

Un último detalle es que no se necesita de *initialize()* para inicializar las variables, puesto que se puede hacer al principio del método *activity(),* justo antes de entrar en bucle infinito.

Método handelMessage(cMessage *msg)

La idea es que se llame a la función cada vez que procesará el mensaje y después retornará al principio. El tiempo de procesado es potencialmente distinto para cada llamada.

Para poder usar el método se debe indicar que el tamaño de pila es cero (*stacksize =0*). Esto le dice a Omnet++ que no utilice *activity()*.

Las principales funciones que se pueden llamar dentro del cuerpo del método son:

- Familia de funciones *send()*
- *ScheduleAt()*
- *cancelEvent(),* para cancelar un evento

No se puede usar ni la familia de funciones *receive()* ni la función *wait().*

Tampoco se puede utilizar la función *end()* puesto que esa función es para finalizar un método que trabaje como un *thread.*

El área de aplicación principal de este método es el siguiente:

- Módulos donde mantienen pocos o ningún estado de información, como paquetes de sumidero
- Donde hay muchos posibles siguientes estados a partir de un estado dado.

2.3.4 Construcción y Ejecución de Simulaciones

Un modelo en Omnet++ consiste en:

El lenguaje NED de descripción de topologías (archivos *.ned*)

Módulos simples en C++ (archivos *.cc*)

El sistema de simulación provee los siguientes componentes que serán parte de la simulación ejecutable:

El *simulation kernel* con las librerías de clases. Este archivo tiene una extensión *.a* o *.lib* normalmente almacenado en el subdirectorio sim del directorio raíz de Omnet++.

Interfaces de usuario. Normalmente almacenadas en el subdirectorio envir del directorio raíz de Omnet++.

Para construir los programas primero se compilan los archivos *.ned* en un archivo fuente C++, usando el compilador NEDC. Después todas las fuentes C++ son compiladas y linkadas con el kernel asignándoles un intefar de usuario para ejecutar la simulación. La figura 2.9 nos muestra este proceso.

A la hora de compilar los archivos se utiliza el comando *opp_makemake* –opciones, para crear el archivo que contendrá las reglas de compilación (*Makefile*).

La opción –h nos permite visualizar todas las demás opciones de compilación que ofrece Omnet++. Algunas de estas son:

- (sin opciones) -> nos compila todos los archivos del directorio en el que se encuentren, dando el nombre del directorio al archivo ejecutable.
- –f -> fuerza al compilador a sobrescribir el archivo de configuración, en caso de tener uno ya creado
- -o +*nombre* -> indica el nombre del archivo ejecutable

Para compilar, una vez creado el archivo de compilación, se ejecuta el archivo *Makefile* y aparece el archivo ejecutable (en caso de no tener errores de compilación en el código).

A la hora de arrancar la simulación se puede seleccionar una serie de opciones, por ejemplo, *nombre_del_ejecutable –f* para indicar cual es el nombre del archivo de configuración. Por defecto *omnetpp.ini.* También se puede seleccionar que *run* deseamos simular (en caso de tener varias simulaciones declaradas, con la opción *–r*.

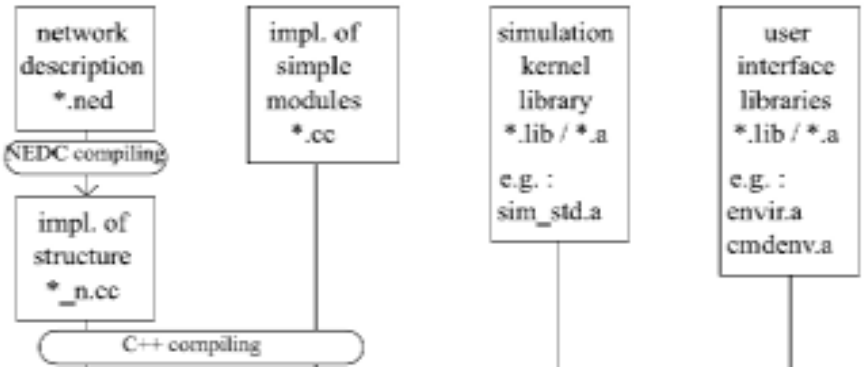

Fig.2.9. Proceso de construcción de una simulación

2.3.4.1 Archivo de Configuración (*omnetpp.ini)*

Este archivo se ha comentado en secciones anteriores. Ahora detallaremos su contenido para explicar su función dentro del simulador.

El archivo contiene opciones que controlan como va a ser ejecutada la simulación y puede contener valores de los parámetros del modelo. El archivo consiste en una serie de entradas agrupadas por secciones. Estas secciones son:

```
[General]
[Cmdenv], [Tkenv],...
[Parameters]
[OutVectors]
[DisplayStrings]
[Machines]
[Slaves]
[Run 1], [Run 2], [Run 3],...
```

Las opciones más importantes de la sección “General” son:

- Permitir mostrar *warnings* en la simulación
- Indicar la red que vamos a simular (de todas las creadas)
- El tiempo de CPU
- El tiempo de simulación

En la sección de *Parameters* o *Run X* se establecen los valores de los parámetros designados en el modelo (en el archivo *.ned*). Si los parámetros no se fijaran a algún valor concreto, al iniciar la simulación el propio Omnet nos pedirá que asignemos el valor a esas variables todavía no establecidas.

En la sección *OutVecotrs* se establece el módulo en el que los *output vectors* se utilizan para recoger las estadísticas dejarán toda la información recogida. El nombre del vector vendrá establecido en el constructor de su declaración (en el código C++).

La sección *Display String* permite al usuario cambiar el aspecto gráfico de algún módulo, en caso de no haberlo realizado en el archivo NED.

El resto de secciones son menos utilizadas, y no son relevantes para realizar este proyecto, por lo que no se mencionarán.

A continuación se muestra un ejemplo de archivo *.ini:*

```
# omnetpp.ini
[General]
ini-warnings = false
network = token
distributed = no
snapshot-file = token.sna
output-vector-file = token.vec
log-parchanges = no
parchange-file = token.pch
random-seed = 1
sim-time-limit = 1000ms
cpu-time-limit = 180s
total-stack-kb = 2048
[Cmdenv]
runs-to-execute = 1-3,5
module-messages = yes
verbose-simulation = no
display-update = 100ms
[Parameters]
token.num_stations = 3
token.num_messages = 10000
[Run 1]
token.wait_time = 10ms
[Run 2]
token.wait_time = 30ms
```

2.3.4.2 Problemas Típicos con las Simulaciones

"Stack violation (*FooModule* stack too small?) in module *bar.foo"*

Omnet detecta cuando un módulo usa más pila de la que se le ha asignado. Para solucionarlo se debe mirar si el módulo, después de realizar su trabajo libera el espacio ocupado de pila.

"Error: Cannot allocate *nn* bytes stack for module *foo.bar"*

Para solucionar este error lo que se debe hacer es incrementar el tamaño de la pila desde el archivo de configuración con una instrucción similar a esta:

```
[General]
total-stack-kb = 2048 # 2MB
```

Aquí se le da un tamaño de pila de dos megabytes. Nosotros podemos poner la cantidad de pila necesaria para hacer funcionar el programa (y que este permitida por el computador donde trabajemos).

2.3.5 Análisis de los Resultados

En esta última sección detallaremos grosso modo la forma de obtener resultados gráficos de las simulaciones realizadas en Oment++.

Para interpretar los resultados almacenados en los *output vectors* Omnet proporciona una herramienta denominada Plove (Apéndice A). Con esta se pueden representar gráficamente todos los resultados obtenidos, en forma de histogramas, diagramas de cajas, puntos, etc.

Otra función interesante de esta herramienta es que permite representar varios vectores dentro de la misma gráfica, contrastando los resultados de cada vector.

Por último, con Plove también se pueden realizar filtros a los vectores, de forma que representemos sólo lo que nos interesa. Estos filtros se pueden realizar como expresiones *awk* , debido a que el formato de los *output vector* es en filas y columnas,

por lo que con *awk* se puede procesar. Un ejemplo de un archivo *output vector* es el siguiente:

```
mysim.vec:
vector 1 "subnet[4].term[12]" "response time" 1
1 12.895 2355.66666666
1 14.126 4577.66664666
vector 2 "subnet[4].srvr" "queuelen+queuingtime" 2
2 16.960 2.00000000000.63663666
1 23.086 2355.66666666
2 24.026 8.00000000000.44766536
```

En la primera línea se indica el nombre del vector en la segunda línea se especifican los distintos campos.

Capítulo 3
CONMUTADORES CON COLAS A LA SALIDA

3.1 Introducción

Un conmutador de paquetes es, básicamente, un dispositivo capaz de conectar *N-entradas* con *M-salidas* y encaminar los paquetes que llegan a sus entradas hacia las salidas apropiadas. Se puede dar que un conmutador tenga la misma cantidad de entradas que salidas, en cuyo caso denominaremos conmutador de matriz cuadrada.

El conmutador puede diseñarse de forma que tenga o no bloqueo interno, pero incluso en conmutadores sin bloqueo interno puede ocasionarse contención de puerto de salida. Este problema aparece cuando se presentan en un mismo instante de tiempo, dos o más paquetes en distintas entradas pero que quieren dirigirse al mismo puerto de salida. Debido a este problema los conmutadores poseen una memoria interna con lo cual el conmutador pueda transmitir un paquete quedando el resto almacenados en la memoria interna del mismo. Según el lugar donde se coloque la memoria interna se podrá diseñar un tipo diferente de conmutador. Se explicará la topología de un conmutador con colas a la salida, que son conmutadores que tienen una memoria individual por cada puerto de salida (tal y como se explicó en el capítulo 2, sección 2.1.2). Posteriormente, este capítulo hará un estudio sobre su comportamiento ante un tipo de patrón de tráfico de entrada. El estudio se realizará mediante la generación de paquetes de tamaño fijo, introduciendo una serie de parámetros de entrada para realizar la simulación y obteniendo unos parámetros de salida para evaluar el conmutador. Para ello se utilizará la herramienta de simulación Omnet.

3.1.1 Conmutadores con Colas a la Salida

Este tipo de conmutador es el que va a centrar toda la atención en este capítulo pues el objetivo del capítulo es llegar a entender su estructura, funcionamiento, prestaciones, limitaciones, etc.

Un conmutador con colas a la salida (Fig. 3.1) tiene la siguiente estructura: la memoria interna del conmutador se encuentra en los puertos de salida, justo después del módulo de conmutación. Con esta estructura lo que se consigue es que todos los paquetes que lleguen a los puertos de entrada en un mismo *slot* de tiempo se puedan encaminar a su salida correspondiente, en función de los datos de encaminamiento que portan. Una vez los paquetes se dirigen a las salidas, allí quedarán almacenados en una cola con disciplina FIFO. En ese *slot* de tiempo la cabeza de la cola será leída y si contiene algún paquete que transmitir será enviado. Con esta técnica eliminamos el bloqueo que existía con los conmutadores con colas a la entrada

En cuanto al coste de un conmutador con colas a la salida (en comparación con un conmutador con colas a la entrada, por ejemplo) se puede decir que es mayor puesto que, en primer lugar, las colas en un conmutador con colas a la salida deben de ser capaces de almacenar *N* paquetes en cada *slot* temporal. Esto hace que las

memorias empleadas sean más caras. En segundo lugar, la matriz de puntos de cruce debe poder enviar hasta *N* paquetes a cada puerto de salida, por lo tanto, la construcción de este tipo de matrices será también más cara.

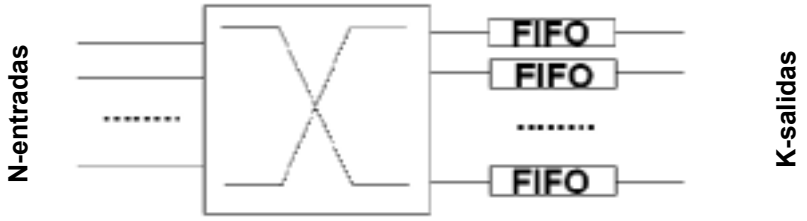

Fig. 3.1 Conmutador con colas a la salida

3.2 Descripción del Entorno

El conmutador con colas a la salida que se ha implementado está diseñado con la herramienta de simulación Omnet++ [5] [6]. Es una herramienta de simulación orientada a objetos, consistente en un anidamiento jerárquico de módulos, basada en eventos y que utiliza el lenguaje C++ [11].

El anidamiento jerárquico de módulos consiste en tener un diseño que lo forman una serie de módulos, pero que a su vez, estos módulos están compuestos por otros módulos, y así sucesivamente hasta llegar a los módulos más elementales del diseño. Estos módulos elementales se denominan *simple modules* mientras que los módulos diseñados a partir de estos módulos simples se denominan *compound modules.* En este capítulo, las topologías implementadas utilizan únicamente módulos simples.

Las simulaciones en Omnet están controladas por eventos. Estos eventos son sucesos que previamente se programan, a la hora de diseñar el entorno, y que modelan la simulación.

Por último, la función que tiene aquí el lenguaje C++ es la de describir el comportamiento de los módulos simples que forman la base de la estructura jerárquica de la arquitectura que se implemente posteriormente con Omnet. Para describir un módulo lo único que hay que hacer es implementar una interfaz que Omnet nos facilita. Además, como el lenguaje es C++, podemos heredar propiedades de otros objetos ya declarados anteriormente, que se usarán para describir el comportamiento del módulo.

3.2.1 Arquitectura Omnet para el Desarrollo de un Conmutador con Colas a la Salida

A la hora de diseñar una arquitectura en Omnet lo que primero se hace es buscar los objetos que servirán para desarrollar esa arquitectura. En otras palabras, encontrar los que serán módulos simples del diseño a implementar. En este caso, lo que se desea es implementar un conmutador y se toman como módulos simples los siguientes: módulo de control, generador, *switch,* colas, sumideros.

Posteriormente se estudian cuales son los eventos que se suceden en el conmutador que se quiere implementar y una vez estén claros, se comenzará a

implementar los módulos en C++, teniendo en cuenta esos eventos que queremos que se generen.

Lo último es diseñar el aspecto gráfico, indicando las conexiones entre módulos, (aunque a veces sea necesario modificar el código de los módulos al mismo tiempo que se diseña la parte gráfica de la arquitectura). La herramienta que se usa para diseñar el aspecto gráfico es *gned,* que es una herramienta que proporciona Omnet.

Es importante mirar el código de la implementación para comprobar cómo se diseña una arquitectura en Omnet, y comprobar como realizan el trabajo los módulos diseñados.

3.2.1.1 Módulo “Control”

La función principal de este módulo es la de controlar los eventos que se llevan a cabo en un *slot* de tiempo dentro del conmutador. Indica a cada módulo cuando debe realizar “su trabajo” .

Los eventos que se suceden, por orden cronológico, desde que se inicia el *slot* hasta que finaliza, son los siguientes:

- Generación del tráfico de entrada.
- Conmutación de los paquetes que llegan a los puertos de entrada del *switch* a los puertos de salida del *switch*.
- Transmisión de los paquetes que ocupan las cabezas de las colas que se encuentran en los puertos de salida del *switch*, siempre y cuando las colas no estén vacías.
- Almacenamiento de los paquetes, que envió el *switch* a los puertos de salida en el evento II, en las correspondientes colas y según el orden en que llegaron.

El único evento que el módulo de control no gestiona es la activación de los módulos sumideros para que estos eliminen los paquetes, pero es debido a que estos módulos no necesitan gestionarse como si fuesen una parte interna del conmutador.

El módulo de control está programado de forma que dentro de un *slot* de tiempo todos los restantes módulos realicen “su trabajo”. Los tiempos que distan entre eventos son aleatorios pero controlando siempre que se realicen todos dentro del mismo *slot.* El periodo de tiempo que dista entre cada evento es el siguiente (suponiendo el primer *slot* que va de 0 a 1): el generador simulará tráfico en el instante 0; el *switch* empezará su trabajo en el instante 0.2; la cola transmitirá en el 0.4 y almacenará los paquetes en 0.6; y al mandar ese paquete de control el módulo control esperará 0.4 para que se cumpla el ciclo de un *slot*. Cuando pasen esos 0.4 instantes volverá a realizarse el ciclo descrito anteriormente (generación de tráfico ...).

El módulo de control es el primer módulo que comienza a funcionar en la simulación, pues si no lo hiciese, ninguno de los restantes podría realizar su trabajo, además el correcto funcionamiento de este módulo es vital para el buen funcionamiento del conmutador implementado.

3.2.1.2 Módulo “Generador”

Es el encargado de simular la generación de tráfico de entrada al conmutador. Está implementado de tal forma que se puede generar sólo tráfico siguiendo una distribución de *Bernouilli* de carga *p*, pero abierto a nuevos posibles patrones de entrada. Si se seleccionasen otros patrones aparecería un mensaje que avisaría de

que no está implementado ese patrón. Lo único que habría que hacer sería implementar el código de ese patrón.

Este módulo se activa cada vez que reciba el paquete de control. Una vez recibido el paquete de control, primero obtendrá un número aleatorio entre 0 y 1 y se comparará con el valor de la carga para esa simulación. Si el número obtenido es menor que la carga se generará un paquete, de la clase *cMessage* de Omnet, de longitud fija, incluyéndole el tiempo en que se ha generado (para poder realizar luego las estadísticas) y el puerto de salida al que va dirigido. Toda esta información será procesada a lo largo de la simulación según el paquete avance por los módulos del conmutador. En caso de que el número obtenido sea mayor que la carga esperaremos al siguiente *slot* de tiempo a que nos vuelva a llegar la información del control para poder repetir el ciclo.

3.2.1.3 Módulo “Switch”

La función de este módulo es la de encaminar los paquetes que le llegan a sus puertos de entrada y dirigirlos, según la información que lleva cada paquete, al correspondiente puerto de salida.

El funcionamiento es el siguiente. Almacena todos los paquetes que le llegan antes de la señal de control en una cola auxiliar, esperando a que le llegue la señal de control de forma que lo active y comience “su trabajo”. Ese “trabajo” consiste en leer la cola auxiliar extrayendo los paquetes, siguiendo el orden en que llegaron los paquetes a la cola, y buscando la información de encaminamiento que estos portan en sus cabeceras, serán encaminados a una de las salidas que tiene el *switch* (estas salidas son los puertos de salida del conmutador implementado y a las que están conectadas colas, una por cada puerto de salida). Una vez que encamina todos los paquetes, este queda a la espera de la nueva señal de control almacenando mientras tanto, los paquetes que le llegan del generador.

3.2.1.4 Módulo “FIFO”

Este módulo tiene ese nombre por la disciplina que sigue: es una cola FIFO. Además, debido a la implementación que tiene, es una cola que primero transmite un paquete, si no esta vacía, y luego lee los paquetes recibidos y los almacena en la cola.

Cada módulo *fifo* se encuentra conectado a un puerto de salida del *switch.* Además tienen dos conexiones con el módulo de control, una que servirá para recibir la señal de control de transmisión y la otra para recibir la señal de lectura de los paquetes.

Funcionamiento de este módulo es el siguiente: la cola está siempre a la espera de recibir la señal de control de transmisión, almacenando todos los paquetes que recibe, en una cola auxiliar. Cuando llega ese paquete de control de transmisión lee la cabeza de la cola y si hay algún paquete lo transmite. Estos paquetes estarán marcados como paquetes cursados (en la simulación se distinguen porque tienen color rojo). Después recibe el paquete de control de lectura y comienza a pasar los paquetes que tiene en la cola auxiliar a la cola principal. La cola principal puede tener una cantidad de posiciones de memoria finita o infinita, es decir, la longitud de la cola puede ser limitada o ilimitada. Esto dependerá de la configuración que se desee. En caso de que la cola sea ilimitada no habrá problema, pues se pasarán todos los paquetes de la cola auxiliar directamente en la cola principal en el mismo orden en que llegaron los paquetes a la cola auxiliar. Sin embargo, el problema surge cuando la cola tiene longitud finita, pues para cada paquete que se extraiga de la cola auxiliar y se quiera introducir en la cola principal, habrá que comprobar si todavía quedan posiciones libres en la cola. En caso de que si quede espacio, se insertará el paquete en cola, pero en caso de estar llena, los restantes paquetes se marcarán como

paquetes perdidos (en la simulación son paquetes en negro), pero esto en la realidad equivaldría a información perdida.

