# UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN


# Análisis UML del simulador PASS

**Diego Linares Pagán**

Cartagena, Octubre 2003

# Índice

# Índice de Figuras

# Capítulo 1
# Introducción

## 1.1 Antecedentes

El simulador *Packet Switch Simulator (PASS)* ha sido desarrollado dentro del área de Ingeniería Telemática con el objetivo de evaluar conmutadores de paquetes.

Por otro lado tal y como veremos en profundidad en el capítulo 2 la tecnología óptica de conmutación de paquetes es la tecnología con mayores perspectivas de futuro para dirigir el tráfico en Internet. Y en la actualidad no se dispone de herramientas de propósito específico para la evaluación de conmutadores en redes *WDM (Wavelength Division Multiplexing).*

Por todo ello se consideró de especial relevancia la necesidad de analizar en profundidad el simulador *PASS*, así como estudiar los cambios necesarios en el mismo para poder evaluar conmutadores o *switchs* ópticos (de cara a una futura implementación).

Para realizar dicho análisis se eligió *UML (Unified Modeling Language)* que es un lenguaje para especificar, construir, visualizar y documentar los sistemas software.

## 1.2 Objetivos

El objetivo del presente proyecto consiste en analizar el simulador *PASS (Packet Switch Simulator*) desarrollado en el Área de Ingeniería Telemática de la Universidad Politécnica de Cartagena (UPCT) mediante el lenguaje de modelado *UML* para dar una visión clara de su estructura y funcionamiento, con el fin de estudiar como abordar las futuras modificaciones a realizar para extender su funcionalidad a conmutadores ópticos de redes *WDM*, conmutadores con número de etapas variable, permitir diversos tipos de interconexión entre los elementos internos del conmutador o realizar gráficas con los resultados de la simulación. (Implementar dichas modificaciones no es objetivo de este proyecto).

## 1.3 Estructura del contenido

En este apartado se hace una breve introducción acerca de los capítulos que forman éste documento:

- Capítulo 2: Situación y contexto

  En este capítulo intentaremos dar una visión global de las redes *WDM* así como de las diversas técnicas que se emplean en la conmutación óptica de paquetes, resaltando su importancia de cara a mejorar la capacidad de tráfico de Internet. Veremos ejemplos de otros simuladores tanto de propósito general como de propósito específico. También describiremos las características del lenguaje de modelado *UML* y las posibilidades que nos ofrece. Por último

describimos la arquitectura general del simulador *Packet Switch Simulator* atendiendo a los diferentes módulos que lo forman y explicando las ventajas que ofrece frente a otras herramientas de simulación.

- Capítulo 3: Modelado estático del simulador

  A lo largo de este capítulo se mostrará una visión arquitectural y modular de *PASS*. Siguiendo la metodología *UML*, el apartado queda dividido en una primera parte en la que se muestra una visión modular mediante diagramas de casos, y una segunda parte en la que se realiza el modelado de los objetos, la cual da idea de cómo la herramienta trata la información necesaria para realizar sus tareas.

- Capítulo 4: Modelado dinámico del simulador

  En este capítulo trataremos la parte correspondiente al modelado dinámico de la herramienta, que muestra cuales son los procesos a ejecutar, cómo se ejecutan, el orden y la información que fluye entre éstos. Siguiendo la metodología *UML* esta última parte se describiría mediante diagramas de estado, diagramas de actividades y diagramas de interacción. Sin embargo debido a que la naturaleza de un proceso de simulación dirigida por eventos, es de carácter totalmente secuencial, los diagramas de estado no son muy representativos ya que las clases modeladas no poseen una gran cantidad de estados (se usan mas en sistemas concurrentes y/o interactivos). En cambio, si se a optado por usar diagramas de actividades que nos representarán los pasos y tareas que se van ejecutando secuencialmente durante el proceso de simulación y algunos diagramas de secuencia para indicar las clases que intervienen y los datos que se manejan con más detalle en una actividad concreta.

- Capítulo 5: Implementación actual del simulador

  Se describen en este capítulo las decisiones que llevaron a la implementación actual del simulador *PASS* (por ejemplo la elección del entorno de desarrollo o del lenguaje de programación). También se intenta dar una visión global de la implementación actual del simulador analizando en detalle cada una de las clases presentadas en los apartados anteriores (mediante fichas individuales).

- Capítulo 6: Líneas futuras y conclusión

  En este capítulo se analiza lo aprendido en este proyecto así como las posibles líneas futuras de desarrollo del proyecto y del simulador. Se muestra además como deberían abordarse las futuras modificaciones para modelar redes *WDM*, multietapa, añadir nuevas formas de interconexión o realizar gráficas con los resultados de la simulación.

# Capítulo 2
# Situación y contexto

## 2.1 Redes WDM

Con el crecimiento del tráfico en Internet esta aumentando la demanda de nuevos servicios y tecnologías sobre redes *IP*. Un ejemplo de esta convergencia es la telefonía *IP (VoIP).* El problema es que la mayoría de las redes en funcionamiento hoy día fueron diseñadas para transportar principalmente telefonía, y se hace necesario cambiar la infraestructura de las redes para transmitir principalmente *IP*.

Uno de los avances más importantes en el campo de las transmisiones ópticas ha sido la aplicación de la multiplexación por división en longitud de onda *(WDM*) a enlaces de fibra óptica. De esta forma se alcanzan tasas de transmisión del orden de 1,6Tb/s (40x40 Gb/s o 160x10 Gb/s) y se estima que la capacidad máxima de transmisión de la fibra impuesta por sus limitaciones físicas es de 50Tb/s. Así *WDM* [1][2][3] proporciona el ancho de banda que necesitan las diferentes aplicaciones: vídeo, audio, vídeo bajo demanda, servicios multimedia, etc.

El crecimiento de *WDM* va ligado a la mejora de sus componentes clave: fibras, amplificadores, láseres sintonizables y dispositivos de conmutación ópticos conformando así un escenario totalmente óptico (al menos en el *backbone*) tal y como se muestra en la figura 1.1. En este modelo encontramos por un lado una serie de *switchs* electrónicos (SE en la figura) que son los que dan acceso a la red óptica de alta velocidad a través de un enlace óptico (por ejemplo WDM), *routers* electrónicos multiprotocolo de gran capacidad (R) y por el otro lado una red totalmente óptica constituida por *switchs* ópticos (S).