Una vez realice todos estos pasos descritos, la cola quedará a la espera de los paquetes de control, para volver a realizar su función.

3.2.1.5 Módulo "Sumidero"

Estos módulos son los encargados de la recolección de los paquetes que las colas envían.

Habrá un sumidero conectado a cada cola y en la arquitectura se distinguen dos tipos de sumideros que se detallan a continuación.

Módulo "sumidero no estadístico"

Este tipo de sumidero lo único que hace es recibir paquetes y eliminarlos sin ningún tipo de acciones adicionales. Es por esto que su implementación es la más fácil que se encuentra en el diseño (la única línea de código que lleva este módulo en el cuerpo del método principal es *"delete msg;"*).

En el diseño del conmutador, todos los sumideros son no estadísticos salvo el que se encuentra a la salida de la primera cola que es sumidero estadístico.

Módulo "sumidero estadístico"

La función principal de este módulo es la misma que la del sumidero no estadístico: recolección y eliminación de los paquetes que le lleguen de la cola a la que está conectado. La diferencia es que además de esta tarea, realiza las estadísticas de la simulación.

El funcionamiento del sumidero estadístico es similar al anterior, recibe un paquete, lo procesa y después lo elimina. En la etapa de procesamiento lo que primero hace es comprobar su clase: paquete cursado o paquete eliminado. En caso de ser paquete cursado primero incrementa el número de paquetes cursados, luego comprueba el tiempo que ha estado en cola y después guarda información para realizar la distribución de retardo en un *array*. Para un paquete eliminado lo único que realiza es incrementar el número de paquetes perdidos.

El sumidero cuenta con un tiempo de transitorio, que es el tiempo que pasa desde que se inicia la simulación hasta que el conmutador adquiere cierta estabilidad, y que una vez que transcurre ese tiempo transitorio, todas las variables que se estaban usando para mostrar luego los resultados se *resetean*.

Al final de la simulación se procesan todas las variables que se han ido utilizando en la simulación, y además se genera un *script* para *gnuplot* de tal forma que al finalizar la simulación podemos visualizar, en un archivo .*eps*, el histograma de la distribución del retardo de la última simulación efectuada.

3.2.1.6 Diseño Gráfico del Conmutador

Una vez descritos todos los módulos que componen la arquitectura del conmutador con colas a la salida implementado, se comprobará que el aspecto gráfico que tiene se parece al diseño mostrado en la figura 3.1 de un conmutador con colas a la salida genérico. Decir que este diseño corresponde a un conmutador de tamaño 4x4 (cuatro puertos de entrada y cuatro puertos de salida), que es una de las 4 topologías implementadas. Las restantes topologías tienen la misma estructura pero con otros tamaños.

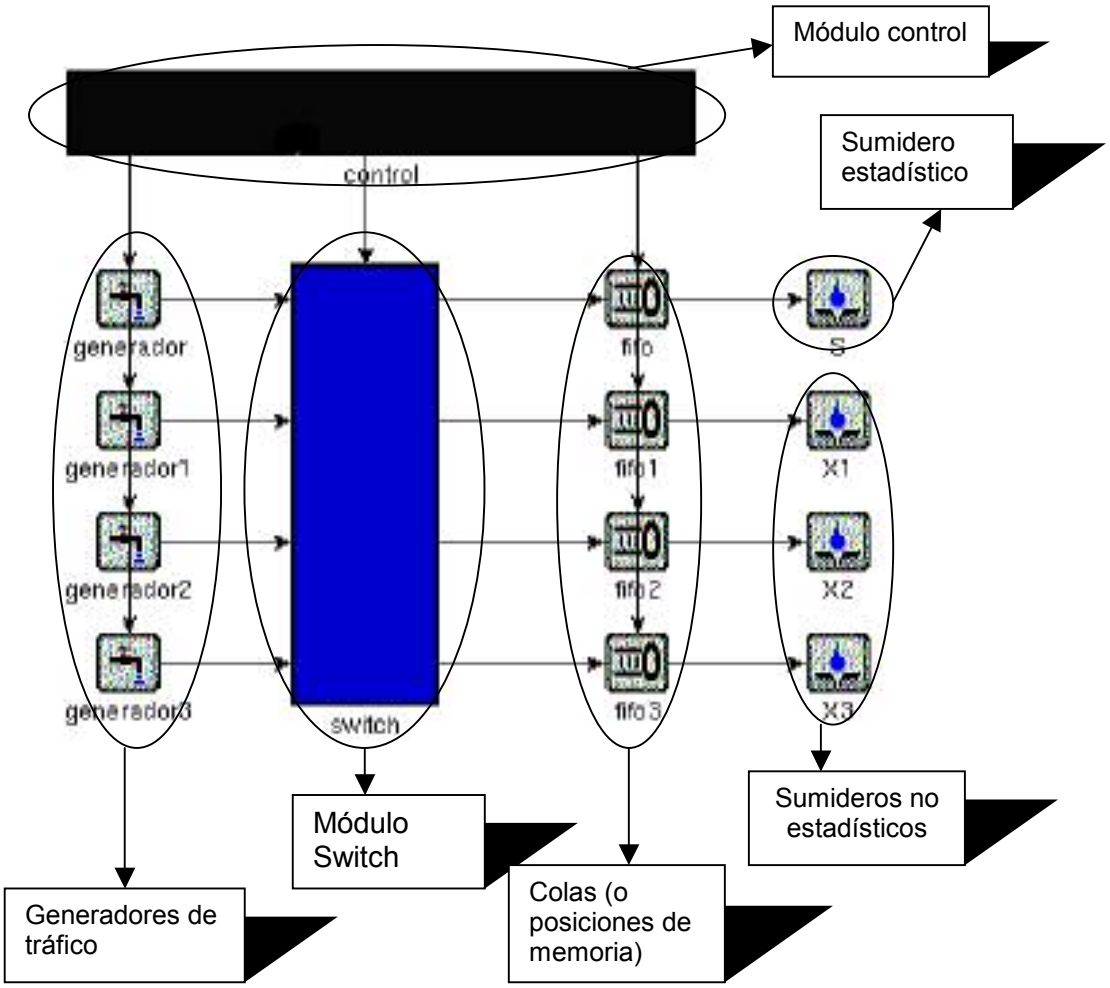

Fig. 3.2. Diseño de un conmutador con colas a la salida

Ahora se entiende de forma más clara la unión de todos los módulos mencionados anteriormente. A continuación se muestra el contraste entre la Fig. 3.1 y la Fig. 3.2:

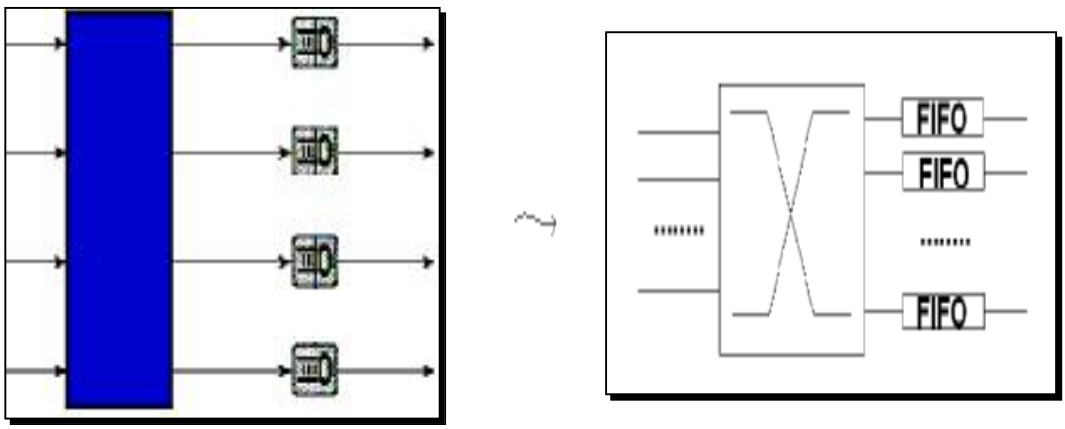

Fig. 3.3. Similitud entre el conmutador diseñado (Fig. 3.1) y uno genérico (Fig. 3.2)

3.2.2 Parámetros de Entrada

A la hora de realizar cualquier simulación en Omnet, previamente habrá que dar valores a una serie de parámetros de entrada definidos en el diseño de la arquitectura. Para asignar los valores a los parámetros se hará desde el archivo de configuración de Omnet (*omnetpp.ini*). Esto facilita mucho la introducción de parámetros puesto que no hay que modificar código y por tanto no hay que compilar cada vez que cambiemos los valores de estos parámetros.

Los parámetros empleados en los diseños resalizados se detallan a continuación.

Tipo de patrón de entrada

Este parámetro lo usará el módulo generador y su función es la de indicar que tipo de distribución de tráfico de entrada en el conmutador se desea tener en la simulación. Sólo está implementada una distribución de *Bernouilli* de carga *p*. En caso de seleccionar otro tipo de tráfico de entrada nos aparecerá por pantalla el siguiente mensaje: "*tipo de tráfico no implementado*".

El valor que esa variable tiene que tener para elegir un tráfico de *Bernouilli* es *1*. Para cualquier otro valor de la variable no se efectuaría nada.

Carga

Este se utilizará también en el módulo generador y sirve para simular el patrón de tráfico de entrada de *Bernouilli*, donde este parámetro indicará la carga *p* a la que se trabajará. El rango de valores de este parámetro es el intervalo *[0,1]*. En el caso de asignar cero, nunca se generará tráfico en la simulación; y si por el contrario, ponemos la carga igual a uno, se generará un paquete en cada ciclo de reloj.

Tipo de b*uffer*

El módulo que usa este parámetro es la cola *FIFO.* La función que tiene es indicar al módulo que tipo de cola se quiere en la simulación: una cola o buffer de tamaño limitado, finito; o por el contrario, una cola o buffer de capacidad ilimitada o infinita. Si se elige un buffer finito, en nuestra simulación habrá pérdida de paquetes mientras que si el buffer es de tamaño infinito esas pérdidas no existirán. El valor que puede tomar esta variable es *1* en caso de desear una cola de longitud finita y *(–1)* en caso de querer una cola infinita.

Longitud del b*uffer*

Se utiliza en el módulo *FIFO.* Este parámetro sirve para indicar el tamaño que queremos darle a la cola, o dicho de otro modo, la cantidad de posiciones de memoria que queremos que tengan las colas.

Este parámetro tendrá sentido siempre que trabajemos con una cola de tamaño finito y los valores que puede tomar son números enteros comprendidos en el siguiente intervalo : [0 , ∞).

Tiempo de transitorio

Este parámetro es usado por el sumidero estadístico. La función de este parámetro es indicar el tiempo a partir del cual los datos que se recogen para realizar las estadísticas son válidos.

Para realizar esta función el sumidero programa un paquete para él mismo tal que se recibirá justo en el tiempo de transitorio. Cuando lo recibe y comprueba que es el paquete que programó lo que hace es *resetar* todas las variables que hasta ese tiempo estaba utilizando para acumular información y poder realizar luego los resultados de la simulación.

Los valores que se le pueden dar son [0 , ∞).

Tiempo de simulación

Para Omnet los *ticks* de reloj tienen duración un segundo, pero la interpretación que se le dará aquí será de *slots* temporales de duración una unidad de tiempo, sin especificar cuanto equivale en tiempo real ese *slot.*

El tiempo de simulación es un parámetro general de Omnet y lo utiliza para detener la simulación en un instante determinado. Cuando se quiera pasar un valor, se le pasará la cantidad de instantes temporales que se quiera simular. Puede darse que no se le pase ningún valor, en cuyo caso la simulación no terminará hasta que pulsemos *stop* dentro de la simulación.

3.2.3 Parámetros de Salida

Como parámetros de salida se entenderán los resultados obtenidos al realizar una simulación. Estos resultados son obtenidos en el módulo sumidero estadístico, y servirán para mostrar las características del conmutador evaluado ante distintos tamaños de búffer y distinta carga de entrada.

Todos los parámetros de salida se muestran y explican a continuación.

Número de paquetes recibidos

Indica cuantos paquetes se procesan el sumidero, tanto paquetes cursados como paquetes perdidos.

Número de paquetes cursados

Esta variable contendrá la cantidad de paquetes que han llegado a cursarse en el conmutador. Estos paquetes son paquetes de clase *ok_paq* (además se distinguen por el color rojo en la simulación).

Número de paquetes perdidos

Con este parámetro indicamos cuantos paquetes se perdieron a la hora de atravesar el conmutador. Para discernir estos paquetes de los cursados, estos van marcados de la clase *lost_paq* (se distinguen dentro de la simulación por ser paquetes de color negro).

Probabilidad de pérdida

Como su nombre indica, este parámetro muestra la probabilidad de pérdida sufrida ante unas condiciones de entrada dadas. La probabilidad mostrada estará en tanto por uno.

Retardo medio

Mostrará el retardo medio de paquete sufrido en la simulación realizada ante las condiciones de entrada establecidas al principio de la simulación.

Distribución de retardo de paquete

Indica cuál es la probabilidad de que un paquete se retarde en *X slot* de tiempo. Si la cola es de tamaño *N*, la probabilidad irá desde 1 hasta *N*, puesto que el conmutador primero transmite y luego lee, así que la probabilidad de retardo de 0 *slots* de tiempo será cero.

Histograma de la distribución de retardo de paquete

Se puede generar una gráfica donde se muestra el histograma de la distribución de retardo con los datos de la distribución de retardo de paquete. Este histograma se genera gracias a un *script*, producido por el sumidero estadístico, de *gnuplot*.

La forma de generar el histograma viene indicada en la pantalla de Omnet, una vez terminada la simulación y justo después de los resultados de salida.

A continuación se muestra un ejemplo de histograma de una simulación dada:

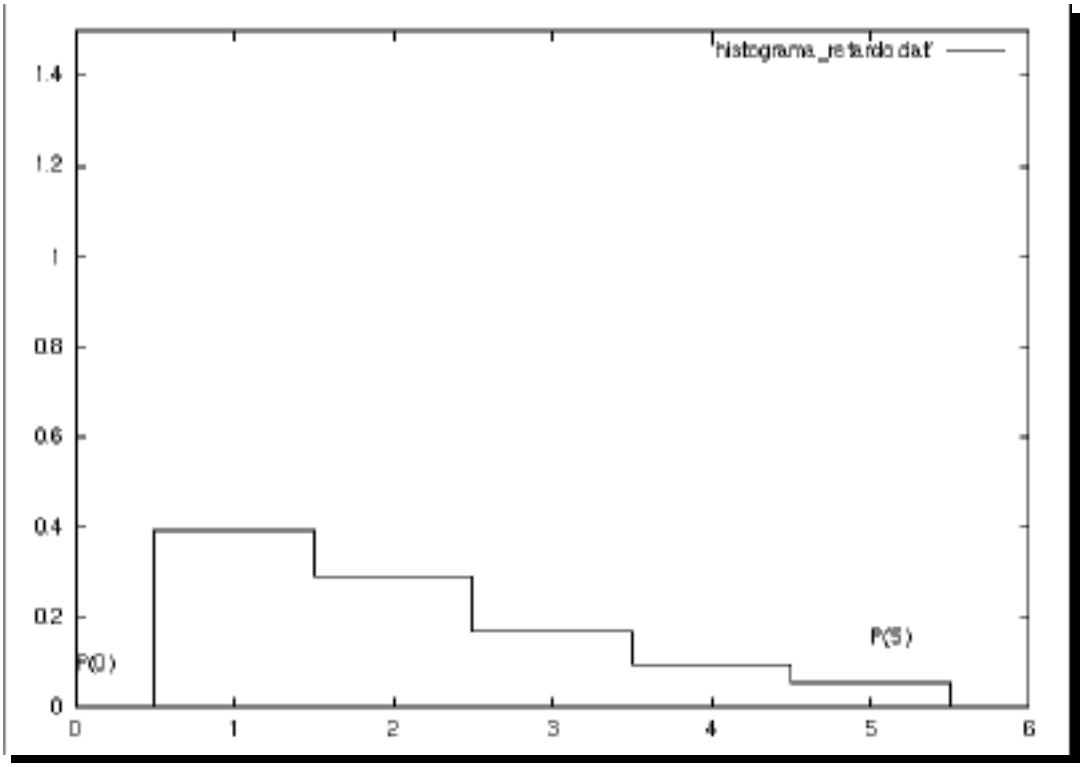

Fig.3.4. Histograma de una simulación

Visualización de los parámetros de salida

En Omnet, cuando una simulación termina se llama automáticamente a las funciones finish() de los métodos y el contenido se muestra en pantalla. Con esto se consigue mostrar los resultados finales de la simulación llevada a cabo. En el conmutador implementado, los resultados se muestran de la siguiente forma en la pantalla de Omnet:

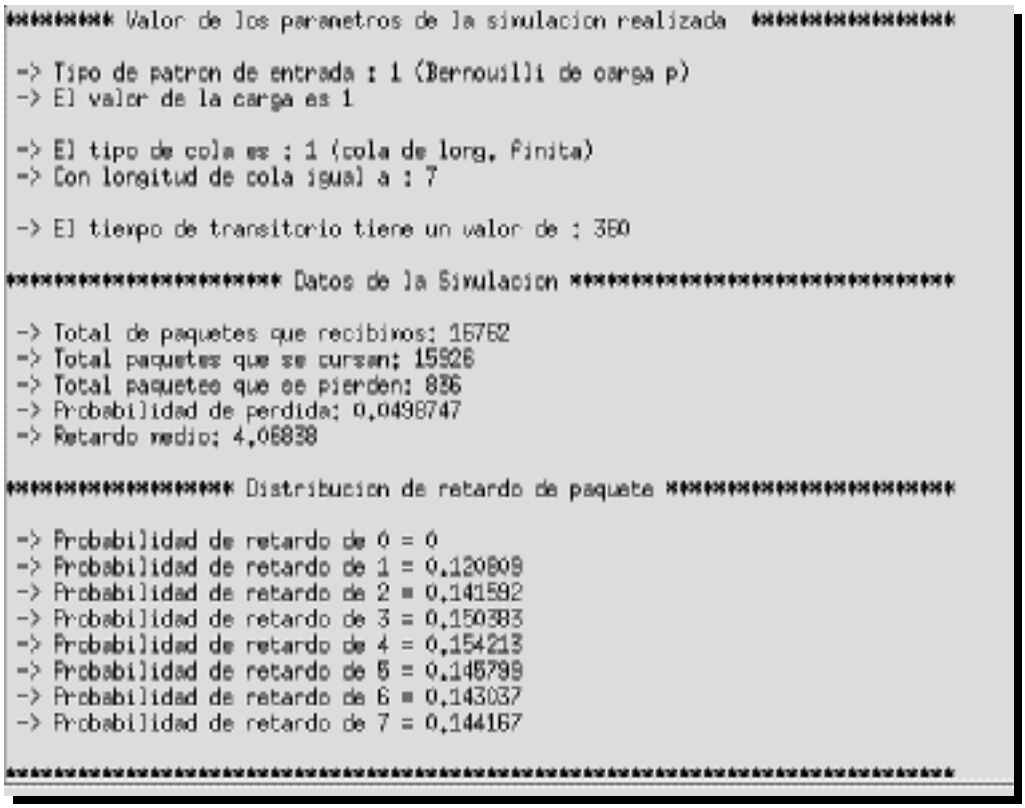

Fig.3.5. Resultados de una simulación

Se observa como se muestran todos los parámetros de salida que se han explicado antes, además de los parámetros de entrada de forma que el usuario, en caso de no acordarse de los parámetros de entrada que introdujo para esa simulación, se le pueda recordar.

3.3 Enunciado de la Practica: "*Estudio de Conmutadores de Paquetes con Colas a la Salida, para Paquetes de Longitud Fija*"

En este apartado se realizará un estudio de la arquitectura diseñada para distintos valores de los parámetros de entrada, así como una enunciar una serie de cuestiones que serán interesantes para la comprensión del funcionamiento del conmutador, pensar en nuevos problemas que pueden surgir en este tipo de conmutadores, cuales son las prestaciones de un conmutador de este tipo, ...

En primer lugar se recuerda de que hay implementadas 4 topologías distintas, es decir, hay 4 conmutadores distintos, pero todos son conmutadores con colas a la salida. Estos se distinguen según su tamaño: 4x4, 4x8, 8x4, 8x8.

Por último, decir que se dividirá la sección en dos partes: la primera servirá para estudiar en profundidad las características de un conmutador con colas a la salida. Para ello se tomará una de las cuatro topologías implementadas como referencia. En la segunda parte se contrastarán las distintas topologías implementadas para ver las prestaciones que ofrece cada conmutador.

3.3.1 Parte I

Aquí se realizará el estudio de un conmutador con colas a la salida 4x4. La elección es de forma arbitraria.

A continuación se muestran una serie de cuestiones que ayudarán a comprender el funcionamiento de un conmutador con colas a la salida.

Cuestión 1

Mirar el código de cada módulo que forma el conmutador, de forma que quede totalmente claro y se comprenda como es el funcionamiento de cada módulo. Así mismo, observar que ese funcionamiento es exactamente el descrito en la sección 3.2 de este capítulo, donde se presentan los módulos.

Cuestión 2

Una vez esté claro el apartado anterior, arrancar el programa (./*nombre del ejecutable*) y seleccionar Run 1. Aparecerá la topología que se desea estudiar y la pantalla principal de Omnet. Mediante la simulación paso a paso (F4) observar cuáles son los eventos a lo largo de un *slot* de tiempo. ¿Cuántos eventos se distinguen?¿Qué pasa con las colas en el primer *slot* de tiempo al llegar el paquete de control de transmisión? Explicar ahora por qué el retardo de un paquete nunca puede ser cero.

Cuestión 3

Observar un paquete a lo largo de un slot de tiempo. ¿Qué le ocurre cuando sale de cada módulo?¿Cómo va marcado ese paquete?

Cuestión 4

Cambiar el valor de la carga a 0.1 y el de longitud de la cola a 5. Para ello abrir el archivo omnetpp.ini utilizando el comando *emacs omnetpp.ini*. En la sección donde se muestran los parámetros de un conmutador 4x4 variar la carga y longitud de cola. El tiempo de simulación fijarlo a 345600 *slots* y en el tiempo de transitorio poner 14400 *slots*. Arrancar el programa de la misma forma que en la cuestión 1. Hacer una simulación rápida (F7) y cuando acabe el tiempo de simulación llamar a las funciones finish() (en el menú Inspect->call finsih() function), y tomar nota de la probabilidad de pérdida y el retardo medio. Realizar esto para los siguientes valores de carga: 0.2, 0.3 , 0.4, 0.5, 0.7, 0.9 ,0.99; manteniendo fijo el tamaño de buffer y apuntando los resultados de la probabilidad de pérdida y retardo medio para cada caso.

Realizar a continuación una representación gráfica con los datos tomados, una de la prob. de pérdida en función de la carga y otra del retardo medio en función de la carga, sabiendo que la longitud del buffer es 5.

¿Cuál es la razón de que se produzcan estos resultados?

Cuestión 5

Se realizará ahora un experimento similar, pero en lugar de ser la carga la que se varíe, ahora será el tamaño de buffer el que se modificará en cada caso. El valor de carga que daremos será una carga alta: 0.99. Se empezará tomando datos desde un tamaño de buffer igual a 5 y proseguiremos con: 10,20,30,40,50,75.

Ahora, en lugar de representar el retardo medio en función del tamaño del buffer sólo representar la probabilidad de pérdida sufrida ante distintos tamaño de buffer y a una carga de 0.99.

- Sin haber representado el retardo medio y con lo visto hasta ahora responder a: ¿cómo sería la evolución de una gráfica en la que se representa el retardo medio en función del tamaño del buffer, suponiendo la

carga expuesta? Realizar la gráfica del retardo medio en función del tamaño de *buffer*.

- Una vez realizada la gráfica de la probabilidad de perdida ante distintos tamaños de buffer, ¿sería posible tener una red conmutación sin ninguna pérdida con colas de tamaño fijo? Si es posible indíquese cómo y si sería factible.
- Qué solución es mejor a la vista de los resultados, teniendo en cuenta que el tráfico de entrada es bastante alto: usar colas de posiciones de memoria limitadas o ilimitadas.

3.3.2 Parte II

La función de esta parte será la de contrastar las distintas topologías implementadas, así como ver las prestaciones de cada una, ventajas, inconvenientes... Todas las topologías que se verán son conmutadores con colas a la salida sólo que de distintos tamaños.

A continuación se muestran las arquitecturas de las 4 topologías implementadas:

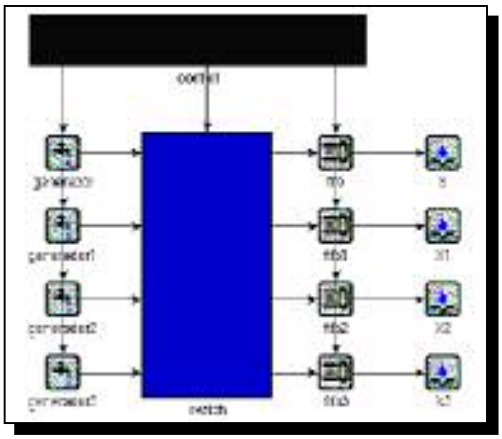

Fig.3.6. Arquitectura Conmutador 4x4

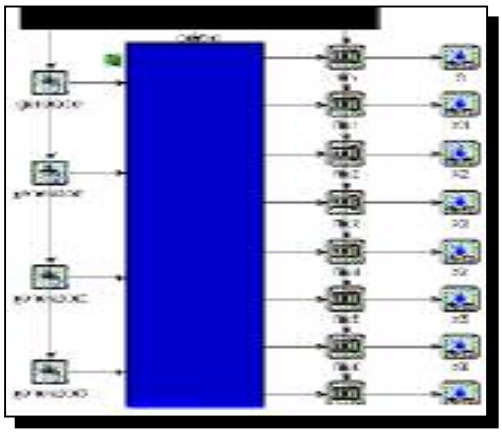

Fig.3.7. Arquitectura Conmutador 4x8

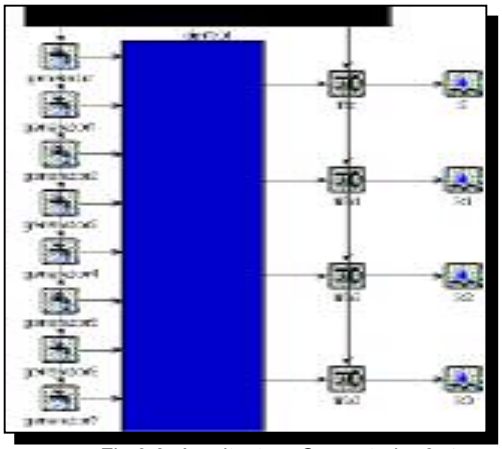

Fig.3.8. Arquitectura Conmutador 8x4

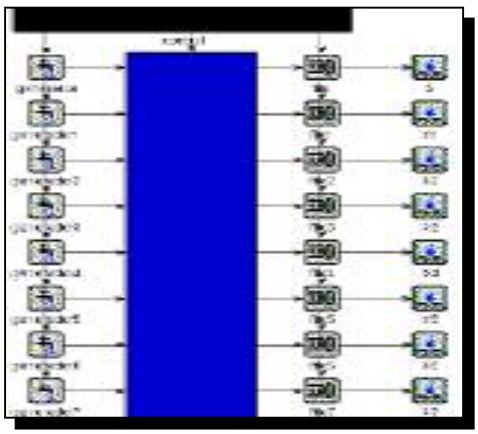

Fig.3.9. Arquitectura Conmutador 8x8

Cuestión 1

La primera cuestión que se presenta es diferenciar cual sería el comportamiento de cada conmutador, sin haber simulado nada, solamente teniendo en cuenta lo dicho hasta ahora, ante la misma carga de entrada y las mismas posiciones de memoria en los puertos de salida.

Para comprobar empíricamente esto se realizará la siguiente simulación aplicada a cada topología: carga = 0.8, tamaño de buffer = 5, T_{trans} = 600 *slots* y T_{sim} = 345600 *slots*. Analizar cual es la probabilidad de pérdida y retardo medio en cada caso.

Es interesante también visualizar cual es la distribución del retardo de paquete de cada conmutador.¿Crees que son coherentes los resultados? Explicar por qué.

Como ejercicio propuesto comprobar los datos de una simulación idéntica pero con una carga = 0.1.

Cuestión 2

La segunda cuestión es observar la diferencia que existe entre conmutadores de tamaño cuadrado, entendiendo por esto conmutadores del mismo número de puertos de entrada que de salida, y los no cuadrados.

Cuestión 3

¿Cuál es la topología que mayor pérdida sufre?. Este tipo de conmutador qué nombre recibe.

3.4 Resultados Obtenidos

En esta parte del capítulo, y para finalizar el estudio de los conmutadores con colas a la salida, se expondrán las soluciones a las preguntas realizadas en la sección anterior, así como una breve conclusión final sobre este tipo de conmutadores.

3.4.1 Respuestas a la Parte I

Cuestión 1

Mirar el código de cada módulo que forma el conmutador, de forma que quede totalmente claro y se comprenda como es el funcionamiento de cada módulo. Así mismo, observar que ese funcionamiento es exactamente el descrito en la sección 3.2 de este capítulo, donde se presentan los módulos.

Todo lo referente al código de los programas está bastante claro y documentado dentro del propio código. Además, todo se ha ido exponiendo a lo largo de todo el capítulo y más concretamente en la sección 3.2 de forma muy detallada.

Cuestión 2

Una vez esté claro el apartado anterior, arrancar el programa (./nombre del ejecutable) y seleccionar Run 1. Aparecerá la topología que se desea estudiar y la pantalla principal de Omnet. Mediante la simulación paso a paso (F4) observar cuáles son los eventos a lo largo de un slot de tiempo. ¿Cuántos eventos se distinguen?

En la simulación de un *slot* paso a paso se puede comprobar que se suceden los siguientes eventos: primero se arranca el módulo de control (instante t = 0) y una vez activado manda las señales de control a los generadores (instante t = 0.2). Una vez recibidas estos comienzan su activación y es cuando generan paquetes(tráfico), los cuales llegan a las entradas del *switch*. Estos, aparentemente quedan almacenados dentro del *switch*, pero en realidad quedan en un buffer auxiliar esperando que el *switch* comience el encaminamiento de los mismos. Este se activa en el instante t =

0.4, que es cuando recibe el paquete de control. Entonces los paquetes atraviesan el *switch*, cada uno a través del puerto de salida que lleva marcado y se almacenan en las colas. Una vez dentro, al igual que en el *switch*, estos quedan en un buffer auxiliar esperando recibir la señal de control. En el instante t = 0.6 la cola recibe el paquete de control de transmisión, y si hay algún paquete en cola, se transmite, sino no. En caso de transmisión el paquete llegará al sumidero y allí se recolectará. Seguidamente, en el instante t = 0.8, se recibe el paquete de control de almacenamiento (o lectura) y los paquetes almacenados en la cola auxiliar pasan a la cola principal. Este ciclo vuelve a repetirse a partir del instante t = 1, que es cuando el módulo control manda de nuevo los paquetes de control a los generadores. Todos estos son los eventos que se suceden en el conmutador a lo largo de un slot de tiempo.

¿Qué pasa con las colas en el primer slot de tiempo al llegar el paquete de control de transmisión? Explicar ahora por qué el retardo de un paquete nunca puede ser cero.

Al no haber ningún paquete almacenado en cola en ese instante, puesto que la cola inicialmente está vacía, no se puede transmitir nada, por lo que los paquetes que se almacenen en ese instante en la cola se transmitirán en el siguiente *slot* de tiempo (el primero que se haya almacenado, puesto que sólo se transmite la cabeza de la cola). Se entiende ahora que el retardo mínimo de un paquete siempre sea el de un *slot* de tiempo y que el retardo máximo sea el mismo que número de posiciones de memoria tenga la cola. Este suceso se ve claro en el primer *slot* de tiempo y se puede dar en cualquier instante de la simulación, sobre todo si la carga es baja, y por tanto las colas no estarán siempre totalmente llenas.

Cuestión 3

Observar un paquete a lo largo de un slot de tiempo. ¿Qué le ocurre cuando sale de cada módulo?¿Cómo va marcado ese paquete?

Un paquete se construye en el generador de tráfico, dándole un tamaño fijo, indicándole cuál es el tiempo de su creación, un nombre y fijando en uno de los campos su destino. Cuando este paquete llega al *switch*, este lee ese campo para averiguar cuál es el puerto de salida que debe emplear para mandar el paquete a su destino correcto. Una vez lo sabe cambia ese campo por un indicador de paquete ofrecido (identificándose en la simulación por tener un color rojo en la simulación). Cuando la cola recoge ese paquete y tiene que almacenarlo en cola comprueba si quedan posiciones de memoria disponibles (en caso de una cola finita) y en caso afirmativo lo vuelve a marcar como paquete cursado o paquete transmitido correctamente (utilizándose también el color rojo en la simulación para su identificación) y si la cola estuviese llena el identificador sería el de paquete perdido (utilizándose el color negro en la simulación). Todas estas son las posibles marcas que puede tener un paquete a lo largo de la simulación y dependiendo que módulo atraviese.

Cuestión 4

Cambiar el valor de la carga a 0.1 y el de longitud de la cola a 5. Para ello abrir el archivo omnetpp.ini utilizando el comando emacs omnetpp.ini. En la sección donde se muestran los parámetros de un conmutador 4x4 variar la carga y longitud de cola. El tiempo de simulación fijarlo a 345600 slots y en el tiempo de transitorio poner 14400 slots. Arrancar el programa de la misma forma que en la cuestión 1. Hacer una simulación rápida (F7) y cuando acabe el tiempo de simulación llamar a las funciones finish() (en el menú Inspect->call finsih() function), y tomar nota de la probabilidad de pérdida y el retardo medio. Realizar esto para los siguientes valores de carga: 0.2, 0.3 , 0.4, 0.5, 0.7, 0.9 ,0.99; manteniendo fijo el tamaño de buffer y apuntando los resultados de la probabilidad de pérdida y retardo medio para cada caso.

Realizar a continuación una representación gráfica con los datos tomados, una de la prob. de pérdida en función de la carga y otra del retardo medio en función de la carga, sabiendo que la longitud del buffer es 5.

¿Cuál es la razón de que se produzcan estos resultados?

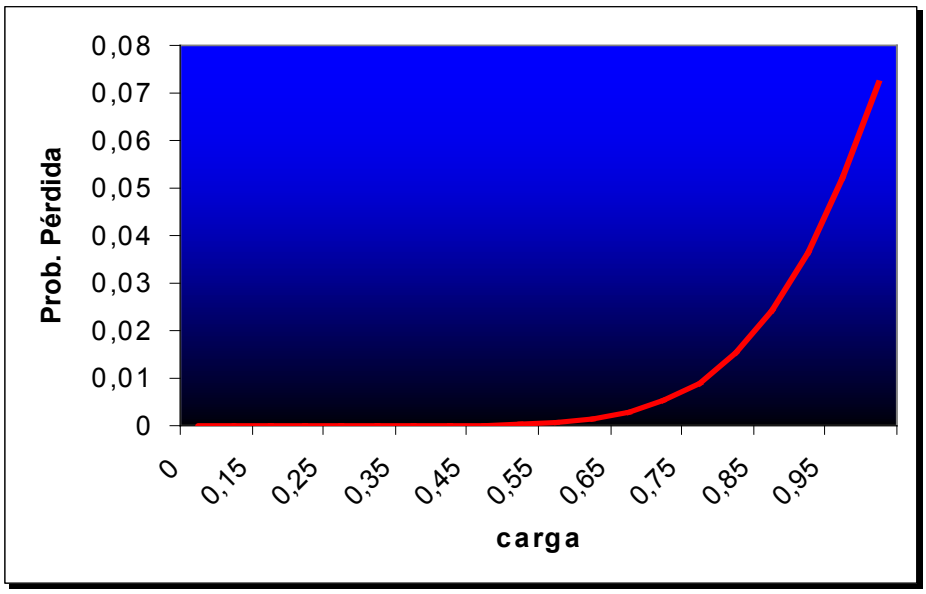

Fig.3.10. Gráfica de la pérdida en función de la carga

Esta gráfica corresponde a la evolución de la pérdida de paquete suponiendo que es la carga de entrada lo que va variando. Si se observa detenidamente, se ve como a partir de una carga de 0.5, esa pérdida se dispara. La exponencial se acentúa cada vez más a partir de esa carga, puesto que ya comienzan a ser tasas de carga bastante altas. Se ve como la tasa máxima de pérdida es de un 8%, lo cual es una pérdida muy alta. Normalmente, cuando se diseña un conmutador, se intentan dimensionar de forma que sus pérdidas sean del orden de 10^{-8}. Hay que recordar que esta simulación se realizó con *buffers* de tamaño 5 posiciones de memoria. Para obtener cotas deseadas de pérdidas deberían utilizarse *buffers* de mayor capacidad.

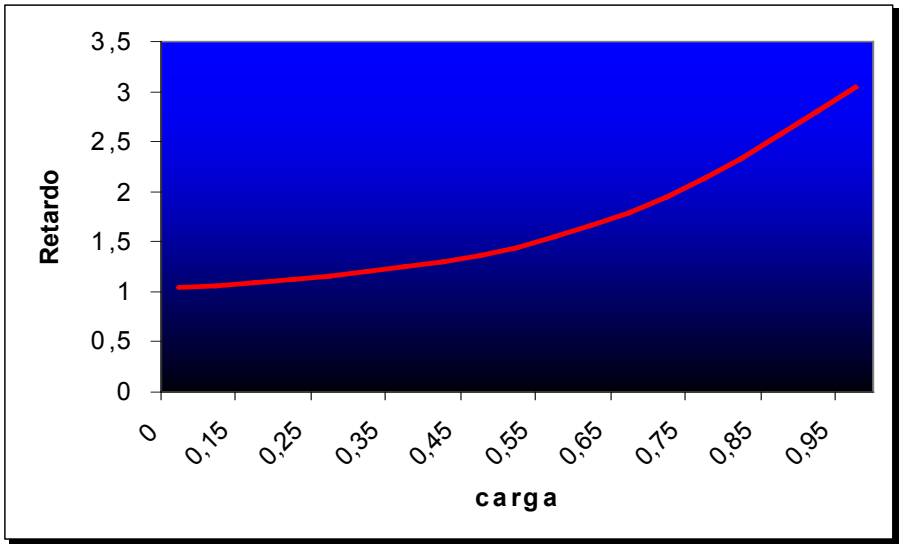

Fig.3.11. Gráfica del Retardo Medio en función de la carga

En esta gráfica, señala el retardo medio de paquete en función de la carga y al contrario de la anterior, el crecimiento del retardo es exponencial, pero poco

acentuada. Apenas si tiene un crecimiento abrupto a partir de una carga de 0.75. Esto es debido a que al haber más carga implica que los paquetes se retarden el máximo posible, puesto que cuando lleguen a la cola estará siempre llena. El máximo tiempo que un paquete podría retardarse en este ejemplo sería 5 unidades de tiempo, puesto que la cola tiene tamaño 5. En media, para una simulación donde tenemos una alta carga el retardo es algo más de 3 unidades de tiempo.

Cuestión 5

Se realizará ahora un experimento similar, pero en lugar de ser la carga la que se varíe, ahora será el tamaño de buffer el que se modificará en cada caso. El valor de carga que daremos será una carga alta: 0.99. Se empezará tomando datos desde un tamaño de buffer igual a 5 y proseguiremos con: 10,20,30,40,50,75.

Ahora, en lugar de representar el retardo medio en función del tamaño del buffer sólo representar la probabilidad de pérdida sufrida ante distintos tamaño de buffer y a una carga de 0.99.

- *Sin haber representado el retardo medio y con lo visto hasta ahora responder a: ¿cómo sería la evolución de una gráfica en la que se representa el retardo medio en función del tamaño del buffer, suponiendo la carga expuesta? Realizar la gráfica del retardo medio en función del tamaño de buffer de forma opcional.*

Con los conocimientos adquiridos a lo largo del capítulo, esta pregunta debe de ser intuitiva. Está claro que el retardo medio irá aumentando en función de la longitud de la cola, bajo estas condiciones de carga. Para colas de tamaño elevado, los paquetes podrán quedar almacenados durante un tiempo mayor que en colas de menor longitud, por lo que el retardo sufrido al llegar también será mayor.