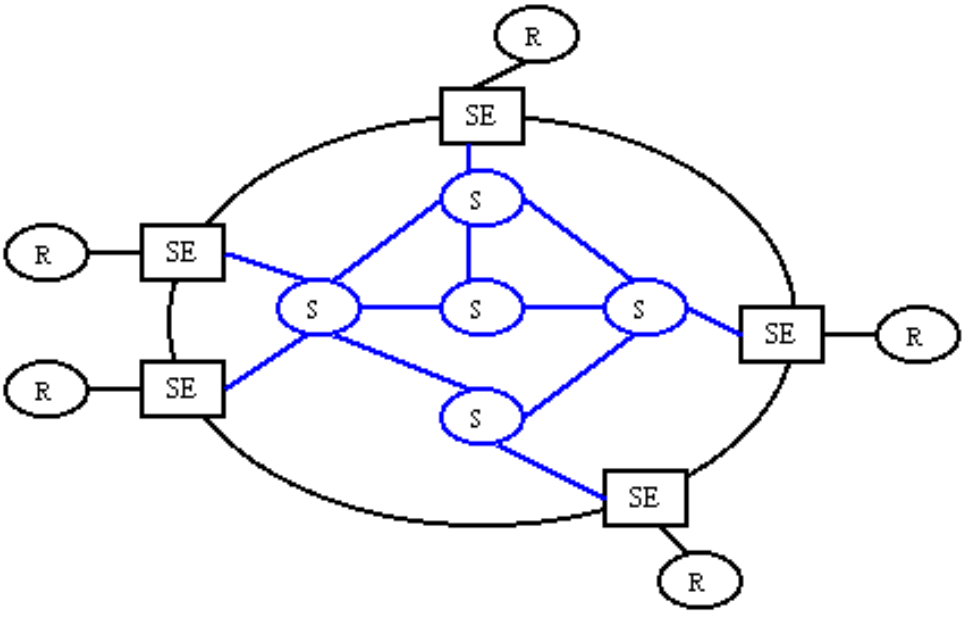


**Figura 2.1 Red óptica**

Han aparecido un sin número de nuevas tecnologías *(SONET/SDH, ATM, POS)* y *WDM* debe permitir transportarlas sobre la capa óptica. El problema surge en que muchas de ellas interoperan en distintos niveles del modelo de referencia *OSI* lo que complica la administración y mantenimiento de las redes. Más aun, muchas de las funcionalidades repiten en distintos niveles de la pila de protocolos y esto se resume en *overhead* innecesario.

Se hace necesaria una mayor integración entre la capa *IP* y la capa óptica. Las propuestas actuales se basan en la creación de una interfaz o capa de adaptación. Para ello existen tres tendencias. La tecnología más avanzada en estado comercial para el *backbone WDM* es la conmutación de circuitos *WDM*, basada en el establecimiento de circuitos virtuales, los cuales son los distintos canales *WDM*. A estos canales se los denomina "*lightpaths*", y transportan una señal óptica transparentemente entre dos nodos frontera sin conversión electrónica en nodos intermedios.

Existen dos conceptos asociados a este tipo de alternativa:

- Topología Virtual: Asignación de canales de acuerdo a distintos parámetros, como matriz de demanda, recursos en los nodos, etc.
- *RWA (Routing and Wavelength Assignment):* Se requiere para administrar eficientemente los recursos de la red, *BW*, números de canales, etc. A fin de optimizar el *throughput* y disminuir la probabilidad de bloqueo.

Hasta ahora se ha hablado de circuitos o "*lightpaths*", pero no se ha estudiado el tiempo de vida de dichos circuitos. El desarrollo de la tecnología *MEMS (Micro-Electronic-Mechanical System*) ha permitido mejorar los sistemas de conmutación de longitudes de onda.

La alternativa más ambiciosa, en estado de prueba, es la conmutación óptica de paquetes, que será introducida en el siguiente apartado.

# 2.2 Conmutación óptica de paquetes

La tecnología óptica de conmutación de paquetes es la tecnología con mayores perspectivas de futuro para dirigir el tráfico en Internet. Además se prevé que los dispositivos ópticos llegarán a ser mas competitivos económicamente con el tiempo, por lo que las redes ópticas de conmutación de paquetes *WDM* pueden jugar un importante papel en la redes de conmutación de alta velocidad. El tipo de función o aplicación que vaya a llevar a cabo la red óptica impondrá grandes restricciones a la conmutación óptica que realice, y nos llevará a elegir un diseño de red u otro.

Entre las diversas técnicas que se emplean en la conmutación *WDM* destacamos:

- Conmutación de circuitos *WDM* (con establecimiento de circuito virtual)
- Conmutación a ráfagas *WDM* (una cabecera por ráfaga)
- Conmutación de paquetes *WDM* (una cabecera por paquete)

A continuación mostramos una figura donde pueden apreciarse las diferencias entre ellas:

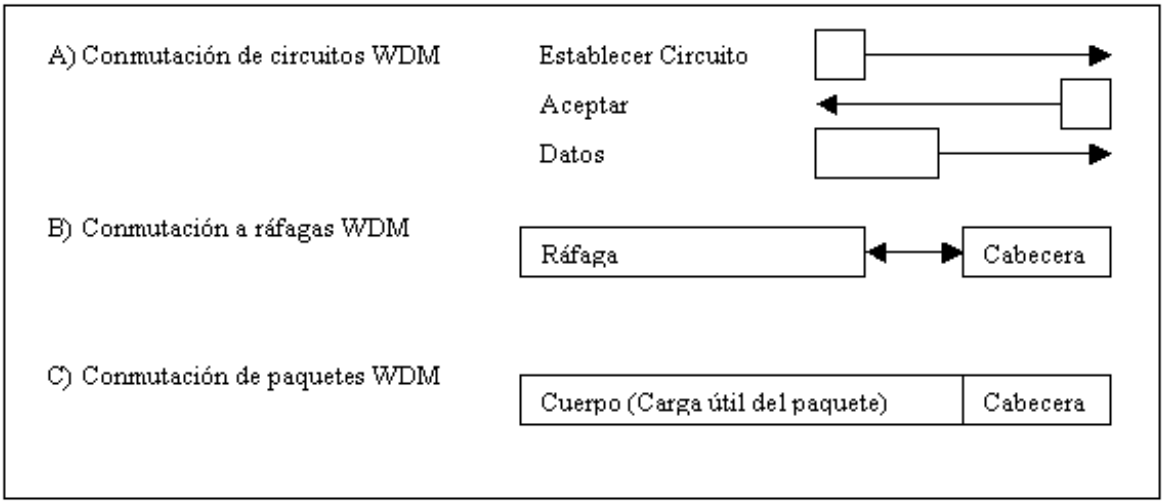


**Figura 2.2 Tipos de conmutación**

Nosotros nos centraremos en el caso de la conmutación de paquetes, ya que parece ser la evolución lógica en el desarrollo de la tecnología *WDM*. Como se ha mencionado anteriormente el principal problema para este tipo de conmutación de alta velocidad es el procesado de la cabecera y el diseño y la fabricación de los dispositivos de conmutación. Relacionadas con el procesado de la cabecera hay una serie de tareas relacionadas como decidir el puerto de salida, la admisión y control de la congestión, la administración de recursos o la actualización de la cabecera (como el valor del tiempo de vida en los datagramas *IP*). Este tipo de funciones siguen siendo desempeñadas por dispositivos electrónicos (después de la adecuada conversión de la interfaz de la línea) y se prevé que esto no cambie a corto-medio plazo. Éste procesado de la cabecera se realiza de forma individual en cada paquete y por las restricciones de tiempo impuestas por la alta velocidad de las redes ópticas se opta por circuitos integrados de aplicación específica (*ASICs*).