La gráfica se muestra a continuación:

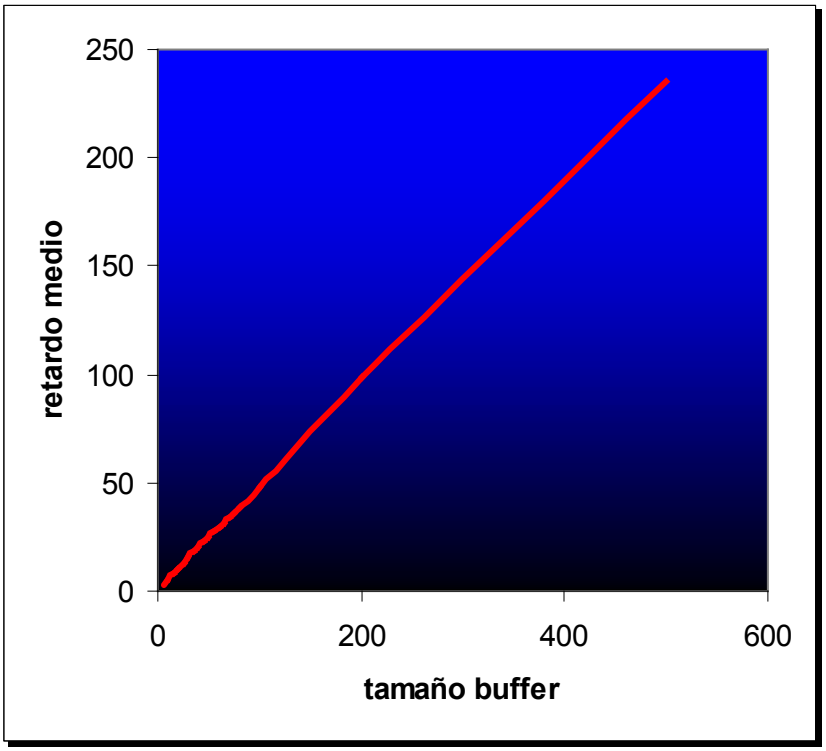

Fig.3.12. Retardo medio en función del tamaño del buffer

Esta gráfica muestra lo comentado anteriormente. Se observa como el retardo medio de paquete sigue un incremento lineal conforme se varía el tamaño de buffer. La explicación es la misma que antes, y es que un paquete podrá retardarse como mucho el número de posiciones de memoria que tenga el buffer y en media, el retardo medio será aproximadamente la longitud media del tamaño de la cola. En la escala de la gráfica aparece desde el punto 0,0 suponiendo que si tenemos una cola que no tiene posiciones de memoria no habrá retardo alguno, puesto que ningún paquete se transmitirá.

- *Una vez realizada la gráfica de la probabilidad de perdida ante distintos tamaños de buffer,¿sería posible tener una red conmutación sin ninguna pérdida con colas de tamaño fijo? Si es posible indíquese cómo y si sería factible.*

La gráfica resultante (Fig.3.13.) acentúa un claro descendimiento al principio y luego esta se estabiliza, en valores cercanos a cero pero nunca este. Esto era un resultado que se prevé: la probabilidad de pérdida nunca será cero puesto que el tamaño del buffer es limitado, aunque este muy cercano a este. Ahora que se sabe esto se puede afirmar, que es imposible el no tener pérdidas. Lo que se puede tener es una probabilidad insignificante pero a costa de emplear grandes memorias. Un caso límite en el que no se encontraría ninguna pérdida con colas de tamaño fijo sería ante una carga de tráfico de entrada bajo. Hay que tener en cuenta que las simulaciones que se han realizado tienen una carga de 0.99.

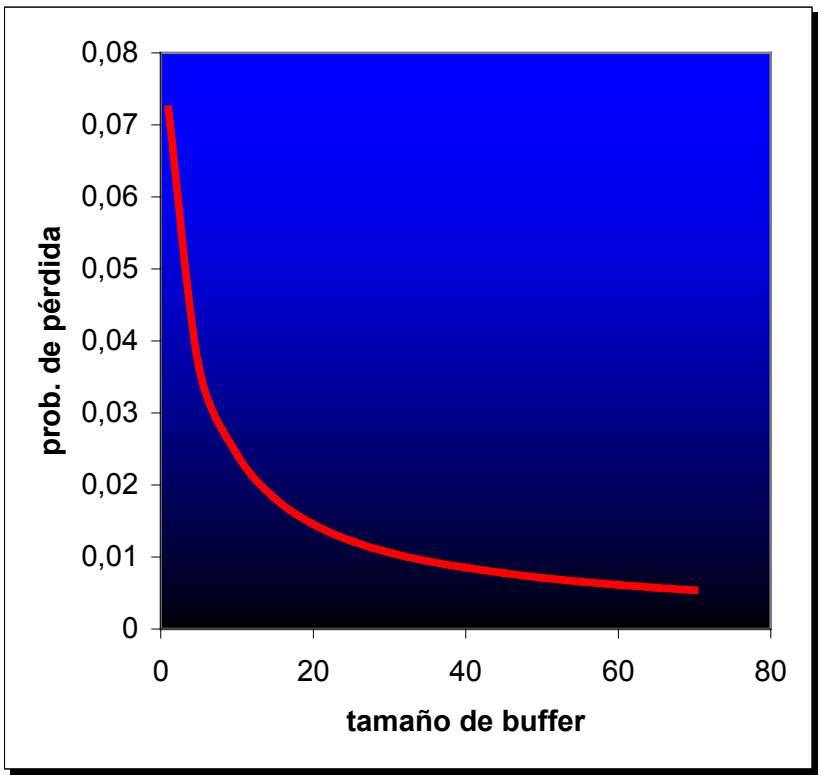

Fig.3.13. Prob. de pérdida en función del tamaño del buffer

- *Qué solución es mejor a la vista de los resultados, teniendo en cuenta que el tráfico de entrada es bastante alto: usar colas de posiciones de memoria limitadas o ilimitadas.*

Cuando se habla de colas de tamaño infinito está claro que es una forma de hablar. No se pueden tener colas de longitud ilimitada, no son realizables físicamente, sólo a nivel de simulación. Lo que si se puede tener son colas, con una cantidad de posiciones de memoria muy alta, consiguiendo así reducir mucho la tasa de pérdidas.

El inconveniente de esto es que, a pesar de tener muy pocas pérdidas, lo que producen este tipo de colas es introducir un mayor retardo de paquete. Por el contrario, si las colas utilizadas tuviesen pocas posiciones de memoria se conseguiría tener poco retardo en los paquetes, pero sin embargo se acentuaría la pérdida de paquetes. Con este tipo de *buffers* se degradaría más la calidad de servicio.

En definitiva, se use el tipo de cola que se use siempre tendrá algún aspecto negativo.

3.4.2 Respuestas a la Parte II

Cuestión 1

La primera cuestión que se presenta es diferenciar cual sería el comportamiento de cada conmutador, sin haber simulado nada, solamente teniendo en cuenta lo dicho hasta ahora, ante la misma carga de entrada y las mismas posiciones de memoria en los puertos de salida. Para comprobar empíricamente esto se realizará la siguiente simulación aplicada a cada topología: carga = 0.8, tamaño de buffer = 5, T_{trans} = 600 slots y T_{sim} = 345600 slots. Analizar cual es la probabilidad de pérdida y retardo medio en cada caso. Es interesante también visualizar cual es la distribución del retardo de paquete de cada conmutador.¿Crees que son coherentes los resultados? Como ejercicio propuesto comprobar los datos de una simulación idéntica pero con una carga = 0.1.

Para comprobar las diferencias y similitudes entre las topologías implementadas se muestran a continuación los resultados de las simulaciones realizadas. Se trata de comprobar la evolución de cada conmutador ante la misma carga de entrada y mismo tamaño de *buffer* (carga = 0.8, tamaño *buffer* = 5):

Conmutador	Prob. Pérdida	Ret. Medio
4x4	0.0152751	2.13243
4x8	5.08244e-5	1.24998
8x4	0.37509	4.5809
8x8	0.022066	2.26541

Un detalle a tener en cuenta a la hora de realizar esta simulación es que los resultados obtenidos no son totalmente relevantes, debido al corto periodo de simulación. Si se quisieran medir cotas de pérdida del orden 10^{-5} habría que simular al menos un tiempo de 10^7, para que el número de paquetes perdidos supere la centena. Esto alargaría mucho el tiempo de la simulación, por lo que esta simulación sirve sólo a nivel orientativo.

También hay que tener en cuenta que en este proyecto no se implementan técnicas para el cálculo de intervalos de confianza de los resultados obtenidos. Lo único que se presentan son resultados puntuales de simulaciones dadas para el contraste de los resultados.

Las conclusiones que se pueden desprender de esta simulación es que para conmutadores de tamaño cuadrado todos tienen unas prestaciones muy similares. Por otro lado, para conmutadores con mayor número de puertos de entrada que de salida es lógico pensar que se tendrá mayor cantidad de pérdida y un retardo medio superior en comparación a un conmutador con mayor número de puertos de salida que de entrada. Esto es lo que muestran los resultados. En la siguiente cuestión se responde al por qué de esto.

Si se comprobase el ejercicio propuesto se observaría que el conmutador 8x4 tendría unos resultados algo más parecidos al de un conmutador cuadrado en estas condiciones por lo que su comportamiento, como era de esperar, está condicionado sobre todo a la carga de entrada.

Para terminar de contrastar las diferencias existentes entre las distintas topologías implementadas se muestran a continuación los histogramas de las distribuciones del retardo (figuras de la 3.14 a la 3.17) de las simulaciones realizadas. Se muestra como el histograma del conmutador 4x4 y el del conmutador 8x8 son casi idénticos por lo explicado anteriormente. Las diferencias se encuentran en los conmutadores no cuadrados.

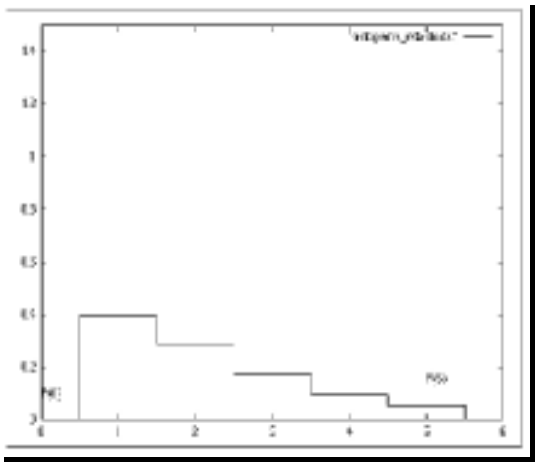

Fig.3.14.Histograma retardo conmutador 4x4

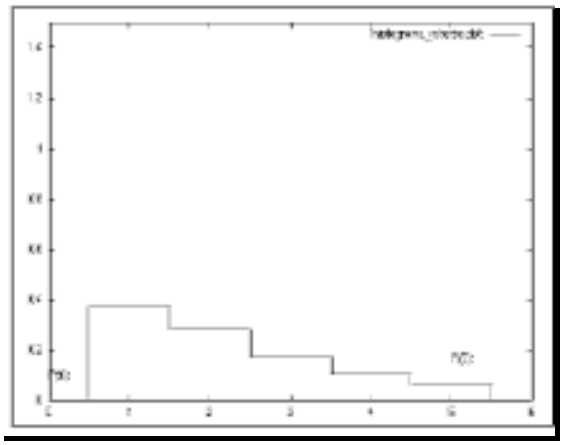

Fig.3.15.histograma retardo conmutador 8x8

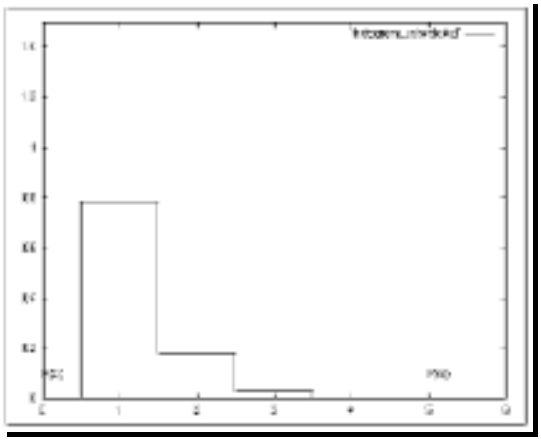

Fig.3.16.Histograma retardo conmutador 4x8

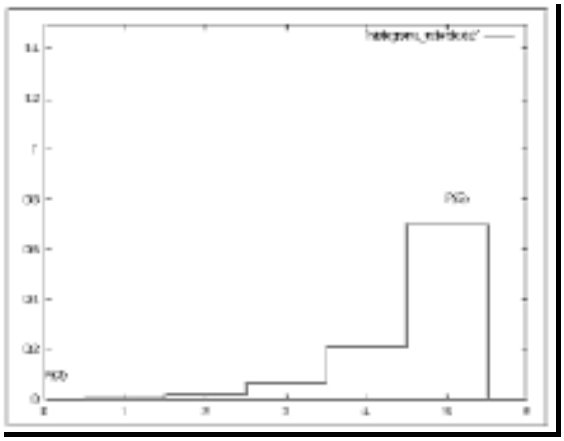

Fig.3.17.Histograma retardo conmutador 8x4

Cuestión 2

Observar la diferencia que existe entre conmutadores de tamaño cuadrado, entendiendo por esto conmutadores del mismo número de puertos de entrada que de salida, y los no cuadrados.

Cuando se usa un conmutador con mayor número de salidas que de entradas (conmutador 4x8) tiene una probabilidad de pérdida insignificante y un retardo inferior al de las topologías cuadradas. Por el contrario, cuando se trata de un conmutador con mayor número de entradas que de salidas (conmutador 8x4) la probabilidad de pérdida se dispara mucho, al igual que el retardo se acentúa. La razón de esto es que la carga por puerto de salida aumenta si el número de puertos de salida disminuye con respecto al número de puertos de entrada. Matemáticamente: carga_entrada*número_puertos_de_entrada = carga_salida*número_puertos_salida. Si por ejemplo, se tuviesen 4 puertos de entrada, con una carga de entrada de 0.5 y se tuvieran 2 puertos de salida. Según la expresión matemática, la carga de los puertos de salida sería de 1, lo que produce los efectos mencionados anteriormente. La solución ante estos casos es incrementar el número de posiciones de memoria.

Cuestión 3

¿Cuál es la topología que mayor pérdida sufre?. Este tipo de conmutador qué nombre recibe.

La explicación es sencilla: un conmutador 4x8, a pesar de tener una carga alta, al tener más salidas que entradas, las colas estarán vacías la mayor parte del tiempo y por tanto un paquete cuando llegue en un *slot* está casi segura su transmisión en el siguiente, por el contrario, en un conmutador 8x4, al tener tanta carga de entrada y tan pocos puertos de salida, los paquetes, cuando lleguen a las colas de salida, estas colas estarán la mayor parte del tiempo llenas, por lo que tendrán que esperar el máximo tiempo que se puede: tantos *slot* como posiciones de memoria tenga la cola.

Este tipo de conmutadores se denominan *concentradores.*

Capítulo 4
SISTEMAS EN 3 ETAPAS DE CONMUTACIÓN DE PAQUETES

4.1 Introducción

Lo que se presentará en este capítulo serán arquitecturas basadas en la interconexión de elementos de conmutación de pequeño tamaño para formar una red de conmutación de mayores dimensiones [9]. Este tipo de redes se denominan redes de conmutación multietapa. Una red de una etapa sería la formada por un solo dispositivo de conmutación. Cuando a ese dispositivo le conectamos a su salida otro, de características similares, se convierte en una red de dos etapas, que ya se considera multietapa. Las redes que en este capítulo se presentarán serán redes multietapa, de tres etapas con estructura red de *Clos*, utilizando para la construcción de estas redes los elementos de conmutación que se han estudiado en el capítulo anterior.

La razón para emplear estas redes es que es tecnológicamente muy complicado hacer conmutadores de alto número de puertos (por ejemplo por que la velocidad de acceso a la que tienen que ir las memorias en conmutadores con colas a la salida es proporcional al número de puertos de entrada) . Lo que se hace para arreglar esto es construir la red como interconexión de conmutadores de menor tamaño. Lo más habitual es emplear topologías de interconexión en red de Clos [9].

4.1.1 Red de Clos

Una red de Clos consiste en una arquitectura de conmutación con el mismo número (*N*) de entradas que de salidas.

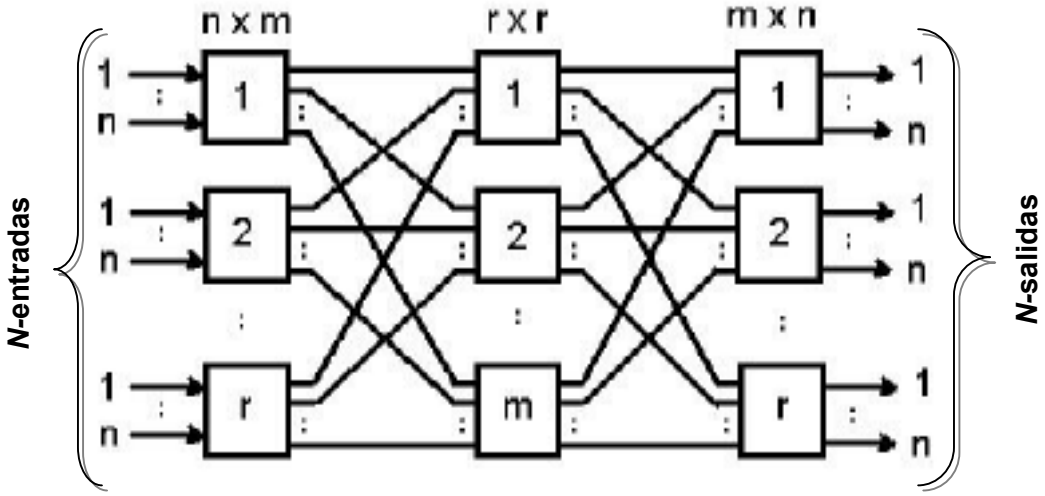

Fig. 4.1 Red de Clos con N entradas

La red está formada por elementos de conmutación, dispuestos según tres etapas tal y como se muestra en la Fig. 4.1. La primera y tercera etapa se componen de *r* elementos de tamaño nxm y mxn respectivamente. La etapa intermedia está compuesta de *m* elementos de tamaño rxr. Además de estas características, una red de Clos puede diseñarse de forma que cumpla la condición de *Nonblocking* (que se tratará en el apartado 4.1.2), aunque a la hora de desarrollar las arquitecturas de este capítulo no influye, pues esta condición se da para redes que no tienen memorias internas.

Todos los elementos de una misma etapa tienen las mismas características, es decir, mismo número puertos de entrada, mismo número de puertos de salida y mismo número de posiciones de memoria.

Por último, la red de Clos tiene aplicación en las redes de conmutación telefónica por conmutación de circuitos. Sin embargo, el uso en el entorno de conmutación de paquetes y con elementos de conmutación basados en memoria compartida requiere un estudio especial.

4.1.2 Condición de Clos.

También denominada condición de *Nonblocking,* y es la condición para la que una red multietapa no presenta bloqueo en sentido estricto. Si se tienen que en una etapa de entrada de la red, de tamaño m*k, y se han realizado todas las conexiones excepto una, se puede solicitar la conexión por *n-1.* Análogamente, también se puede tener una etapa de salida con *n-1* conexiones realizadas. Por tanto, en el peor de los casos posibles, que es que ninguna de las *n-1* conexiones de la matriz de entrada sea con alguna de las *N-1* líneas de la matriz de salida, se ocuparán 2*(n-1) etapas de distribución, o número de módulos de segunda etapa.

Entonces, para que no haya posibilidad de bloqueo, el número de etapas de distribución debería ser mayor o igual a 2n-1, que es la condición de Clos.

Esta condición sólo es aplicable en redes donde las distintas etapas no tienen memorias internas. Sin embargo los dispositivos de etapa intermedia que se han empleado son conmutadores con colas a la salida, estudiados en el capítulo anterior, así que esta condición no influye a la hora de diseñar las arquitecturas, pues no es posible que se de bloqueo interno en la red.

4.1.3 Política de Encaminamiento

Las arquitecturas que se han implementado son redes de conmutación en tres etapas, como se ha mencionado antes.

Cuando un paquete intenta dirigirse desde un origen hacia un destino, el camino que sigue a través de la red se basa en la información de encaminamiento que lleva. Pero se presenta un problema cuando debe dirigirse a la etapa intermedia, y es que el paquete sólo contiene información del módulo de tercera etapa al que se dirige, y el puerto de salida dentro de este, pero no dice a través de qué nodo de segunda etapa llegar a ese destino.

En este caso se dan dos posibles alternativas: utilizar un algoritmo de encaminamiento aleatorio o distribuido.

Encaminamiento aleatorio

Los paquetes, cuando llegan al módulo de primera etapa, se encaminan de forma aleatoria a un módulo de segunda etapa.

Encaminamiento secuencial

Este algoritmo consiste en encaminar los paquetes recibidos en el módulo de primera etapa de forma secuencial a los módulos de segunda etapa. Esto es que el primer paquete recibido saldrá por el primer puerto de salida del módulo, el segundo paquete por el segundo puerto y así hasta completar todos los puertos de salida. Si se completan todos los puerto y todavía quedan paquetes se vuelve a comenzar por el primer puerto, y así hasta encaminar todos los paquetes recibidos.

4.2 Descripción del Entorno

Las redes de conmutación aquí implementadas están desarrolladas con la herramienta de simulación Omnet++ [5] [6], al igual que en el capítulo anterior. Como se dijo en el capítulo pasado, Omnet++ es una herramienta de simulación orientada a objetos, basada en eventos, usando lenguaje C++ [11] y anidamiento jerárquico de los módulos creados.

Como todos estos conceptos ya han sido explicados en capítulos anteriores, lo que se realizará a continuación es aplicar estos conceptos a las implementaciones realizadas.

A diferencia del capítulo anterior, en el que todos los objetos creados eran módulos simples, ahora se tienen tanto módulos simples como módulos compuestos (o *compound modules* como se denotaría en Omnet). Esto hace que el nivel de anidamiento en estas implementaciones sea más complejo que en el capítulo anterior.

En las implementaciones de las arquitecturas nos encontraremos unas “cajas negras” que representarán a los elementos de conmutación de cada etapa, pero que a su vez estos están diseñados como conmutadores con colas a la salida, que es la implementación que se ha desarrollado en el capítulo anterior. Esto se explicará más detalladamente en la sección 4.2.1, donde se explicará el comportamiento, funcionamiento y forma de implementación de cada módulo empleado para el desarrollo de las redes implementadas.

En cuanto a los eventos, estos son sucesos que gestionan la simulación que se lleve a cabo y que modela el comportamiento de la arquitectura implementada. Estos eventos los usa Omnet para diferenciar en cada instante de tiempo qué módulo es el que se activa y para qué.

Por último decir que el lenguaje C++ aquí se usa para diseñar los módulos simples implementados, al igual que en el capítulo anterior.

4.2.1 Arquitectura Omnet para el Desarrollo de Una Red de Conmutación de Tres Etapas

Para implementar las redes que se han desarrollado, lo primero es tener claro cuales van a ser los módulos simples que ayuden a diseñar toda las arquitecturas. Estos son los mismos que en el capítulo anterior: módulo control, generadores de tráfico, *switchs,* colas , sumideros (tanto estadísticos como no estadísticos). Estos se desarrollarán primero en C++ [11] y después se emplearán para construir las arquitecturas. Decir que estas redes se construirán con la herramienta de Omnet *gned.*

En la implementación se desarrolla también una clase nueva que hereda de *cMessage* con nuevos campos en la cabecera para poder tener información de lo que le va sucediendo a cada paquete cuando atraviesa cada etapa de conmutación y poder realizar así las estadísticas finales.

A continuación se cada módulo, pero se recomienda mirar el código de implementación realizado para que quede todo más claro.

4.2.1.1 Módulo “Control”

Módulo simple encargado de la gestión de los principales eventos de la simulación en cada *slot* de tiempo. En definitiva, indica a cada módulo cuando debe realizar “su trabajo”. Como se observa, tiene la misma función que el módulo de control del capítulo anterior. La diferencia viene dada por los eventos generados que no son exactamente los mismos. Estos son:

- Generación del tráfico de entrada.
- Conmutación de los paquetes en la primera etapa.
- Conmutación de los paquetes en la segunda etapa.
- Conmutación de los paquetes en la tercera etapa.

Al igual que en el capítulo anterior, el evento que no gestiona el módulo de control es la recolección de los paquetes por los sumideros.

El funcionamiento detallado a lo largo de un *slot* temporal es el siguiente (suponiendo el primer *slot*): en el instante *t*=0 el módulo manda el paquete de control al generador, y simula la generación de los paquetes de entrada a la red. En *t*=0.2 se manda la señal de control a los módulos de la primera, y estos empiezan a conmutar los paquetes que tienen en los puertos de entrada. En *t*=0.4 se manda la señal a los módulos de la segunda etapa y realizan la misma función que los módulos de la primera etapa, y en *t*=0.6 se manda la señal de control a los módulos de la tercera etapa, realizando estos el mismo trabajo que los conmutadores de las etapas anteriores. Después el módulo espera un tiempo *t*=0.4 *ticks* de reloj y comienza de nuevo el ciclo.

El correcto funcionamiento de este módulo es vital para que la simulación transcurra con total normalidad ya que si no mandase correctamente una de las señales de control, los módulos podrían quedar bloqueados.

4.2.1.2 Módulo “Generador”

También es un módulo simple, al igual que el módulo de control. Este módulo se encarga de generar todo el tráfico de entrada a la red, siguiendo un patrón de entrada de *Bernouilli* de carga *p*. La función es la misma que el módulo generador del capítulo anterior salvo que en la implementación de los elementos de conmutación del capítulo anterior a cada entrada se le conectada un generador y ahora el generador es único para todos los puertos de entrada. Tal vez el nombre más apropiado de este módulo sería el de *multigenerador*.

En cuanto al funcionamiento de este generador se refiere, se detalla seguidamente.

Se activa mediante señal de control que recibe por la única línea de entrada que tiene. Mientras no la recibe no realiza ninguna función, y una vez llega la señal entra en un bucle donde podrá generar hasta *N* paquetes, uno por cada puerto de entrada que haya en la red.

Para cada iteración se obtiene un número aleatorio entre 0 y 1 y si el valor obtenido es menor que el valor de la carga se generará un paquete para ese puerto de entrada. En caso de no generar paquete continuará con la siguiente iteración, pero si tuviese que generar un paquete, este sería de la nueva clase creada y que hereda de *cMessage.* Este paquete en su creación se le asigna un nombre y se le introduce el tiempo de creación. Además se le incluye el módulo de tercera etapa al que va

destinado y el puerto de salida de ese módulo. En definitiva, se le indica el destino que tiene. Esto lo realiza de forma aleatoria.

Una vez que termina todas las iteraciones, y por tanto recorre todos los puertos de entrada de la red, queda a la espera de un nuevo paquete de control para volver a realizar la misma tarea.

4.2.1.3 Módulos "*Switch*"

Estos siguen siendo módulos simples al igual que los anteriores. En este caso se encuentran distintos tipos de *switch* dependiendo de la etapa a la que esté destinado. Esto es que los dispositivos de conmutación de cada una misma etapa serán iguales entre sí pero con alguna diferencia a los de las restantes etapas.

La función final es la misma para todos: encaminar los paquetes que se reciben por los puertos de entrada a los puertos de salida, en función de la información de encaminamiento. La diferencia es que, en cada etapa, esa información de encaminamiento se obtiene de forma distinta.

A continuación se presentan los distintos tipos de *switch* implementados y de donde obtienen la información de encaminamiento:

4.2.1.3.1 *Switch* de primera etapa

Estos módulos se emplean el los módulos de conmutación de primera etapa y tienen el siguiente comportamiento. Están constantemente recibiendo paquetes por los puertos de entrada y almacenándolos y una cola auxiliar, hasta que reciben la señal de control y comienzan a extraer, por orden de llegada, los paquetes que se encuentran en la cola auxiliar. El encaminamiento de los paquetes hacia los módulos de segunda etapa puede seguir una política de encaminamiento aleatoria o secuencial (ambas explicadas en la sección 4.1.3).

Una vez que termina de encaminar todos los paquetes manda una señal de *FINISH* a las colas que tiene conectadas a sus puertos de salida, para que estas puedan comenzar su trabajo.

Terminado todo este proceso quedan nuevamente a la de la señal de control, volviendo a almacenar los paquetes en la cola auxiliar.

4.2.1.3.2 *Switch* de segunda etapa

Se emplean en los módulos de conmutación de segunda etapa y su funcionamiento es similar a los *switch* de primera etapa.

Están almacenando todos los paquetes que llegan a los distintos puertos de entrada esperando recibir la señal de control. Una vez recibida comienzan a extraer los paquetes de la cola auxiliar en el mismo orden de llegada y dirigiéndolos a los puertos de salida correspondientes, según la información de encaminamiento. Esta información se obtiene de la siguiente manera en esta etapa: los paquetes tienen un campo donde indican cual es el módulo de tercera etapa al que van dirigidos. Como un módulo de segunda etapa está conectado a cualquier módulo de tercera etapa, se manda el paquete al puerto de salida correspondiente, que le comunicará con el módulo de tercera etapa al que va dirigido.

Cuando se encaminan todos los paquetes se vuelve a mandar una señal de *FINISH* a las colas que hay en los puertos de salida para que comiencen su trabajo.

4.2.1.3.3 *Switch* de tercera etapa

Este último tipo de *switch* se encuentra en los módulos de conmutación de tercera etapa. El funcionamiento el mismo que los *switch* descritos anteriormente: almacenan los paquetes que llegan a los puertos de entrada en una cola auxiliar. Una vez llega la señal de control los paquetes de esta cola se extraen en el mismo orden de llegada al módulo de conmutación.

Esta vez, la información de encaminamiento se extrae de un campo de la cabecera del paquete, en el que se indica cual es el puerto de salida al que va dirigido ese paquete. Cuando se conoce ese puerto el paquete es encaminado por el switch.

Al finalizar de encaminar todos los paquetes el *switch* vuelve a mandar una señal de *FINISH* a cada una de las colas que se encuentran en cada puerto de salida, para que estas comiencen a realizar su trabajo.

El *switch* queda a la espera de la nueva señal de control para volver a realizar el mismo proceso.

4.2.1.4 Módulos "*FIFO*"

Al igual que los módulos *switch*, en las implementaciones realizadas se encuentran tres tipos de colas, dependiendo de la etapa que se analice. En estos módulos, lo único que los diferencia entre sí es el nombre y una línea de código que sirve para marcar en el paquete la etapa donde fue eliminado, en caso de pérdida. De esta forma, cuando el paquete llegue al sumidero, se podrá saber en que etapa intermedia el paquete fue eliminado. El resto de código es igual en todas las colas. Es por esta razón que se describirá exhaustivamente la cola de primera etapa y las colas de segunda y tercera etapa se mencionarán por encima, puesto que son prácticamente iguales.

4.2.1.4.1 Colas de primera etapa

Como su nombre indica, este tipo de colas se encuentran en los dispositivos de conmutación de primera etapa. En un dispositivo, se localizarán tantas colas como puertos de salida tenga, puesto que cada cola está conectada a un puerto de salida.

En cuanto al funcionamiento se refiere, estas tienen un comportamiento similar a los módulos *fifo* detallados en el capítulo anterior: Una cola almacena todos los paquetes recibidos a su entrada en una cola auxiliar. Cuando llega el paquete de control comienzan "su trabajo".

La única diferencia con las colas implementadas en el capítulo anterior surge en este punto. El paquete de control no lo manda el módulo de control sino que lo manda el *switch* al que está conectado.

Una vez se recibe ese paquete, primero se lee la cabeza de la cola principal, y si hay algún paquete se transmite. Si no lo hay, se pasan todos los paquetes que hay en la cola auxiliar, en el mismo orden que llegaron, a la cola principal. Aquí se pueden dar dos casos, dependiendo del tipo de cola que se esté utilizando. Si se utiliza una cola de longitud infinita, los paquetes pasan directamente de la cola auxiliar a la principal sin ningún tipo de comprobación.

En caso de que la cola utilizada tenga posiciones de memoria finita, cada vez que se extrae un paquete de la cola auxiliar se comprueba la ocupación de la cola y si esta tiene posiciones vacías, el paquete se inserta al final. En caso de estar la cola llena, el paquete se marca como paquete eliminado. Es aquí donde aparece la diferencia que hace que las colas implementadas para una etapa sean distintas que las colas utilizadas en el resto de etapas: cuando un paquete se marca como

eliminado lo que se hace es utilizar un *flag* de la cabecera del paquete y establecerlo a *true*. Este *flag* indica en que etapa intermedia de la red ha sido el paquete eliminado. Además, en la cola, una vez se marca como eliminado, lo que se hace es cambiar la clase del paquete, que hasta ese instante es de *paq_offer* (distinguido por el color rojo), y una vez se marca como eliminado la clase que se le asigna al paquete es de *lost_paq* (diferenciado por el color negro).

4.2.1.4.2 Colas de segunda etapa

Estas colas tienen el mismo funcionamiento que las colas de primera etapa. Lo único que se mencionará en este apartado es la diferencia entre estas colas y las de primera etapa, que surge cuando se marca un paquete como eliminado en esta etapa. Cuando se da este caso se activa a *true* el *flag* del paquete que indica que un paquete se ha perdido en la segunda etapa.

4.2.1.4.3 Colas de tercera etapa

En este apartado se volverá a repetir lo dicho en el apartado 4.2.1.4.2. Lo único que diferencia a estas colas de las colas de tercera etapa es el *flag* que se activa cuando se pierde un paquete en esta etapa: ese campo indica que el paquete se pierde en la tercera etapa. El funcionamiento es idéntico.

4.2.1.5 Módulos Sumideros

En las redes implementadas se encuentran sumideros que se encargan de la recolección de los paquetes de la simulación.

Se implementan dos tipos de sumideros, uno no estadístico y otro estadístico, y ambos tienen el mismo código que los implementados en el capítulo anterior. Seguidamente se describen los dos tipos:

Sumidero no estadístico

Módulo simple que lo único que hace es recibir paquetes y eliminarlos sin ningún tipo de acciones adicionales. No necesita ninguna señal de control pues este módulo se activa cada vez que le llega un paquete, debido a que la interfaz que implementa es la *handleMessage()*.. En el diseño del conmutador, todos los sumideros son no estadísticos salvo el que se encuentra en el primer puerto de salida del primer módulo de conmutación de la tercera etapa.

Sumidero estadístico

También es un módulo simple, encargado de la eliminación de los paquetes que le llegan pero además se encarga de realizar las estadísticas de la simulación.

Este módulo se utiliza a la salida de un único puerto: el primer puerto del primer módulo de conmutación de tercera etapa.

El funcionamiento es similar al del sumidero no estadístico: recibe paquetes y antes de eliminarlos, comprueba de que clase son. Si son de tipo *ok_paq* (paquete cursado) se le extraen una serie de parámetros al paquete y si por el contrario es de tipo *lost_paq* (paquete perdido), se le extraen otra serie de parámetros. Cuando la simulación termina, el módulo procesa toda la información acumulada y muestra los resultados de esa simulación. Además genera un *script* para *gnuplot* que se utilizará para representar una serie de histogramas de la simulación realizada. Añadir por último al funcionamiento, que el módulo cuenta con un tiempo de transitorio, que es el tiempo que tarda el conmutador en adquirir estabilidad, y cuando este se cumple, las variables que se emplean para la recolección de datos se establecen a cero nuevamente.

4.2.1.6 Módulos Compuestos

Corresponden a los elementos de conmutación de primera, segunda y tercera etapa. Su composición consta de un *switch* (que dependiendo de que etapa se quiera implementar será de un tipo específico) y, conectadas a los puertos de salida de este *switch,* colas de la etapa que le corresponda. Con esto lo que se consigue es tener implementado un conmutador con colas a la salida, visto en el capítulo anterior.

El diseño de este tipo de módulo es fácil. Para implementarlo únicamente hace falta la herramienta *gned.* A partir de esta se pueden diseñar declarando un nuevo *compund module*, y una vez establecido se le agregan los elementos necesarios para construirlo, que serán los módulos simples *switch* y colas. Una vez agregados estos módulos se establecen las conexiones y se especifican los puertos de entrada, salida, parámetros, etc de los que consta el módulo.

En las arquitecturas desarrolladas se implementas tres tipos de módulos compuestos (o complejos): módulos de conmutación de primera etapa denominados *k1* , y que están compuestos por *switch* y colas de primera etapa; módulos de conmutación de segunda etapa denominados *k2,* compuestos por *switch* y colas de segunda etapa; y módulos de conmutación de tercera etapa denominados *k3,* compuestos por *switch* y colas de tercera etapa.

El aspecto gráfico de estos módulos es el mismo para todas las etapas: se ven como “cajas negras” (Fig. 4.2) pero se puede mirar su interior si se hace *clic* en uno de ellos. Si se mirase el interior se observaría como su arquitectura es idéntica (Fig. 4.3).

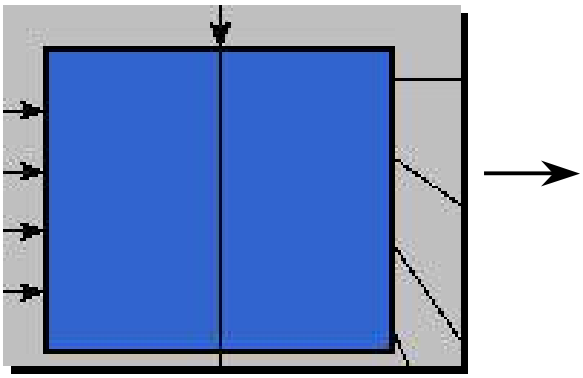

Fig.4.2 Diseño de un módulo de conmutación

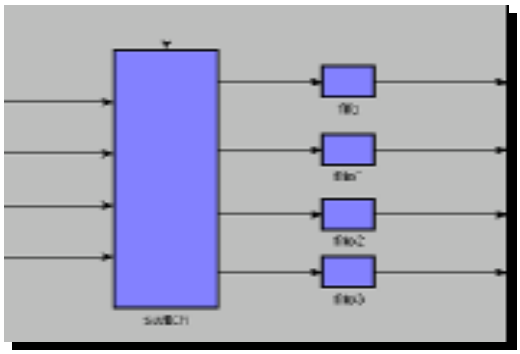

Fig.4.3 Interior de un módulo de conmutación

4.2.1.7 Diseño Gráfico de la Red

En este apartado se mostrará el diseño gráfico de una de las redes implementadas, en concreto será una red tres etapas, con cuatro módulos de conmutación en la primera etapa, cuatro módulos en la segunda y cuatro módulos en la tercera etapa (Fig.4.4). Los módulos de primera etapa tienen cuatro puertos de entrada cada uno y los módulos de tercera etapa tienen cuatro puertos de salida cada uno. El resto de las redes tienen la misma estructura salvo que tienen distinto número de módulos de segunda etapa.

El diseño se realiza con la herramienta *gned* proporcionada por Omnet.

4.2.1.8 Paquete “*Packet.msg*”

Este paquete se crea para realizar la simulación. Esta clase hereda todas las propiedades de la clase *cMessage* y aparte se le añaden una serie de campos que se utilizarán a lo largo de la simulación.

Los campos de este nuevo paquete son:

- SetArrivalK1(), SetArrivalK2(), SetArrivalK3().- Establece el tiempo de llegada a la primera, segunda y tercera etapa respectivamente.
- GetArrivalK1(), GetArrivalK2(), GetArrivalK3().- Devuelve el tiempo de llegada del paquete a la primera, segunda y tercera etapa respectivamente.
- SetModuleK3(), SetPortDest() .- Establecen el módulo de salida del paquete (módulo de 3ª etapa) y puerto de destino en este módulo, respectivamente.