La funcionalidad de los *switchs* o conmutadores debe de ser doble: transmitir los paquetes desde los puertos de entrada hasta los puertos de salida y controlar los *buffers* si hay congestión.

Podemos agrupar los *switchs* en dos categorías: "ranurados" (síncronos) y "no ranurados" (asíncronos). En el caso de los asíncronos paquetes de distinta longitud llegan al *switch* en cualquier instante de tiempo y no alineados. En cambio en los conmutadores síncronos los paquetes tienen la misma longitud, y las funciones del *switch* (como la transmisión de una etapa a otra) se realizan con una señal de reloj (a un tiempo determinado). En el caso de las redes asíncronas al no haber un alineamiento de paquete no se necesita un fuerte control del tiempo y por eso este tipo de redes resultan más baratas de construir, aunque a costa de perder funcionalidad debido a la mayor posibilidad de congestión en las mismas. Los conmutadores síncronos han sido ampliamente estudiados para conformar un escenario de conmutación totalmente óptico, y por ello serán importante objetivo de estudio en nuestra herramienta *PASS*.

## 2.3 Simuladores de sistemas de conmutación multietapa

Los conmutadores multietapa que se fabrican hoy en día están formados usando bloques o elementos de conmutación que se alinean es diferentes etapas y se conectan según una topología dada. Esta es la estrategia que se ha adoptado para la construcción de conmutadores *ATM* de gran escala, para evitar la complejidad propia del crecimiento cuadrático de los conmutadores monoetapa (*crossbar*).

En el caso general, cada elemento de conmutación del SCM tendrá un *buffer* para tratar los problemas de congestión. De forma que la memoria total del sistema se haya distribuida en cada uno de los elementos que conforman las etapas, lo cual supone una buena aproximación al marco óptico. Por lo que describiremos la arquitectura de estos sistemas dada la similitud que poseen con las redes "ranuradas" o síncronas y será modelada en nuestra herramienta de simulación.

En los SCM la necesidad de un enrutado interno de los paquetes a través de las diferentes etapas aumenta. Por ejemplo en un conmutador *ATM* de 3 etapas cada celda puede conmutar a cualquier elemento de la segunda etapa independientemente del elemento de procedencia. Así para clasificar los SCM *ATM* nos fijaremos en la forma en la que se toman las decisiones de encaminamiento: sistemas dinámicos y estáticos. Para los primeros cada celda es enrutada independientemente de su circuito virtual (CV) preestablecido por las celdas anteriores, y para los segundos el CV establece la ruta que seguirá el paquete de forma que todos los paquetes de un CV seguirán dicha ruta. En sistemas de enrutado estático, un procesador controla la creación de los CV asignando la ruta con menos recursos usados. La principal ventaja de los SCM dinámico radica en la total libertad de dividir el tráfico de entrada en diferentes rutas optimizando así el uso de la red, pero tienen un inconveniente, y es que cada ruta puede introducir un retardo diferente con lo que en una conexión establecida los paquetes no llegarían en el orden adecuado y puede ser necesario algún mecanismo de resincronismo de las celdas.

Una de las limitaciones que podemos encontrar en las herramientas de simulación ATM viene determinada por la capacidad de proceso, ya que al trabajar con probabilidades del orden de $10^{-10}$ de perdida de celdas o de otros eventos de interés, se requiere un mayor esfuerzo de simulación (aplicando el método de Montecarlo). Para llegar a la solución del problema se han seguido diferentes aproximaciones: ejecución en dispositivos hardware que desarrollan tareas específicas, tratamiento estadístico de eventos o ejecuciones en paralelo/distribuidas. Un ejemplo de esta primera aproximación es *FAST* (Simulación *ATM* ejecutada sobre *FPGA*) desarrollada por la universidad de California. Es decir una *FPGA* es usada para implementar los componentes clave de la simulación como la generación de tráfico o la recolección de estadísticas. Técnicas estadísticas de tratamiento de eventos tales como el muestreo es el segundo método alternativo, donde es necesario un tratamiento diferente para estimar retardos, probabilidades de pérdida, y longitudes de la cola. Por último, la ejecución distribuida o en paralelo se presenta como una oportunidad interesante, pero limitaciones inherentes a la linealidad de la ejecución tales como la conmutación de paquetes la convierten en una opción relativamente costosa. La solución ideada para el simulador *PASS* se enmarca en el entorno *HTC High Throughput Computing).* Ya que en una universidad como la Politécnica de Cartagena donde todos los laboratorios poseen muchas estaciones de trabajo, muchos de los ciclos que no usan dichas estaciones pueden ser usados con propósitos de simulación. Para aprovechar dichos periodos inutilizados el sistema *CONDOR (HTC)* [4][5] se instalaría en diferentes estaciones de trabajo, de forma que la aplicación distribuida de simulación se

comunicaría con *CONDOR* para controlar las tareas de la simulación y el ensamblado de resultados. El hecho de que las simulaciones *PASS* están generalmente compuestas de diferentes experimentos independientes, variaciones del tráfico o carga, o simplemente del tamaño del *buffer* nos permite determinar una política adecuada que permita realizar diferentes experimentos de una forma distribuida. De forma que no sea necesaria una comunicación entre los diferentes módulos y no es necesario código específico de sincronización, proporcionando una solución simple al tiempo que eficiente.

Así pues la simulación por software ha sido la herramienta más utilizada para realizar el diseño de los distintos SCM. Lo ideal es una herramienta que proporcione el equilibrio necesario entre la funcionalidad de un sistema específico enfocado al modelado de SCM y que al mismo tiempo proporcione la flexibilidad necesaria para configurar una gran cantidad de parámetros de simulación por parte del usuario.

A continuación mostraremos algunos ejemplos de otros simuladores y como se han modelado:

## 2.3.1 Herramientas de propósito general

En este apartado agruparemos aquellas herramientas que proporcionan un entorno en el cual poder desarrollar y llevar a cabo la simulación. Suelen ser de libre distribución lo cual es una ventaja, pero tienen el inconveniente de que antes de poder llevar a cabo la simulación deseada debemos familiarizarnos con el entorno de desarrollo y adaptar el mismo a nuestras exigencias o tipo de simulación (conmutación en nuestro caso). Algunas herramientas de este tipo son:

### 2.3.1.1 Ptolemy

El proyecto *Ptolemy* [6] pretende modelar, simular, y diseñar sistemas heterogéneos, concurrentes, y en tiempo real. El objetivo principal es poder construir componentes concurrentes, de forma que se vuelve de vital importancia el uso de modelos bien definidos para disminuir el cómputo relativo a la interacción entre los componentes. Por lo tanto se optó por un lenguaje orientado a objetos, en concreto Java.