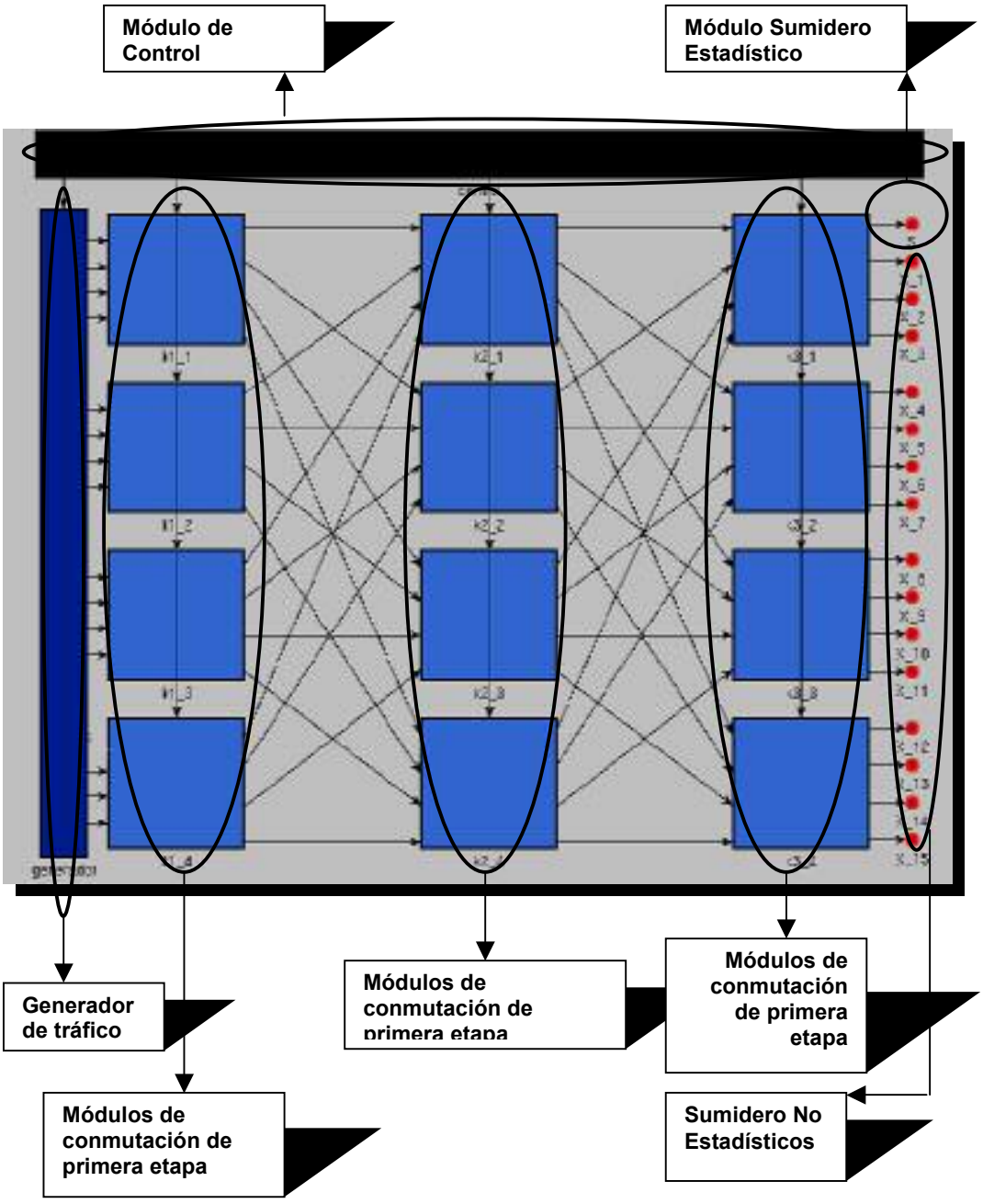

Fig.4.4 Diseño de un red de conmutación tres etapas

- GetModuleK3(), GetPortDest() .- Devuelven el módulo de 3ª etapa al que va destinado el paquete y puerto de este módulo, respectivamente.
- SetLostInK1(), SetLostInK2(), SetLostInK3().- Establecen si un paquete se ha perdido en la 1ª, 2ª o 3ª etapa respectivamente. Cuando un paquete lo analiza el sumidero, solo uno de estos *flags* puede estar activado.
- GetLostInK1(), GetLostInK2(), GetLostInK3().- Devuelve si un paquete se ha perdido en la 1ª, 2ª o 3ª etapa, respectivamente.

4.2.2 Parámetros de Entrada

A la hora de realizar cualquier simulación en Omnet, previamente habrá que dar valores a una serie de parámetros de entrada definidos en el diseño de la arquitectura. En el conmutador implementado encontramos una serie de parámetros que modelarán la simulación y que usarán los módulos descritos anteriormente.

Para asignar los valores a los parámetros se hará desde el archivo de configuración de Omnet (*omnetpp.ini*). Esto facilita mucho la introducción de parámetros puesto que no hay que modificar código y por tanto no hay que compilar cada vez que cambiemos los valores de estos parámetros.

Los parámetros de entrada que utilizan las arquitecturas implementadas se detallan a continuación.

Tipo de patrón de entrada

Este parámetro lo usará el módulo generador. La función de este parámetro es la de indicar que tipo de distribución de tráfico de entrada en la red queremos para nuestra simulación. Sólo está implementada una distribución de *Bernouilli* de carga *p*. En caso de seleccionar otro tipo de tráfico de entrada nos aparecerá por pantalla el siguiente mensaje: "*tipo de tráfico no implementado*". El valor que esa variable tiene que tener para elegir un tráfico de *Bernouilli* es 1. Para cualquier otro valor de la variable no se efectuaría nada.

Carga

Este se utilizará también en el módulo generador. Sirve para simular el patrón de tráfico de entrada de *Bernouilli* donde este parámetro indicará la carga *p* a la que se trabajará. El rango de valores de este parámetro es este intervalo *[0,1]*. En el caso de asignar cero, nunca se generará tráfico en la simulación; y si por el contrario, ponemos la carga igual a uno, se generará un paquete en cada ciclo de reloj.

Tipo de búffer de primera etapa

Parámetro aplicable a las colas de los módulos de conmutación de primera etapa para indicar si se trabaja con colas de tamaño infinito o con colas de longitud finita. El valor para indicar colas de tamaño infinito es –1 y si se quiere colas de tamaño finito hay que poner 1.

Longitud de búffer de primera etapa

Parámetro que tienen efecto en caso de que se trabaje con colas de tamaño finito. Este parámetro indica el número de posiciones de memoria que tienen las colas de los conmutadores de primera etapa. El rango de valores es el siguiente: [0 , ∞).

Tipo de búffer de segunda etapa

Parámetro aplicable a las colas de los módulos de conmutación de segunda etapa para indicar si se trabaja con colas de tamaño infinito o con colas de longitud finita. El valor para indicar colas de tamaño infinito es –1 y si se quiere colas de tamaño finito hay que poner 1.

Longitud de búffer de segunda etapa

Parámetro que tienen efecto en caso de que se trabaje con colas de tamaño finito. Este parámetro indica el número de posiciones de memoria que tienen las colas de los conmutadores de segunda etapa. El rango de valores es el siguiente: [0 , ∞).

Tipo de búffer de tercera etapa

Parámetro aplicable a las colas de los módulos de conmutación de tercera etapa para indicar si se trabaja con colas de tamaño infinito o con colas de longitud finita. El valor para indicar colas de tamaño infinito es –1 y si se quiere colas de tamaño finito hay que poner 1.

Longitud de búffer de tercera etapa

Parámetro que tienen efecto en caso de que se trabaje con colas de tamaño finito. Este parámetro indica el número de posiciones de memoria que tienen las colas de los conmutadores de tercera etapa. El rango de valores es el siguiente: [0 , ∞).

Tipo de algoritmo de selección de etapa intermedia

Con este parámetro se le indica a la simulación que algoritmo de selección de etapa intermedia se desea (de los explicados en el apartado 4.1.3). Se recuerda al lector que es posible elegir entre un encaminamiento aleatorio o secuencial.

En las arquitecturas implementadas, para seleccionar el algoritmo aleatorio hay que asignar a la variable el valor 1 y si por el contrario, queremos realizar la simulación con encaminamiento secuencial hay que asignar el valor 2.

Tiempo de transitorio

Este parámetro es usado por el sumidero estadístico. Su función es decirle al sumidero estadístico el tiempo a partir del cual los datos que está recogiendo para realizar las estadísticas son válidos. Para realizar esta función el sumidero programa un paquete para él mismo tal que se recibirá justo en el tiempo de transitorio. Cuando lo recibe y comprueba que es el paquete que programó lo que hace es *resetar* todas las variables que hasta ese tiempo estaba utilizando para acumular información y poder realizar luego los resultados de la simulación. Los valores que se le pueden dar son [0 , ∞).

Tiempo de simulación

Para Omnet los *ticks* de reloj tienen duración un segundo, la interpretación que se le dará aquí será de *slots* temporales de duración una unidad de tiempo, sin especificar cuanto tiempo real dura ese *slot.* El tiempo de simulación es un parámetro general de Omnet y lo utiliza para detener la simulación en un instante determinado. Cuando se quiera pasar un valor, se le pasará la cantidad de instantes temporales que se quiera simular. Puede darse que no se le pase ningún valor, en cuyo caso la simulación no terminará hasta que pulsemos *stop* dentro de la simulación.

4.2.3 Parámetros de Salida

Como parámetros de salida se entenderán los resultados obtenidos al realizar una simulación. Estos resultados son obtenidos en el módulo sumidero estadístico, y servirán para mostrar la calidad del conmutador evaluado ante distintos tamaños de *búffer* en las colas de los módulos de conmutación de las distintas etapas y distinta carga de entrada.

Todos los parámetros de salida se muestran a continuación.

Número de paquetes recibidos

Indica cuantos paquetes se procesan el sumidero, tanto paquetes cursados como paquetes perdidos.

Número de paquetes cursados

Esta variable contendrá la cantidad de paquetes que han llegado a cursarse en la red. Estos paquetes son paquetes de clase *ok_paq* (además se distinguen por el color rojo en la simulación).

Número de paquetes perdidos

Con este parámetro indicamos cuantos paquetes se perdieron a la hora de atravesar la red. Para discernir estos paquetes de los cursados, estos van marcados de la clase *lost_paq* (se distinguen dentro de la simulación por ser paquetes de color negro).

Probabilidad de pérdida de paquete

Indica, tanto por uno, la probabilidad de pérdida total que se produce en la simulación realizada, ante los parámetro de entrada establecidos.

Probabilidad de que si un paquete se pierde sea en la 1ª etapa

Este parámetro indica cual es la probabilidad de que si un paquete se ha perdido, esta pérdida haya sido en la primera etapa. Para hallar la probabilidad de pérdida de la primera etapa habrá que multiplicar la probabilidad de pérdida de paquete por este resultado.

Probabilidad de que si un paquete se pierde sea en la 2ª etapa

Este parámetro indica cual es la probabilidad de que si un paquete se ha perdido, esta pérdida haya sido en la segunda etapa. Para hallar la probabilidad de pérdida de la segunda etapa habrá que multiplicar la probabilidad de pérdida de paquete por este resultado.

Probabilidad de que si un paquete se pierde sea en la 3ª etapa

Este parámetro indica cual es la probabilidad de que si un paquete se ha perdido, esta pérdida haya sido en la tercera etapa. Para hallar la probabilidad de pérdida de la tercera etapa habrá que multiplicar la probabilidad de pérdida de paquete por este resultado.

Retardo Medio de paquete

Se muestra con este parámetro el retardo medio total de paquete sufrido en la simulación ante las condiciones de entrada establecidas.

Retardo Medio de paquete en la primera etapa

Parámetro que indica el retardo medio que sufre un paquete al atravesar la primera etapa.

Retardo medio de paquete en la segunda etapa

Parámetro que indica el retardo medio que sufre un paquete al atravesar la segunda etapa.

Retardo medio de paquete en la tercera etapa

Parámetro que indica el retardo medio que sufre un paquete al atravesar la tercera etapa.

Distribución del retardo de paquete

Se muestran las distribuciones de retardo de paquete total, el producido en la primera etapa, el producido en la segunda y el producido en la tercera.

Histograma de la distribución del retardo de paquete

Con los datos tomados en las distintas distribuciones de retardo, y gracias a un *script* generado por la función *finish()* del módulo sumidero estadístico para *gnuplot*, se muestran tantos histogramas como distribuciones de retardo calculadas, es decir, se mostrará el histograma del retardo total de paquete, el histograma del retardo de paquete en la primera etapa, el histograma de retardo de paquete de la segunda etapa y el histograma de retardo de paquete de la tercera etapa. Todos se visualizan en un mismo archivo *.eps* tal y como se muestra en la figura 4.5.

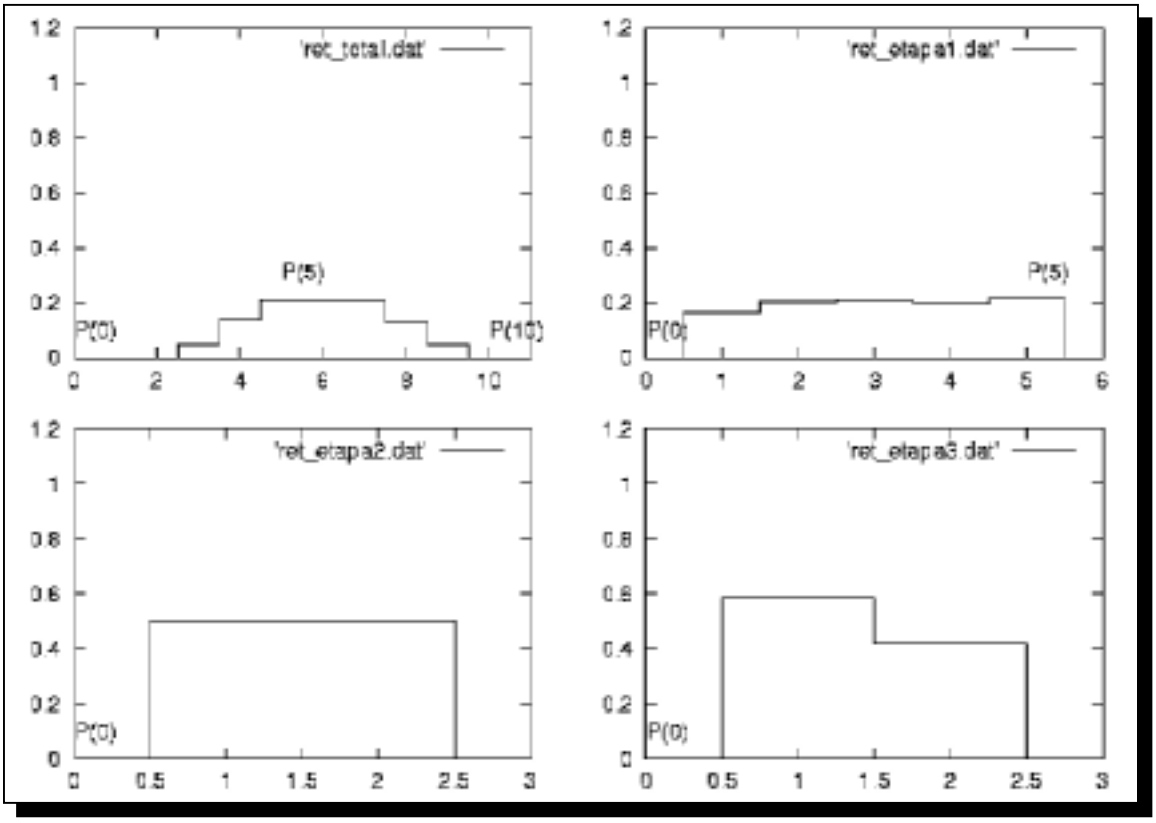

Fig.4.5 Histogramas de retardo de paquete

Visualización de los parámetros de salida

En Omnet, cuando realizamos una simulación termina se llama automáticamente a las funciones finish() de los métodos y el contenido se muestra en pantalla. Con esto se consigue mostrar los resultados finales de la simulación llevada a cabo. En las

arquitecturas que hemos implementadas los resultados visualizados se muestran en la figura 4.6.

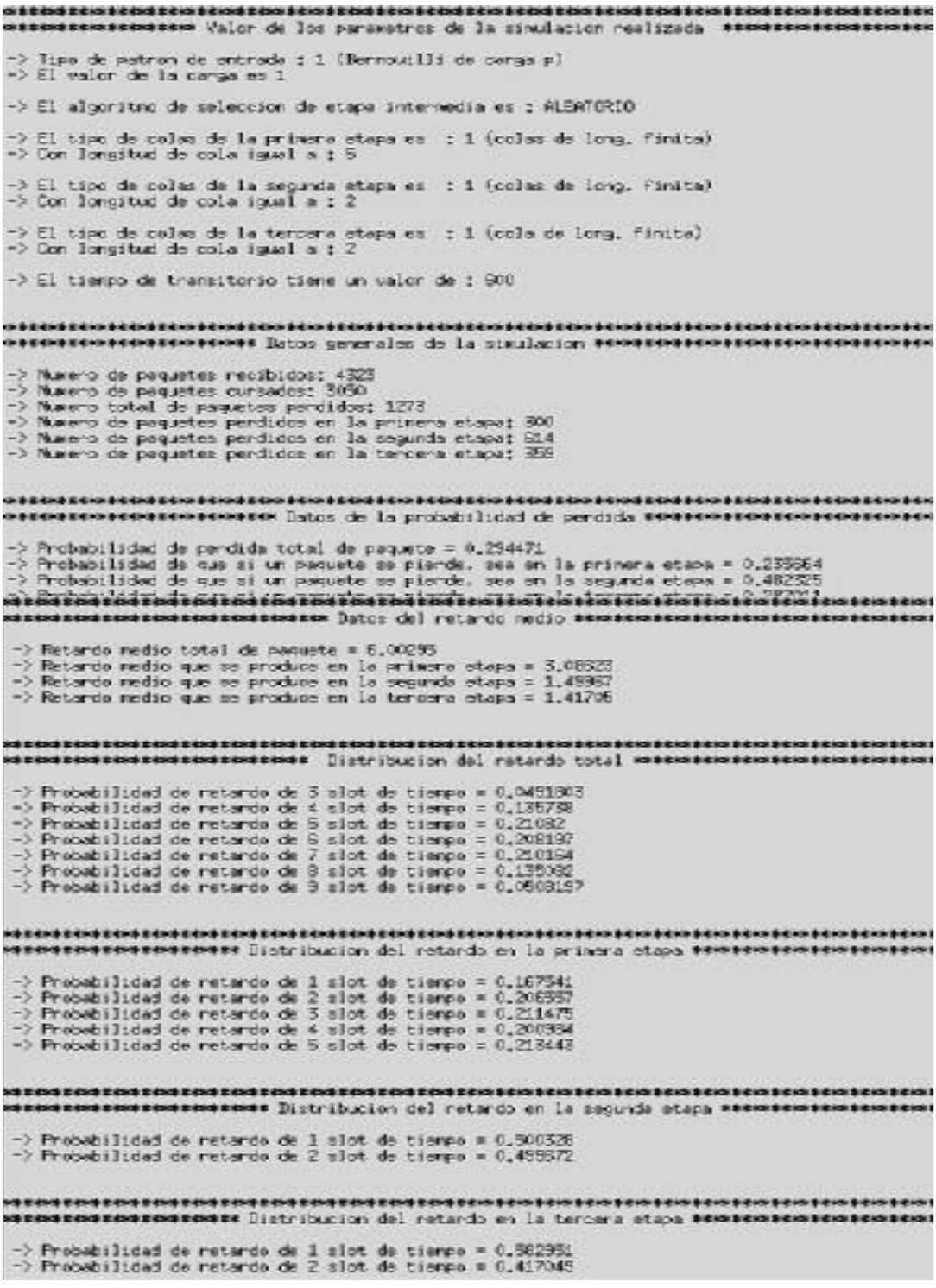

Fig. 4.7. Resultados de una simulación

Se observa como se muestran todos los parámetros de salida que se han explicado antes, además de los parámetros de entrada de forma que el usuario, en caso de no acordarse de los parámetros de entrada que introdujo para esa simulación, se le pueda recordar.

4.3 Enunciado de la Practica: "*Evaluación de una Red en 3 Etapas de Conmutadores Electrónicos de Paquetes con Colas a la Salida, para Paquetes de Longitud Fija"*

Al igual que en el estudio anterior, la práctica se dividirá en dos partes. En la primera se analizarán todos los detalles referentes a una red tres etapas genérica, tomando para este estudio una de las implementaciones realizadas (que será la red 4x4x4). Una vez estudiadas todas las características importantes de una red tres etapas se analizarán las distintas arquitecturas desarrolladas para ver las ventajas, inconvenientes,... que se presentan al utilizar una de esas arquitecturas en lugar de otra.

4.3.1 Parte I

Como ya se ha dicho, en esta parte se realizarán cuestiones referentes a una red en tres etapas tomando como referencia una de las redes implementadas, que será la red 4x4x4. La decisión de tomar esta red como referencia ha sido de forma arbitraria. Hubiese tenido el mismo resultado tomar otra de las topologías implementadas como referencia.

Cuestión 1

Para empezar es recomendable mirar el código y la implementación de esta red, de forma que se entienda el funcionamiento de todos los módulos detallados en el apartado 4.2.1. Observar como la descripción dada del funcionamiento de cada módulo se interpreta en el lenguaje, y como se diseñan los módulos compuestos con la herramienta *gned.*

Cuestión 2

Una vez entendido el código, arrancar el programa (./*nombre del ejecutable*) y realizar una simulación paso a paso (presionando F4). Observar cuales son los eventos que aparecen en la simulación. Observar también el interior de los módulos de conmutación para ver como entran y salen de ellos los paquetes de tráfico.

Cuestión 3

Realizar una simulación con los siguientes parámetros de entrada: carga =0.99, capacidad de buffer de primera etapa =5, tiempo de simulación =3000 ticks de reloj y tiempo de transitorio =300 ticks. El resto de los parámetros no importan. Los resultados que se tomarán son la probabilidad de retardo en la primera etapa, probabilidad de pérdida total, la probabilidad pérdida en la primera etapa y distribución de retardo de la primera etapa. Una vez anotados estos resultados se realizará una simulación con el conmutador del capítulo anterior 4x4 (que corresponde a la primera topología), introduciendo los mismos valores a los parámetros de entrada, recogiéndose el retardo medio, probabilidad de pérdida y distribución del retardo medio. ¿Qué ocurre? (**Nota:** la probabilidad de pérdida en una de las etapas se calcula como el producto de la pérdida total por la probabilidad de que si un paquete se pierda, este se pierda en una etapa concreta)

Con esta cuestión lo que se quiere comprobar es que realmente los nodos de las redes implementadas son los conmutadores con colas a la salida estudiados en el capítulo anterior.

Cuestión 4

Ahora se comprobará la diferencia que hay entre la elección de un algoritmo de encaminamiento de segunda etapa aleatorio o secuencial.

Para ello realizar dos simulaciones, ambas con los mismos parámetros de entrada, siendo los parámetros introducidos los siguientes: tipo de colas (todas las etapas) =1, longitud colas (de todas las etapas) =5, carga =0.9, tiempo de simulación = 3000 *ticks,* tiempo de transitorio =300 *ticks.* El único parámetro que se cambiará de una simulación a otra es el de selección de algoritmo de encaminamiento. Se pondrá en la primera simulación se pondrá un encaminamiento aleatorio (valor 1) , y en la segunda simulación se pondrá un encaminamiento secuencial (valor 2).

El resultado que se observará es la probabilidad de que si un paquete se pierda, este sea en la primera etapa. ¿Cuál es la diferencia entre ambas simulaciones?¿Son coherentes los resultados?¿ Qué ocurriría, en el caso de que tuviesen más puertos de entrada en cada módulo de primera etapa, si se utilizase el algoritmo secuencial?

Cuestión 5

Se realizará a continuación una gráfica donde se representará la perdida sufrida en la red ante distintos valores de carga de entrada utilizando los siguientes parámetros de entrada: tipo de colas (todas las etapas) =1, longitud colas (de todas las etapas) =5, tiempo de simulación = 3000 *ticks,* tiempo de transitorio =300 *ticks,* tipo de algoritmo de selección de segunda etapa =1 (aleatorio). Los valores de la carga serán 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99.

Mirar también para cada simulación la probabilidad de perdida en la 1ª, 2ª, o 3ª etapa y ver cual es el mayor. ¿Por qué crees que sucede así?¿Se puede balancear la perdida entre las distintas etapas? Si es posible indicar cómo. (**Nota:** la probabilidad de pérdida en una de las etapas se calcula como el producto de la pérdida total por la probabilidad de que si un paquete se pierda, este se pierda en una etapa concreta)

Cuestión 6

Realizar, ante las mismas condiciones anteriores, una gráfica del retardo medio de paquete. ¿Cuál es el mínimo y máximo retardo que se puede dar?¿Por qué?

4.3.2 Parte II

En esta parte se estudiarán las tres redes implementadas (Fig.4.8,4.9,4.10), comparándolas entre ellas para ver las prestaciones que ofrece cada topología: ventajas, inconvenientes,..

Cuestión 1

Lo primero que se hará es observar el retardo medio de paquete en las distintas topologías ante las mismas condiciones de entrada. Para esto realizar la siguiente simulación para cada topología: carga =0.8, algoritmo selección de segunda etapa =1 (aleatorio), tipo de colas (en cualquier etapa) =1, longitud de las colas (para cualquier etapa) =5, tiempo simulación =3000 *ticks,* tiempo transitorio =300 *tikcs.*

Hacer una tabla con el retardo medio en cada simulación (tanto el retardo medio total como el de cada etapa), y contrastar los resultados.

Por último, observar los histogramas de cada simulación para terminar de entender los resultados.

Cuestión 2

Realizar la misma simulación anterior y comprobar la probabilidad de pérdida sufrida en cada red.

Para esto realizar una tabla en la que se muestren los resultados de la probabilidad de pérdida para cada red y después analizar los resultados.

Como estudio optativo, comprobar en cada simulación la probabilidad de que si un paquete se pierde, este se pierda en una etapa dada y posteriormente analizar los resultados.

4.4 Resultados Obtenidos

En esta sección se detallan las respuestas a las cuestiones realizadas en la sección 4.3. Además se expondrá una breve conclusión sobre las redes multietapa y con esto se finalizará el capítulo.

4.4.1 Respuestas a la Parte I

Cuestión 1

Para empezar es recomendable mirar el código y la implementación de esta red, de forma que se entienda el funcionamiento de todos los módulos detallados en el apartado 4.2.1. Observar como la descripción dada del funcionamiento de cada módulo se interpreta en el lenguaje, y como se diseñan los módulos compuestos con la herramienta gned.

Todo lo referente al código de los programas está bastante claro y documentado dentro del propio código. Además, todo se ha ido exponiendo a lo largo de todo el capítulo y más concretamente en la sección 4.2 de forma muy detallada.

Cuestión 2

Una vez entendido el código, arrancar el programa (./nombre del ejecutable) y realizar una simulación paso a paso (presionando F4). Observar cuales son los eventos que aparecen en la simulación. Observar también el interior de los módulos de conmutación para ver como entran y salen de ellos los paquetes de tráfico.

En la simulación de un *slot* se puede comprobar, si se sigue detenidamente, que los eventos que se van sucediendo son los siguientes (se tomará como referencia el primer *slot* temporal) : en el instante t =0 se genera la señal de control al generador y comienza a generarse tráfico en el generador; en el instante t =0.2 se produce la señal de control de los conmutadores de primera etapa, y cuando se recibe, si se viese el interior de uno de estos conmutadores, se observaría como comienzan a trabajar los componentes que forman el conmutador (*switch* y colas), almacenando todos los paquetes el *switch* antes de la señal de control, una vez recibe esta señal manda todos los paquetes a sus correspondientes puertos de salida, y cuando termina envía un paquete de *finish* a cada cola (el paquete se distingue por el color blanco), cuando las colas reciben el paquete de *finish* transmiten el paquete de cabeza de cola (si lo hay), y después almacenan todos los paquetes en la cola principal; en el instante t =0.4 se produce la señal de control de los conmutadores de segunda etapa, y al igual que antes, una vez que los conmutadores reciben la señal comienzan a trabajar, mandando los paquetes a los correspondientes puertos y quedando almacenados en las colas hasta que se puedan transmitir. El último evento ocurre en el instante t =0.6 con la señal de control de los conmutadores de tercera etapa.

Después de estos, el siguiente evento aparecerá en t =1, volviéndose a repetir el ciclo.

Cuestión 3

Realizar una simulación con los siguientes parámetros de entrada: carga =0.99, capacidad de buffer de primera etapa =5, tiempo de simulación =3000 ticks de reloj y tiempo de transitorio =300 ticks. El resto de los parámetros no importan. Los resultados que se tomarán son la probabilidad de retardo en la primera etapa, probabilidad de pérdida total, la probabilidad pérdida en la primera etapa y distribución de retardo de la primera etapa. Una vez anotados estos resultados se realizará una simulación con el conmutador del capítulo anterior 4x4 (que corresponde a la primera topología), introduciendo los mismos valores a los parámetros de entrada, recogiéndose el retardo medio, probabilidad de pérdida y distribución del retardo medio. ¿Qué ocurre?(***Nota:*** *la probabilidad de pérdida en una de las etapas se calcula como el producto de la pérdida total por la probabilidad de que si un paquete se pierda, este se pierda en una etapa concreta)*

Con esta cuestión lo que se quiere comprobar es que realmente los nodos de las redes implementadas son los conmutadores con colas a la salida estudiados en el capítulo anterior.

Si se realiza la simulación solicitada se obtienen los siguientes resultados en la red:

- Probabilidad de pérdida total = 0.286733
- Probabilidad de que si un paquete se pierde, esto suceda en la primera etapa = 0.243008
- Prob. de pérdida en 1ª etapa =Prob. pérdida total * Prob. de que si un paquete se pierde, esto suceda en la 1ª etapa => Prob. pérdida 1ª etapa = 0.286733 * 0.243008 = 0.0696784 .
- Distribución del retardo:
  - P(0) = 0
  - P(1) =0.177965
  - P(2) =0.208535
  - P(3) =0.214658
  - P(4) =0.201885
  - P(5) =0.196957
- Retardo medio = 3.03134

Si ahora se realiza la simulación en el conmutador 4x4 del capítulo anterior, los resultados obtenidos son los siguientes:

- Probabilidad de pérdida = 0.0642
- Distribución del retardo:
  - P(0) = 0
  - P(1) =0.176
  - P(2) =0.218
  - P(3) =0.214
  - P(4) =0.203
  - P(5) =0.187
- Retardo medio = 3.00011

Como se puede ver, ambos resultados son casi idénticos, y es así porque los dos dispositivos estudiados son iguales, ambos son conmutadores con colas a la

salida, de tamaño 4x4, con la misma cantidad de posiciones de memoria y gestionados por un módulo de control.

Con esto lo que se pretendía mostrar es que realmente los conmutadores implementados en el capítulo anterior han sido utilizados a la hora de construir las redes de este capítulo. Así se pueden evaluar redes formadas con estos dispositivos para determinar las prestaciones que ofrecen este tipo de redes, que es el objetivo de este capítulo.

Cuestión 4

Ahora se comprobará la diferencia que hay entre la elección de un algoritmo de encaminamiento de segunda etapa aleatorio o secuencial.

Para ello realizar dos simulaciones, ambas con los mismos parámetros de entrada, siendo los parámetros introducidos los siguientes: tipo de colas (todas las etapas) =1, longitud colas (de todas las etapas) =5, carga =0.9, tiempo de simulación = 3000 ticks, tiempo de transitorio =300 ticks. El único parámetro que se cambiará de una simulación a otra es el de selección de algoritmo de encaminamiento. Se pondrá en la primera simulación se pondrá un encaminamiento aleatorio (valor 1) , y en la segunda simulación se pondrá un encaminamiento secuencial (valor 2).

El resultado que se observará es la probabilidad de que si un paquete se pierda, este sea en la primera etapa. ¿Cuál es la diferencia entre ambas simulaciones?¿Son coherentes los resultados?¿Qué ocurriría, en el caso de que tuviesen más puertos de entrada en cada módulo de primera etapa, si se utilizase el algoritmo secuencial?

Con esta cuestión lo que se pretende es determinar el funcionamiento de los dos algoritmos de encaminamiento de segunda etapa implementados y como afecta la elección de uno u otro al rendimiento de la red.

Si se realiza la simulación se obtienen los siguientes resultados para el algoritmo aleatorio:

- Probabilidad de pérdida total = 0.14005
- Probabilidad de que si un paquete se pierda sea en la primera etapa =0.494192
- Número de paquetes perdidos en la primera etapa =2042 de un total de paquetes perdidos de 4132.

Los datos obtenidos para el algoritmo distribuido son:

- Probabilidad de pérdida total = 0.10518
- Probabilidad de que si un paquete se pierda sea en la primera etapa =0
- Número de paquetes perdidos en la primera etapa =0 de un total de paquetes perdidos de 3074

Se advierte que se tendrían mejores resultados en esta red implementada si se utilizase en las simulaciones el algoritmo distribuido, puesto que se elimina la pérdida de paquetes en la 1ª etapa, y esto a su vez reduce la probabilidad de pérdida total y el número de paquetes perdidos. Y este es un resultado lógico, puesto que con el algoritmo distribuido se balancea mucho más la carga entre los distintos puertos de salida que si se utiliza un algoritmo aleatorio.

La razón de que se obtengan esos resultados en la primera etapa es que estos módulos tienen 4 puertos de entrada y 4 de salida, lo que hace que al utilizar el algoritmo distribuido (a pesar de que la carga sea la máxima, que equivaldría a un paquete por puerto de entrada en cada *slot* de tiempo), nunca saturaría una cola. Esto es así porque, en un instante de tiempo, como mucho, se almacenará un paquete en una cola y a su vez se transmitiría un paquete (el que hubiese almacenado desde el

slot anterior), lo que produce que en una cola de primera etapa, como máximo, en un *slot* de tiempo, sólo haya almacenado un paquete.

Para terminar se puede decir que el algoritmo de encaminamiento aleatorio resulta menos eficiente que un algoritmo distribuido debido a que se podría saturar un puerto de salida debido a que con el algoritmo aleatorio no balanceamos la carga, simplemente se encaminan los paquetes de una forma arbitraria, y al no tener conocimiento de los estados de los puertos puede que se saturen si se le continuasen enviando paquetes de esta manera.

Cuestión 5

Se realizará a continuación una gráfica donde se representará la perdida sufrida en la red ante distintos valores de carga de entrada utilizando los siguientes parámetros de entrada: tipo de colas (todas las etapas) =1, longitud colas (de todas las etapas) =5, tiempo de simulación = 3000 ticks, tiempo de transitorio =300 ticks, tipo de algoritmo de selección de segunda etapa =1 (aleatorio). Los valores de la carga serán 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99.

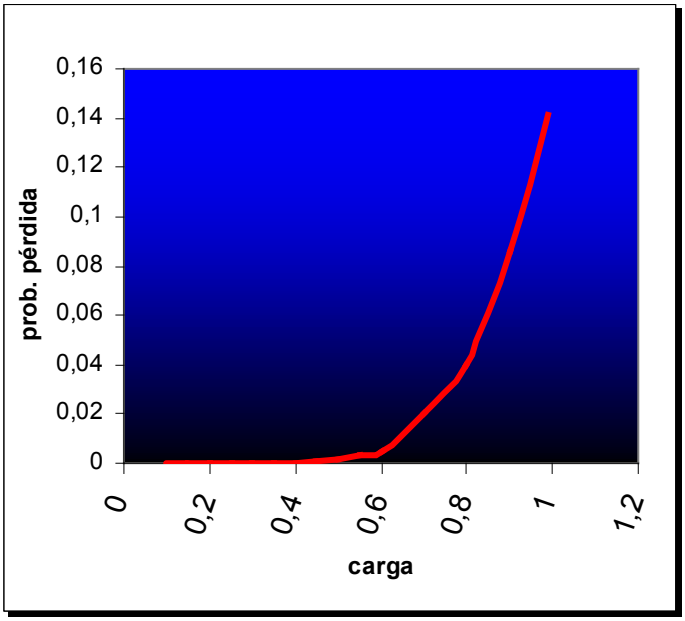

Fig.4.8. Gráfica de la Prob. Pérdida total ante distintas cargas

Como se observa en esta curva (Fig.4.8.), el aspecto es similar al de los conmutadores estudiados en el capítulo anterior, puesto que la red está formada por ellos.

La pérdida al principio es nula, hasta que el valor de la carga adquiere valores superiores a 0.5. A partir de ahí la curva se incrementa de forma abrupta, y es debido principalmente a la ocupación de las colas de los conmutadores de las distintas etapas. A esos niveles de carga, cuando un paquete llega a un etapa de conmutación se encuentra la cola del puerto de salida al que va dirigido totalmente ocupada, produciendo su eliminación.

*Mirar también para cada simulación la probabilidad de perdida en la 1ª, 2ª, o 3ª etapa y ver cual es el mayor. ¿Por qué crees que sucede así? (**Nota:** la probabilidad de pérdida en una de las etapas se calcula como el producto de la pérdida total por la probabilidad de que si un paquete se pierda, este se pierda en una etapa concreta)*

A continuación se muestran las gráficas de la probabilidad de pérdida correspondientes a las distintas etapas (figuras de la 4.9 a la 4.11), y se observará

como a pesar de tener la misma curva, las pérdidas entre las distintas etapas son diferentes.

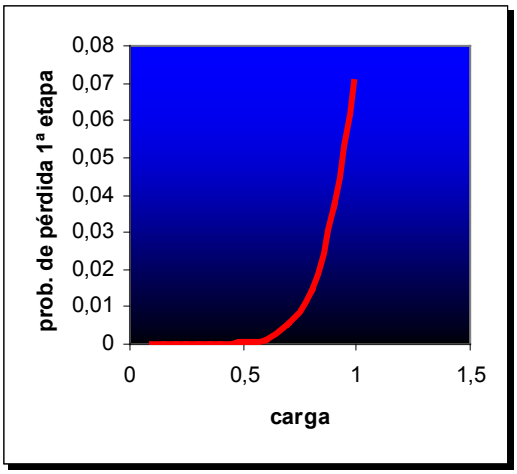

Fig.4.9 Prob. de pérdida en la primera etapa

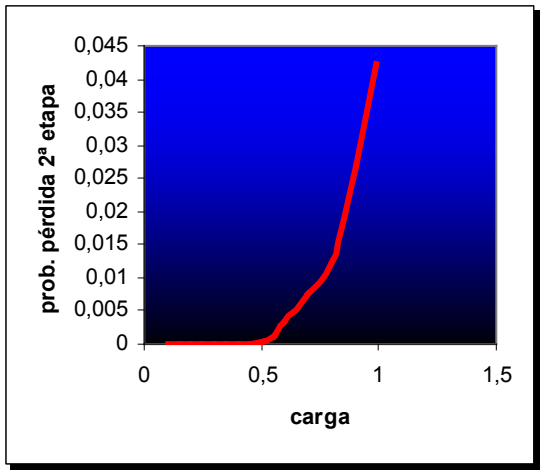

Fig.4.10 Prob. de pérdida en la segunda etapa

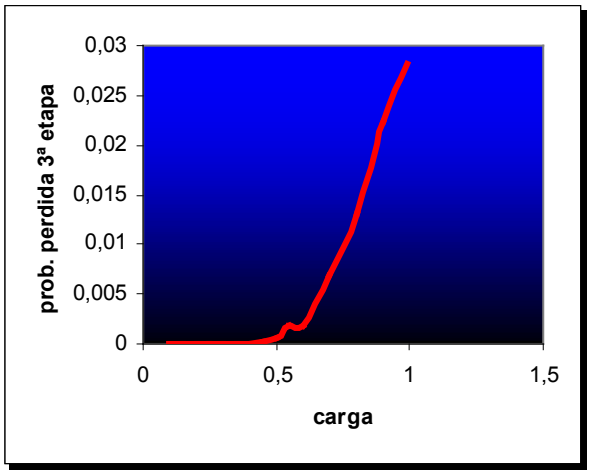

Fig.4.11 Prob. de pérdida en la tercera etapa

Como se puede observar, todas las gráficas tienen la misma curva, al principio la pérdida es nula, hasta que el valor de la carga para de 0.5 y comienza a ser considerablemente alta. Sin embargo, los valores de pérdida son más altos en la primera etapa, luego la segunda etapa y por último, la etapa que menos pérdida sufre es la tercera. Y esto es así porque al tener todas las etapas el mismo tamaño de *buffer,* es lógico pensar, que las primera colas que se saturarán serán las colas de primera etapa, lo que hará que eliminen más paquetes que las restantes etapas.