Los sistemas tratados en este proyecto suelen ser sistemas reactivos. Los sistemas reactivos son aquellos que interactúan adaptándose a la velocidad de su entorno. Son a menudo sistemas encajados, y en contraste con los sistemas interactivos, que interactúan con el entorno pero a su propia velocidad, y los sistemas transformacionales, que procesan un conjunto de datos de entrada para producir un conjunto de datos de salida. Los sistemas reactivos incluyen típicamente elementos de procesado de señal, de comunicaciones, y de control en tiempo real. Se suelen poner en ejecución usando diversas tecnologías, posiblemente incluyendo software encajado, hardware digital, hardware configurable, circuitos analógicos, circuitos de microondas, y sistemas microelectromecánicos (*MEMS*).

Uno de los principios claves en el proyecto *Ptolemy* es el uso de múltiples modelos de cómputo en un entorno de desarrollo heterogéneo y jerárquico. Una de las premisas de este proyecto es que no existe un modelo de cómputo para fines genéricos siendo posible en un futuro cercano adaptarse a lo que necesiten los diseñadores.

El proyecto apunta a desarrollar técnicas de modelado heterogéneo, incluyendo “meta-modelos” y un laboratorio de software para experimentar con dichos modelos. En este contexto, ha explorado los métodos basados en redes del flujo de datos y de proceso, sistemas de tiempo discreto o basado en eventos, lenguajes síncronos/reactivos, máquinas de estado finito, y procesos secuenciales en las

comunicaciones. Extendiendo desde la semántica fundamental hasta la síntesis del software y del hardware más comunes.

Basándose en las especificaciones anteriormente mencionadas las decisiones de diseño que se adoptaron fueron las siguientes:

- Uso de conceptos de lenguajes de programación tales como semántica, tipos de datos, y concurrencia en el diseño de sistemas electrónicos.
- Enfocar el modelado de forma que el diseñador solo se tenga que preocupar por el problema a modelar y no en la herramienta.
- Énfasis en la comprensión del sistema global, que es promovida por las representaciones visuales, los modelos ejecutables, y la verificación.
- El uso de *Java*, y de los patrones de diseño *UML*
- El software de *Ptolemy* proporciona un laboratorio para el lado experimental del proyecto. Apoya la interacción de diversos modelos de cómputo usando los principios del software orientado a objetos tales como el ocultamiento de la información y el

Por ultimo mencionar que *Ptolemy* se ha empleado en sistemas de comunicación óptica (nuestro caso), sistemas en tiempo real así como en laboratorios de procesado de señal y cursos de telecomunicaciones.

Concluyendo: *Ptolemy* es un entorno de trabajo que nos permite llevar a cabo simulaciones de sistemas heterogéneos. Su interfaz gráfica permite una fácil modificación y configuración, pero conceptos como enrutado y algoritmos de encaminamiento no se encuentran incluidos limitando su aplicación en el campo de la investigación.

#### 2.3.1.2 OMNET

*OMNeT++* [7] *(Objective Modular Network Testbed in C++)* es un simulador orientado a objetos, modular y basado en eventos. El simulador se puede utilizar para:

- Modelar el tráfico en redes de telecomunicaciones
- Modelar protocolos
- Modelado de colas
- Modelar sistemas multiprocesador y otros sistemas de hardware distribuido
- Validar arquitecturas hardware
- Evaluar aspectos relacionados con el rendimiento en sistemas complejos de software
- Modelar cualquier otro sistema donde se posible una aproximación basada en eventos

Un modelo de *OMNeT++* consiste en un conjunto de módulos anidados jerárquicamente. La profundidad del anidamiento del módulo no esta limitada, lo que permite que el usuario refleje la estructura lógica del sistema real en la estructura del

modelo. Los módulos se comunican mediante paso de mensajes. Los mensajes pueden contener arbitrariamente estructuras de datos complejas. Los módulos pueden enviar mensajes directamente a su destino o a lo través de una ruta predefinida, mediante puertas y enlaces.

Además a los módulos se les pueden pasar parámetros con tres funciones principalmente:

- Para modificar el comportamiento del módulo
- Para crear una topología flexible del modelo (de forma que los parámetros pueden especificar el número módulos, la forma de realizar las conexiones, etc.)
- Para las comunicaciones entre módulos por variables compartidas.

Los módulos conforman el nivel más bajo de la jerarquía del modelo que debe de proporcionar el usuario, además estos contienen los algoritmos del modelo. Durante la ejecución de la simulación, los módulos simples funcionan en paralelo, ya que cada uno se ejecuta como un *thread* o proceso ligero. Para escribir los módulos simples, el usuario no necesita aprender un nuevo lenguaje de programación, solo se asume tener cierto conocimiento de *C++*.

Las simulaciones de *OMNeT++* ofrecen diversas interfaces para diversos propósitos: depurar, realizar demostraciones o ejecuciones punto por punto. Se proporcionan además interfaces para el usuario experimentado que hacen el interior del modelo visible al usuario, permitiendo detener o proseguir la simulación e intervenir cambiando variables o objetos dentro del modelo. Esto es muy importante en la fase de desarrollo/depurado del proyecto de la simulación. El simulador así como las interfaces y las herramientas utilizadas es portable: pueden trabajar en *Windows* y en varios entornos de *Unix*, usando varios compiladores de *C++.*

### 2.3.1.3 Network Simulator de Berkeley

El *NS* [8] es un simulador basado en eventos desarrollado en *UC Berkeley* que simula una gran variedad de redes *IP*. Implementa protocolos de red tales como *TCP* y *UDP*, modela fuentes de tráfico tan diversas como *ftp, telnet, web, CBR y VBR,* políticas de colas, algoritmos de encaminamiento tales como *Dijkstra*, y más. El *NS* también implementa *multicasting* y protocolos de la capa *MAC* (control de acceso al medio) para las simulaciones de redes *LAN*. El *NS* forma parte del *"VINT Proyect"* [9] que además desarrolla herramientas para mostrar los resultados de la simulación, desarrollar análisis previos a la misma y convertidores de las topologías de la red generadas a un formato compatible con *NS*.

Actualmente, *NS* (versión 2) está escrito en *C++* y *OTcl* (lenguaje desarrollado por el *Massachusetts Institute of Technology*, básicamente *Tcl* con las extensiones necesarias para hacer de él un lenguaje orientado a objetos).

Como se muestra en la figura 2.3 el *NS* es un intérprete de *OTcl* (*Tcl* orientado a objetos) que tiene un manejador de eventos (de simulación), librerías de objetos (componentes de red y módulos de interconexión). De forma que para configurar y poder llevar a cabo la simulación de una red el usuario debe programar un *script OTcl* que arranque el manejador de eventos, y establecer la topología de la red mediante los objetos y funciones de interconexión que poseen las librerías. Una de las facilidades que ofreces es que cuando un usuario quiere crear un nuevo objeto, puede

escribirlo directamente o construirlo a partir de los objetos que posee la librería, posteriormente solo deberá realizar la interconexión.

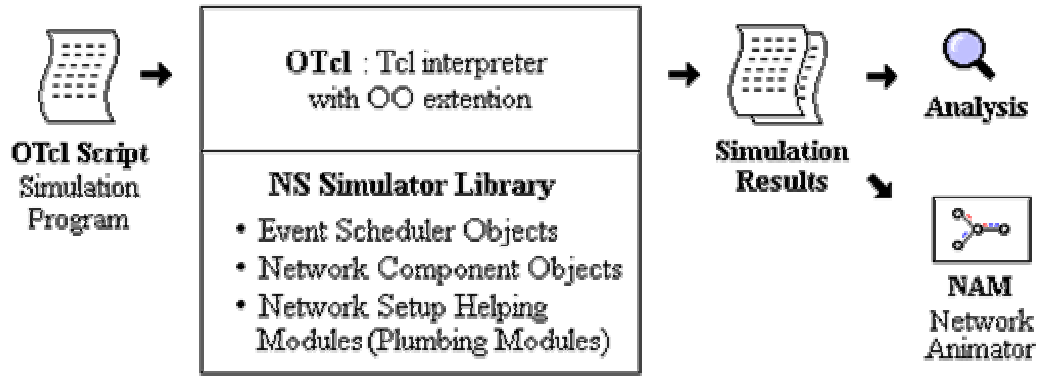


**Figura 2.3 Vista simplificada del usuario del *Network Simulator***

Otro de los componentes más importantes del *NS* es el manejador de eventos. Un evento en *NS* en un identificador que es único para cada paquete, que posee un tiempo concreto y un puntero al objeto que maneja el evento. De forma que el manejador de eventos controla el tiempo de simulación y lanza los eventos (sacándolos de la cola) en el momento apropiado invocando al componente de red necesario. Por otro lado la comunicación entre los diferentes objetos de la red no consume tiempo de simulación. De forma que todos los componentes de la red que necesitan introducir un retardo o dejar pasar un tiempo de simulación (retardo de propagación, modelado del retardo de conmutación etc) usan para ello el manejador de eventos

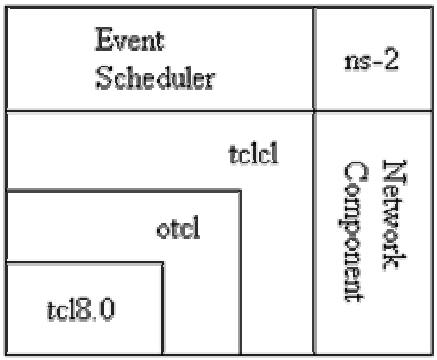


**Figura 2.4 Arquitectura del *Network Simulator***

La figura 2.4 muestra la arquitectura general del *NS*. En esta figura el usuario por defecto se encontraría en la esquina abajo izquierda, diseñando y ejecutando simulaciones en *Tcl* usando los objetos del simulador de la librería *OTcl*. Los manejadores de eventos y muchos de los componentes de la red están implementados en *C++* y están disponibles para *OTcl* a través de un enlace implementado en *tclcl.*

Después de haber descrito la estructura general y la arquitectura del *NS* explicaremos como obtener los resultados de la simulación. Como se muestra en la figura 2.3 cuando la simulación finaliza *NS* genera uno o más archivos de texto de salida que contienen los datos en detalle de la simulación. Estos datos pueden usarse

para realizar un análisis de la simulación o como parámetros de entrada para una herramienta de simulación gráfica llamada *NAM (Network Animator)* que también forma parte del *"VINT project".*

## 2.3.2 Herramientas de propósito específico

Aquellas programadas para simular un sistema concreto. Mucho más especializadas. El simulador que modelaremos *(PASS)* está incluido en este grupo. Otros ejemplos de simuladores de este tipo son:

### 2.3.2.1 SIGLA: Simulador global para redes ATM

*SIGLA* [10] es un simulador global para redes *ATM* que usa programación orientada a objetos y que permite una ejecución distribuida en varios sistemas mono y multiprocesadores que estén conectados por medio de una red local. Además se pretende que existan versiones modulares simplificadas del programa global que funcionen en otras plataformas, como por ejemplo en *PC*. Aunque la versión actual solo funciona bajo el sistema operativo *Linux*. Esta implementado en *C++* y bajo licencia *GNU*.

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 *ATM* 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 *ATM* 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.

### 2.3.2.2 VISTA

*VISTA* [11] es una herramienta para la visualización y simulación de conmutadores *ATM* claramente diseñada con propósitos educativos. Implementada en *C*, se puede configurar para visualizar el funcionamiento de varias arquitecturas de conmutación. *VISTA* proporciona dos arquitecturas de conmutación para la simulación (*Tándem* y *shared bus*), un modelo de generación de tráfico parametrizable, tres políticas para los *buffers* (desechar al pulsar, *buffer* intermediario compartido, y descarte selectivo), y una interfaz de usuario gráfica interactiva. Además un usuario puede agregar arquitecturas adicionales de conmutación a través del interfaz configurable que la herramienta proporciona.

A través de la interfaz de *VISTA*, un usuario puede visualizar el funcionamiento de la arquitectura de conmutación. Las medidas de funcionamiento tales como la pérdida de celdas en los puertos de salida en función del tiempo, retardo medio de celdas en el conmutador, trafico medio, y la probabilidad de perdida de celda pueden ser visualizados gráficamente. *VISTA* también permite al usuario simular la transmisión de ficheros de datos, tales como imágenes, a través del conmutador durante la simulación para determinar la calidad del servicio en función de las pérdidas de celdas. Además, *VISTA* permite que el usuario haga una película comprimida en *MPEG* de la visualización que puede ser vista a diversas velocidades y de esa forma comparar el rendimiento entre varias arquitecturas.

# 2.4 Introducción a UML

*UML (Unified Modeling Language)* es un lenguaje para especificar, construir, visualizar y documentar los sistemas software. Con *UML* [12][13] se tiene un lenguaje visual para el modelado, pero no un lenguaje visual de programación, es decir, a partir de él no se deriva el código en algún tipo de lenguaje de programación. Algunos elementos del software (bucles, saltos) quedan mucho mejor expresados con el propio código fuente, pero la arquitectura y estructura del sistema se puede expresar y comprender perfectamente a partir del lenguaje *UML*. Con esta metodología se cubrirán las etapas de desarrollo software de análisis, diseño, programación y testado.