¿Se puede balancear la perdida entre las distintas etapas? Si es posible indicar cómo.

Para balancear esta pérdida lo que se suele hacer es poner los *buffers* de las primeras etapas de la red con mayor número de posiciones de memoria que las colas de las etapas finales de la red. De esta forma, pérdida se equilibra más entre las distintas etapas.

A continuación y a modo opcional se muestra una gráfica (figura 4.12) de la simulación realizada donde se muestra el porcentaje de pérdida de cada etapa. En ella se observa lo dicho anteriormente, que la etapa que más pérdida sufre es la primera, luego la segunda y por último la tercera.

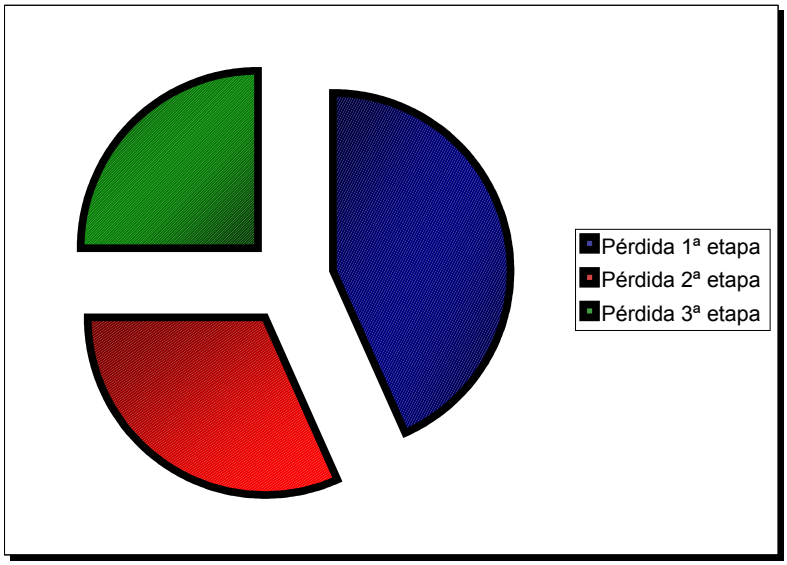

Fig.4.12 Porcentaje de pérdida en cada etapa

Cuestión 6

Realizar, ante las mismas condiciones anteriores, una gráfica del retardo medio de paquete.

Esta cuestión servirá para terminar de establecer las similitudes entre estas redes y lo que influye el estar compuestas por los elementos estudiados en el capítulo anterior (conmutadores con colas a la salida).

Si en el apartado anterior las curvas de la probabilidad de pérdida son idénticas (en aspecto) a las curvas de probabilidad de pérdida mostradas en el capítulo anterior, en esta cuestión se verá como el retardo medio sufrido en las redes también coincide con las curvas mostradas del retardo medio en el anterior capítulo.

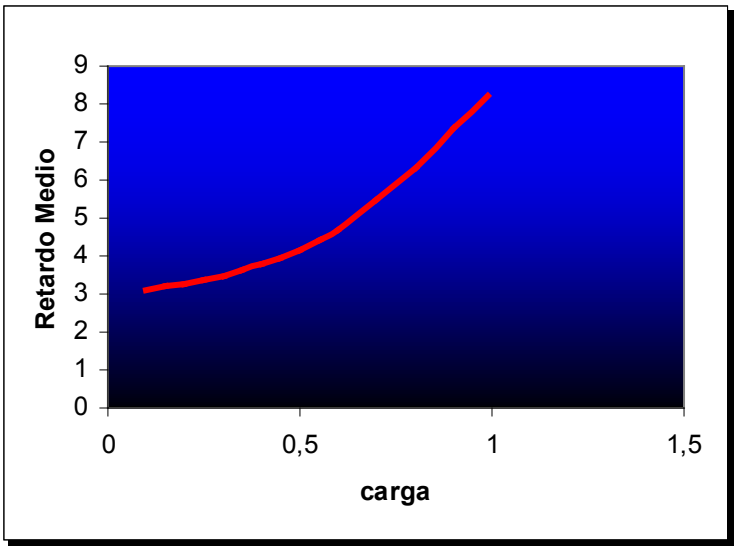

Fig.4.13 Retardo medio en función de la carga

La figura 4.13 muestra como el retardo medio es una función exponencial pero a diferencia de las representadas anteriormente, esta función crece lentamente, sin tener ningún punto crítico a partir del cual se dispare el retardo. Únicamente, se podría decir que a partir de una carga superior del 50% aumenta algo más rápido el retardo medio.

¿Cuál es el mínimo y máximo retardo que se puede dar? ¿Por qué?

Si se observa el gráfico, como era de esperar, el retardo medio es de tres *slots* temporales, y esto es un resultado lógico, puesto que un paquete tardará al menos tres unidades de tiempo en atravesar la red, puesto que según llega a una etapa de conmutación, el paquete deberá esperar al siguiente *slot* temporal para transmitirse a la siguiente etapa de conmutación, debido a la configuración de las colas. Recordar que la colas implementadas primero transmiten y luego leen.

Por otro lado, el retardo máximo que se podría dar, como es lógico, en esta gráfica no aparece, puesto que es una representación del retardo medio. Aun así se puede deducir, sabiendo que las colas de cada etapa tienen 5 posiciones de memoria cada una. Es por esto, que un paquete, que tenga que esperar el máximo en cada etapa sin ser eliminado, esperará un total de 15 *sltos* temporales. Así, se deduce que el máximo retardo que un paquete puede sufrir al atravesar la red es la suma siguiente: posiciones de memoria de 1ª etapa + posiciones de memoria de 2ª etapa + posiciones de memoria de 3ª etapa. Para comprobar esto, lo único que hay que hacer es mirar la función de distribución del retardo total, una vez termine la simulación y comprobar cual es el máximo retardo sufrido (para comprobar esto, realizar la simulación ante una carga de entrada alta, >0.5).

4.4.2 Respuestas a la Parte II

Cuestión 1

Lo primero que se hará es observar el retardo medio de paquete en las distintas topologías ante las mismas condiciones de entrada. Para esto realizar la siguiente simulación para cada topología: carga =0.8, algoritmo selección de segunda etapa =1(aleatorio), tipo de colas (en cualquier etapa) =1, longitud de las colas (para cualquier etapa) =5, tiempo simulación =3000 ticks, tiempo transitorio =300 tikcs.

Hacer una tabla con el retardo medio en cada simulación (tanto el retardo medio total como el de cada etapa), y contrastar los resultados.

La tabla de resultados que se debe de obtener es la siguiente:

Red	Retardo	Retardo en 1ª etapa	Retardo en 2ª etapa	Retardo en 3ª etapa
4x4x4	7.3721	2.58	2.43	2.36
4x6x8	5.73105	1.53935	1.55879	2.63292
4x8x4	5.26873	1.303	1.32	2.644

Como se puede ver, la red en la que mayor unidades de tiempo se retardan los paquetes es en la primera topología y esto puede resultar curioso. A simple vista, se podría haber dicho que la red que mayor retardo sufre sería la red 4x8x4 al tener un mayor número de dispositivos. Pero esto no es así, y la razón es bastante sencilla: al tener menos dispositivos la red, cuando un paquete llegue a una etapa se encontrará

todos las posiciones de memoria de las colas que hay en los distintos puertos de salida ocupadas, como consecuencia de tener menos dispositivos en la red.

Este retardo también va afectado por la carga de la simulación. Hay que recordar que las simulaciones se han realizado con una carga igual a 0.9. Pero aun cambiando los valores de carga, los resultados serían similares hasta llegar a un punto de carga extremadamente baja, en cuyo caso los resultados del retardo serían similares en todas las topologías.

Si ahora se analiza el retardo medio de cada red en cada etapa se puede decir que el retardo en primera y segunda etapa de una red con mayor número de etapas intermedias que otra ejercen menos retardo en los paquetes, a la hora de transmitirlos. Sin embargo ese retardo, analizado en la tercera etapa, el retardo es más o menos idéntico en las tres arquitecturas. La conclusión que se saca de esto es que la etapa que mayor influencia ejerce en el retardo medio de paquete es la primera etapa y la segunda.

Como último detalle del análisis, se puede ver en la tabla como la red 4x8x4 tiene un retardo de primera etapa casi mínimo (el mínimo sería 1 *slot*). Es debido a lo mencionado anteriormente, la red, al tener mayor número de etapas intermedia , los conmutadores de primera etapa también tienen mayor número de puertos de salida, y esto hace que los paquetes no se saturen en un mismo puerto sino que pueden elegir entre un mayor rango de puertos de salida. Así las colas de primera etapa tendrán una menor ocupación que en una red 4x4x4.

Cuestión 2

Realizar la misma simulación anterior y comprobar la probabilidad de pérdida sufrida en cada red.

Para esto realizar una tabla en la que se muestren los resultados de la probabilidad de pérdida para cada red y después analizar los resultados.

Como estudio optativo, comprobar en cada simulación la probabilidad de que si un paquete se pierde, este se pierda en una etapa dada y posteriormente analizar los resultados.

La tabla de resultados, junto con los datos optativos, es la siguiente:

Red	Prob. de pérdida total	Prob. de que si un paq. Se pierda, sea en la 1ª etapa	Prob. de que si un paq. Se pierda, sea en la 2ª etapa	Prob. de que si un paq. Se pierda, sea en la 3ª etapa
4x4x4	0.086	0.428	0.3118	0.2599
4x6x4	0.0460	0.0278	0.045	0.927
4x8x4	0.0465	0.002	0.004	0.991

Lo primero que se analizará será la pérdida total sufrida. Como se observa, y como era de esperar, la arquitectura que mayor pérdida sufre es la arquitectura 4x4x4, y es debido a que al tener menor número de etapas intermedias, los paquetes tienen menos dispositivos donde poder almacenarse, la red consta con menos posiciones de memoria que las demás. Esto produce que cuando un paquete llegue a una etapa se encuentre las colas del conmutador llenas y tengan que rechazarse.

Si ahora se pone atención en la probabilidad de que un paquete si se ha perdido, la pérdida se produjera en una etapa concreta, se advierte que, en general, una red

con menor número de etapas intermedia produce mayor pérdida que una red con mayor número de etapas intermedias en la primera etapa. Sin embargo la tercera etapa sufrirá más pérdidas en una red con mayor número de etapas intermedias que otra con menor número de etapas intermedias. En las simulaciones realizadas, la red 4x4x4 tiene mayor porcentaje de pérdidas que una red 4x8x4 en la primera etapa (0.428 y 0.002 respectivamente), y por el contrario, la red 4x8x4 tiene mayor porcentaje de pérdidas en la tercera etapa que la red 4x4x4 (0.991 y 0.2599 respectivamente). Esto es producido a que una red con mayor número de etapas intermedias actuaría, estableciendo un símil, como un concentrador de los estudiados en el capítulo anterior.

Capítulo 5
CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo de este proyecto se han analizado algunas características de diseño de conmutadores electrónicos de paquetes con colas a la salida, redes de conmutación de paquetes con topología de red de Clos, formada por módulos conmutadores con colas a la salida. El estudio se ha centrado en conmutadores trabajando con paquetes de longitud fija. El enfoque ha sido una aproximación didáctica, en forma de prácticas, de manera que es el lector el que busca las respuestas a las cuestiones realizadas indagando en la problemática presentada. De esta forma tendrá más conciencia sobre los elementos que influyen en las prestaciones de estos dispositivos, y como afectan de manera positiva o negativa al rendimiento de una red.

Se analizan dos aspectos básicos del rendimiento de un dispositivo de conmutación: el retardo que sufre un paquete cuando atraviesa un conmutador, y la probabilidad de pérdida de paquete. Se ha intentado mostrar por medio de prácticas cómo estos factores dependen de parámetros como la carga de entrada y el número de posiciones de memoria de las que consta el conmutador. El lector debe de tener claro llegado a este punto de que no se puede controlar la carga de una red en cualquier instante pero si se puede diseñar el conmutador de forma que produzca un buen rendimiento de la red hasta ciertas cotas de carga.

Por sus ventajas didácticas, todas las simulaciones han sido realizadas con la herramienta de simulación *Omnet++* [5][6] por lo que el lector también ha podido aprender en que consiste esta herramienta de simulación, cuáles son sus características principales, y su aplicación para la evaluación de las arquitecturas seleccionadas. El enfoque didáctico pretende ayudar a la formación del lector a la hora de desarrollar algún ejercicio o proyecto, investigar alguna arquitectura que desee implementar en la realidad.

Por último, *Omnet++* se basa en el lenguaje C++ [12], lo que ha permitido al autor de este proyecto (y permitirá a los posibles lectores) adquirir valiosos conocimientos en este lenguaje, muy extendido y útil

Como línea futura indicar que el simulador ha sido diseñado de manera modular, con vistas a poder ser extendido con otros módulos conmutadores y/o generadores de tráfico que complementen su funcionalidad.

Apéndice A
REPRESENTACION DE GRÁFICAS USANDO PLOVE

A.1 Introducción

Plove es una herramienta utilizada para la configuración de gráficas a partir de los vectores de salida de *Omnet++* [5] [6] (donde están almacenados los resultados de la configuración). Además utiliza *gnuplot* para realizar la configuración.

Se le puede especificar el estilo de gráfica que se desea obtener (líneas, puntos, etc) para cada vector, así como el establecimiento de las configuración características de dibujo como son las escalas, títulos, etc.

Antes de realizar la configuración se puede realizar un filtrado para obtener una gráfica determinada (obtener una media, una configuración normal, etc). Hay algunos filtros que ya están definidos en la misma herramienta y a parte, el usuario puede diseñar sus propios filtros. Además estos se pueden establecer de tres maneras distintas: con configuración *awk*, con programas *awk* o con *scripts* de otros programas externos.

Una limitación que presentan estos filtros es que no pueden representar varios filtros sobre el mismo vector.

Por otro lado, *Plove* no utiliza archivos temporales, por lo que el usuario no tiene que preocuparse por el espacio que ocupe la gráfica en el disco (sólo se almacena cuando se le indica). Además, si el vector está en un archivo comprimido del tipo *gzipped* se podrá realizar su gráfica sin necesidad de descomprimir el archivo.

Decir también que *Plove* no altera el contenido de los vectores y cuando arranca el programa, el archivo que lee es el *.ploverc* donde se encuentra la configuración del programa.

A.2 Forma de Uso

En la figura A.1 se muestra la pantalla de inicio de *Plove*, donde se realizará todo el trabajo antes de representar la gráfica. A continuación se explicará la manera de representar los vectores utilizando esta herramienta.

Primero, se debe cargar un vector (*.vec*) en el plano izquierdo de la pantalla. Pueden ser también archivos comprimidos, como se mencionó anteriormente (*.vec.gz*), sin necesidad de ser descomprimidos.

Se puede trasladar un vector seleccionado en el plano izquierdo de la pantalla al plano derecho haciendo clic en el icono de la flecha que indica la dirección a la derecha (en el centro de la pantalla).

El icono de la parte derecha que indica *PLOT!* Representa la gráfica del vector seleccionado en el plano derecho de la pantalla.

Para cambiar el estilo de la línea, añadir un filtro o cambiar el estilo de la línea se debe de hacer clic en el botón de la parte inferior del plano derecho que dice *Options...*

El botón *Options...* de la parte derecha de la pantalla es utilizado para indicar los títulos del gráfico, las escalas y algunos comandos para la representación del gráfico.

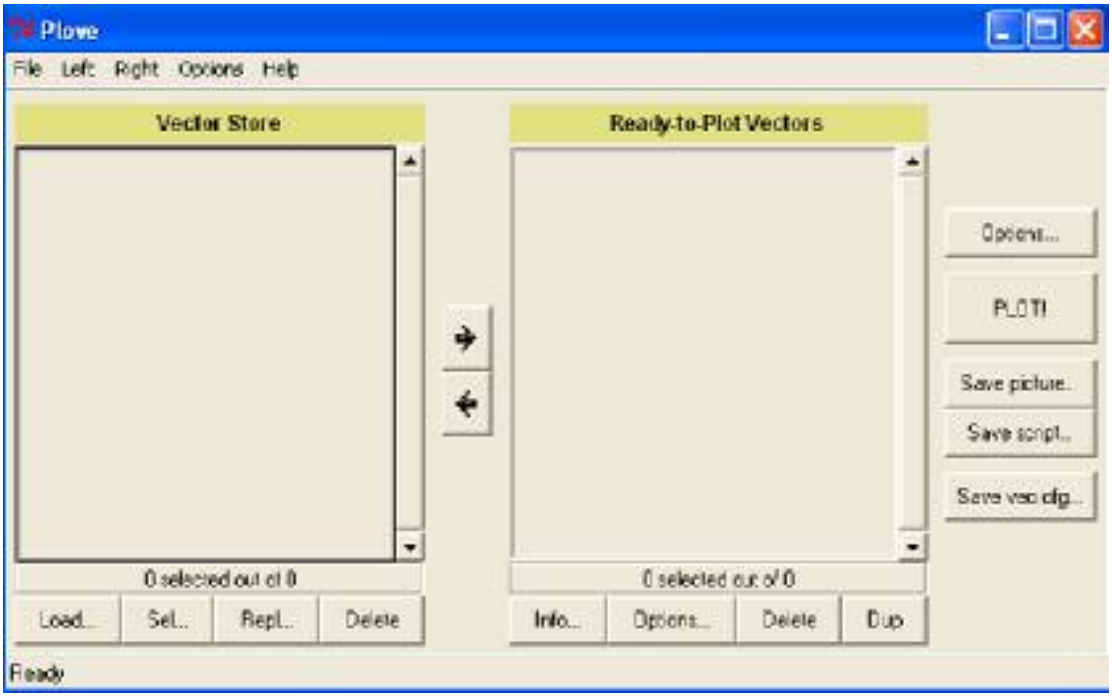

Fig. A.1 Pantalla principal de Plove

En el botón *Info...* de la parte inferior del plano derecho sirve para indicar al usuario sobre la información que porta el vector que está seleccionado y el botón de *Delete* sirve para eliminar el vector seleccionado del plano derecho.

En la parte del plano izquierdo, las operaciones que se nos permiten realizar mediante los botones que aparecen en la parte inferior son: cargar un vector (*Load...*), seleccionar otros vectores (*Sel...*), duplicar un vector (*Repl...*) o borrar el vector/es seleccionado/s (*Delete*).

A.3 Diseño de los Filtros

Como se mencionó los filtros pueden ser implementados de tres formas distintas: con expresiones awk, con programas awk o mediante la utilización de *scripts* de programas externos.

Una expresión awk tiene esta forma:

... | awk '{$3 = <expression>; print}' | ...

un programa awk se diseña de la siguiente manera:

... | awk '{$3 = <expression>; print}' | ...

y por último, un programa externo se ejecutará de la siguiente manera:

... | <program> <parameters> | ...

(para más información acerca de estos filtros mirar la ayuda proporcionada por la propia herramienta o en el manual de referencia de *Omnet++*).

Referencias

[1] "Comunicaciones y Redes de Computadores". Sexta edición.
STALLINGS, William
Editorial Prentice Hall, 2000.
ISBN: 84-205-2986-9

[2] pcl.cs.ucla.edu/projects/mirsim/mirsim.html

[3] pcl.cs.ucla.edu/projects/parsec/manual

[4] pcl.cs.ucla.edu/projects/maisie/maisie.html

[5] Manual de Referencia de *Omnet*++

[6] www.omnet.org/external/doc/html/usman.php

[7] www.rediris.es/rediris/boletin/38/ponencia5.html

[8] 166.144.106.9/~herrera/INF-241/atm.htm

[9] http://agamenon.uniandes.edu.co/~c21437/alejan-d/rdetapas.html

[10] http://www.lcc.uma.es/~eat/services/di_si_ma.htm#link1_1

[11] "Cómo programar en C++", 2ª Ed.
Deitel, H.M.
ISBN: 968-880-471-1

[12] Manual OpNet.
http://telematica.cicese.mx/revistatel/telem@tica_anoii_no6.pdf