Haciendo una breve historia, se puede decir que el desarrollo de UML comenzó en octubre de 1994 cuando Grady Booch y Jim Rumbaugh de *Rational Software* comenzaron a trabajar en la unificación de los lenguajes de modelado *Booch y OMT*, es entonces cuando fueron reconocidos mundialmente a la cabeza del desarrollo de metodologías orientadas a objetos. Fue entonces cuando terminaron su trabajo de unificación obteniendo el borrador de la versión 0.8 del denominado *Unified Method* en octubre de 1995. Tras esto también en 1995, Ivar Jacobson padre del la metodología OOSE se unió con *Rational Software* para obtener finalmente *UML* 0.9 y 0.91 en junio y octubre de 1996. Tras esto muchas organizaciones como *Microsoft, Hewlett-Packard, Oracle, Sterling Software MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjectTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam.* se asociaron junto con *Rational Software* para dar como resultado *UML* 1.0 y UML 1.1. llegando hoy en día hasta *UML* 1.4 y *UML* 2.0. En la Figura siguiente se muestra de forma gráfica y resumida la evolución de *UML*.

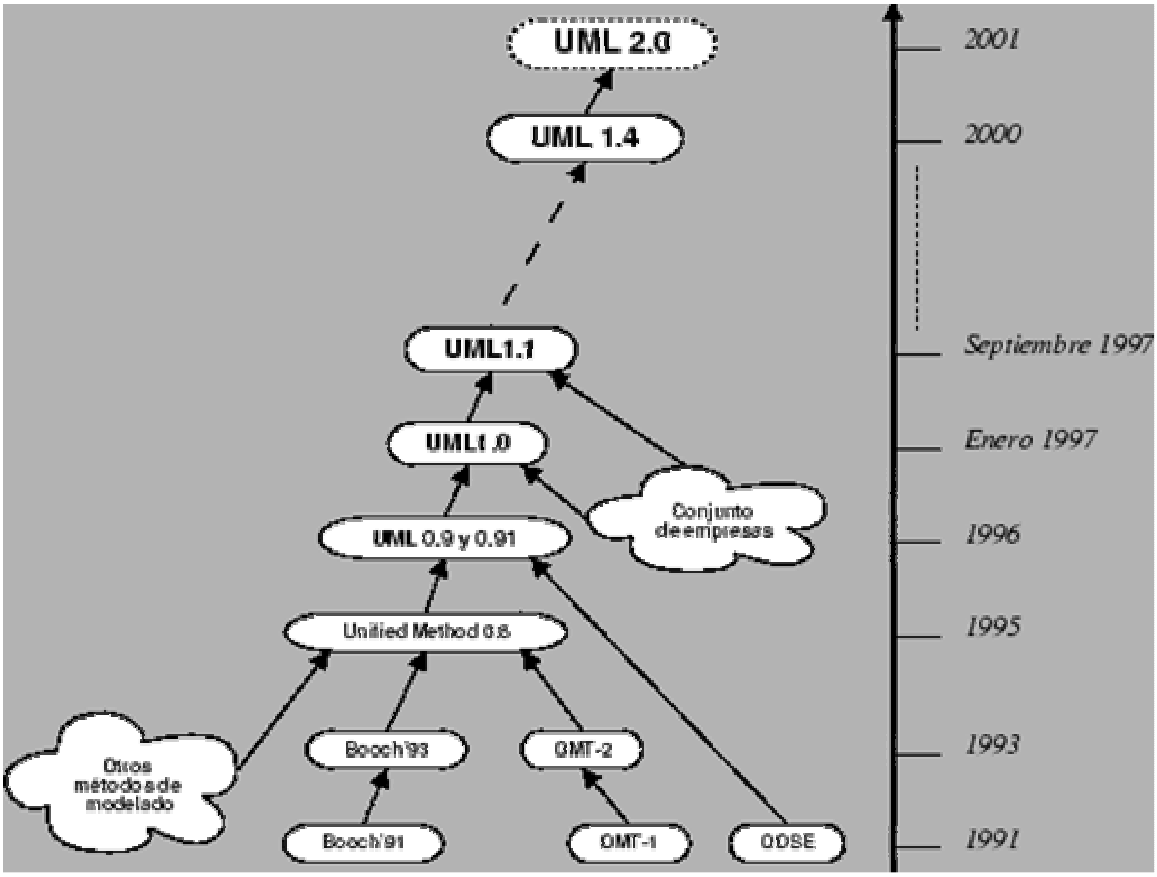


**Figura 2.5 Fases de UML**

Son muchas las organizaciones que han incorporado *UML* como un estándar en el desarrollo de sus procesos y productos, teniendo como gran ventaja que al ser no propietario está abierto a su uso en la comunidad científica. Durante el desarrollo del capítulo se utilizarán una serie de elementos gráficos que componen la metodología *UML*. Estos elementos gráficos de la metodología, pretenden aportar un lenguaje de descripción común, y fácil de entender, entre todas las personas involucradas en un proyecto. Este es uno de los objetivos del uso de *UML*: conseguir intercambiar información sin invertir gran esfuerzo en estas tareas, puesto que se comparte una determinada notación. *UML* dispone de diferentes construcciones para representar gráficamente el sistema. Estas construcciones se denominan vistas, las cuales, intentan resaltar determinadas características que componen el software. Durante las etapas del ciclo de vida se utilizarán aquellas que se consideren necesarias según la naturaleza del software que se pretenda modelar, por ejemplo:

- Diagrama de casos: Para realizar la descripción del sistema desde un punto de vista de alto nivel. Éstos describen la funcionalidad de un sistema siguiendo una metodología descendente, es decir, partiendo de una visión de alto nivel del sistema y continuando con una descomposición del mismo en partes más concretas. El propósito de un caso de uso es definir una pieza de comportamiento coherente, sin revelar la estructura interna del sistema. La notación consiste en círculos que representan tareas que se realizan unidos entre sí mediante líneas con dos tipos de cualificación:
  - *uses:* Representa una relación de inclusión, es decir, una tarea que incluye a otras determinadas tareas.
  - *extend:* Representa una relación de extensión. Se pueden tratar estás, como diferentes formas de realización o especializaciones de dicha tarea.
- Diagrama de objetos: Este diagrama representa los aspectos estáticos del sistema a desarrollar. Estos aspectos estáticos modelan características del software como pueden ser la estructura interna, tanto del mismo código fuente como lo que es más importante, la representación que se hará de la información en el sistema informático. Éste suele ser el núcleo de todos los desarrollos software, ya que un buen modelado produce un software de buena calidad, por lo que se debe emplear gran esfuerzo en esta tarea. Este diagrama estará formado por objetos, representados con cajas cuadradas, y enlazados entre sí. Estas relaciones junto con los objetos tienen como objetivo conseguir, en el mayor grado posible, una abstracción de la realidad que pretendemos representar, que abarque en su totalidad los elementos y características del sistema que pretendemos representar.
- Diagramas de estados y Diagramas de Actividad: *UML* provee una serie de mecanismos para modelar el comportamiento dinámico de un sistema, como son los diagramas de estado y diagramas de actividad. La descripción se va a centrar en los diagramas de actividad ya estos se suelen usar cuando el comportamiento de un sistema no depende en gran medida de eventos externos, se requiere un flujo de objetos/datos entre los diferentes pasos, es decir, modela sistemas con un fuerte componente secuencial. Es evidente que la herramienta de simulación que se está modelando es de naturaleza totalmente secuencial. En cambio, los diagramas de estado nos describen mediante máquinas de *mealy* ó *moore* el comportamiento de las diferentes clases que

modelamos, las cuales cambian de estado al recibir diferentes eventos. La notación *UML* de los diagramas de actividad consiste en un cuadro dividido en una serie de columnas, cada columna representa un objeto/clase, de forma que cuando éste realiza alguna actividad se incluye en un cuadro la descripción de la actividad que realiza. El resultado de una actividad es un conjunto de datos, estos se indican en un cuadro en la frontera entre dos columnas, para indicar que esos datos resultantes fluyen de una clase a otra. Todo el diagrama está unido por flechas, indicando así el orden en el que se van realizando las diferentes actividades.

Una vez descritos los componentes básicos la metodología, en los sucesivos apartados del capítulo dicha metodología será utilizada para describir todas las fases del ciclo de vida del desarrollo de *PASS*. Es necesario indicar que tradicionalmente, este ciclo de vida se compone de las siguientes fases, a las cuales se han asociado determinados elementos de *UML*:

- Análisis global o conceptualización: Es una primera fase en la que el principal objetivo es marcar las pautas del problema a resolver. Se suelen realizar conjuntos de requisitos, tareas, resultados, etc. que se pretende que el sistema pueda realizar.
- Análisis de requisitos: Es una etapa en la que se pretende organizar los requisitos de la primera fase de forma ordenada y refinada, con la finalidad de poder realizar un análisis de los mismos para detectar posibles inconsistencias, omisiones, redundancias, etc.
- Diseño del sistema: Se obtendrá una visión global del sistema que queremos desarrollar desde un punto de vista de alto nivel. En esta fase se ha han usado los diagramas de casos.
- Diseño de objetos: Esta fase obtiene como resultado un modelo de objetos que podrá ser en la siguiente fase implementado. Este modelo de objetos debe representar la realidad que se desea abstraer en el sistema informático fielmente. Esta fase ha usado los elementos de *UML* denominados diagrama de clases, diagramas de estado y diagrama de actividades.
- Implementación: Es necesario por último decidir el lenguaje en el que se va a implementar *PASS*. Para ello se mostrarán las características o facilidades de las que disponemos en determinados lenguajes para declinarnos finalmente por un determinado lenguaje

## 2.5 Introducción al simulador PASS

Actualmente la aplicación *PASS (Packet Switch Simulator)* es una herramienta orientada a objetos diseñada para la evaluación de conmutadores síncronos trietapa. Algunas de sus características principales son:

- Diferentes elementos de interconexión, o de conmutación que pueden ser combinados y testados.
- Simulación de diversos tipos de tráfico.
- Posibilidad de añadir diferentes políticas de enrutado en un futuro (actualmente la ruta interna de los paquetes se genera de forma aleatoria)
- Ordenamiento de paquetes
- Un código limpio y ordenado definido mediante interfaces en *C++* [14] [15]. Para lo cual se usó el entorno de desarrollo *Kdevelop* [16] proporcionando un diseño abierto fácilmente extensible (diseñado bajo licencia *GNU* [17]) pero que al mismo tiempo permite una exhaustiva evaluación de los diferentes tipos de conmutadores actuales.

## 2.6 Análisis global de la arquitectura del simulador PASS

El simulador *PASS* fue diseñado siguiendo la arquitectura de las redes de paquetes, permitiendo en un futuro la evaluación de diferentes estrategias de interconexión (actualmente la interconexión es del tipo *fullconnect*), para diferentes tipos de elementos de conmutación y políticas de enrutado, con especial interés en la medición de retardos y probabilidades de pérdida. Una arquitectura flexible, y claros mecanismos de extensión hacen posible la evaluación de todos los tipos de elementos de conmutación de paquetes actuales así como de los que aparezcan en un futuro.

Una de las primeras decisiones de diseño fue la elección del paradigma orientado a objetos (OO), de forma que elementos físicos reales tales como "red de interconexión", "paquete", "cola", "memoria de resincronismo", "etapa", "algoritmo de enrutamiento" o "elemento de conmutación" sean específicamente modelados. Por ello se eligió *C++* como lenguaje de programación, ya que además de ser un lenguaje orientado a objetos tiene la ventaja de la optimización y el rendimiento frente a otros lenguajes del mismo tipo (por ejemplo *Java*). Siguiendo esta arquitectura las extensiones del sistema serán implementaciones de la interfaz adecuada, como veremos posteriormente. La arquitectura global del sistema se muestra en la figura 2.6 en la que puede apreciarse dos módulos claramente diferenciados: el generador de tráfico (con las fuentes y los multiplexores de tráfico) y la red de conmutación (formada por etapas, elementos de conmutación y la red de interconexión).

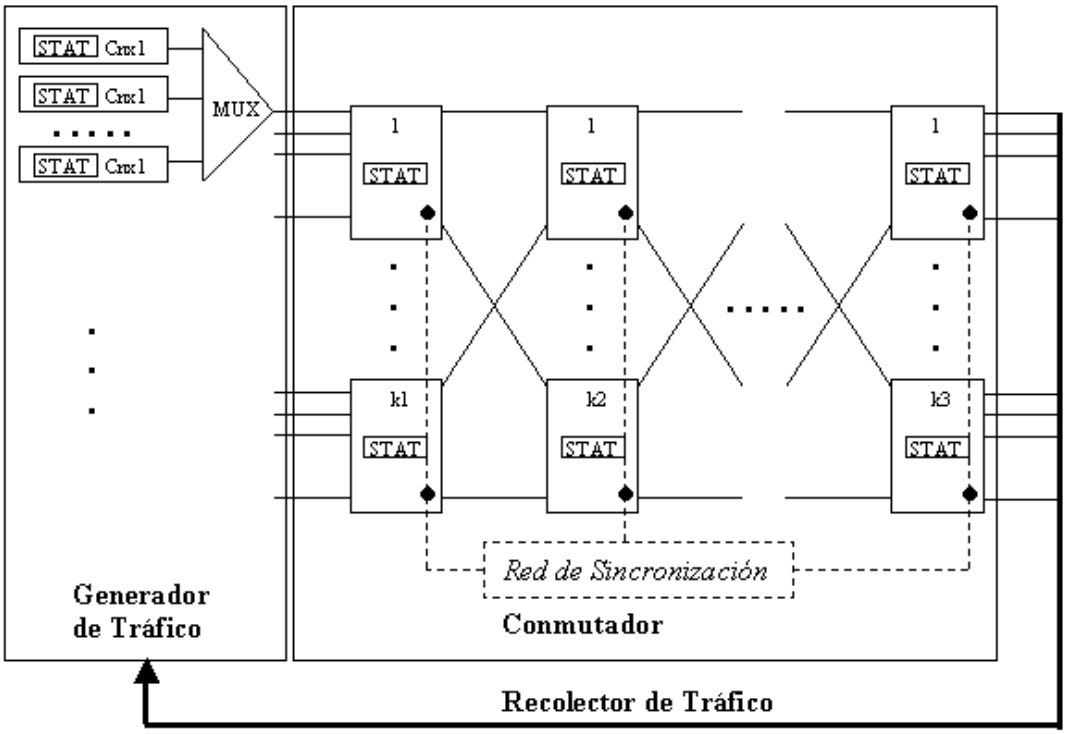


**Figura 2.6 Esquema general del simulador *PASS***

# Capítulo 3
# Modelado estático del simulador

A lo largo de este apartado se mostrará una visión arquitectural y modular de PASS. Siguiendo la metodología UML, el apartado queda dividido en una primera parte en la que se muestra una visión modular mediante diagramas de casos, y una segunda parte en la que se realiza el modelado de los objetos, la cual da idea de cómo la herramienta trata la información necesaria para realizar sus tareas.

## 3.1 Descripción funcional

Para poder realizar la simulación es imprescindible la realización de otras subtareas dependientes entre si. Algunas de estas tareas serían:

- Leer los parámetros que el usuario a introducido en función del tipo de simulación, estos parámetros van desde el tipo y volumen de tráfico, hasta la arquitectura interna del conmutador (nº de etapas, elementos por etapa, interconexiones entre elementos, política de enrutado, etc.)
- Crear los elementos encargados de generar el tráfico y la red de conmutación.
- Aplicar el algoritmo de simulación al tráfico que atraviesa la red de conmutación.
- Recoger y presentar los resultados de la ejecución.

Así en el siguiente diagrama de casos de uso pueden apreciarse las relaciones de dependencia entre dichas tareas:

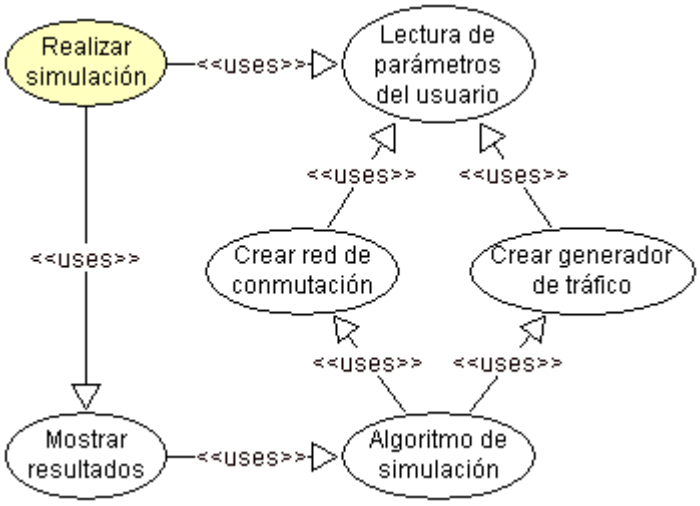


**Figura 3.1 Diagrama de casos de la tarea simulación**

Es importante resaltar que si una tarea dependediera de otra tarea que a su vez dependediera de una tercera, la primera tarea dependería indirectamente de la tercera. Por ejemplo en el esquema anterior para realizar la simulación seria necesaria la correcta creación de la red de simulación.

# 3.2 Modelo de objetos

En este apartado analizaremos como se ha llevado a cabo el modelado del simulador y veremos cuales son las clases principales que intervienen en dicha tarea (para mayor detalle sobre la estructura de dichas clases puede consultarse el capítulo 5 donde se adjunta una descripción de las mismas, así como sus variables y métodos principales).

## 3.2.1 Modelar la simulación

Tal y como vimos en el apartado anterior para llevar a cabo la simulación es necesario realizar algunas tareas tales como leer los parámetros del usuario, crear la red de conmutación y el generador de paquetes, realizar la simulación y presentar los resultados. Así pues tiene sentido que la clase encargada de modelar la simulación cumpla dichas tareas.

A continuación mostramos un diagrama estático de la clase *simulation*:

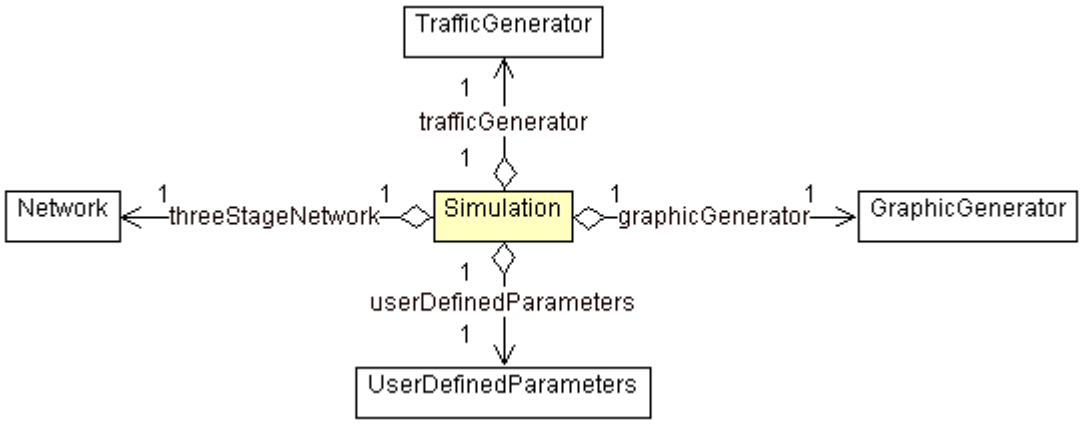


**Figura 3.2 Diagrama de clases de la simulación (*Simulation*)**

En el anterior diagrama además de la clase *Simulation* introduciremos la clase *UserDefinedParameter* que recogerá los parámetros introducidos por el usuario, la clase *Network* que modela la red de conmutación, la clase *TrafficGenerator* que será la encargada de generar el tráfico y la clase *GraphicGenerator* que presentará los resultados de la simulación en forma de gráficas..

## 3.2.2 Modelado del tráfico

Las clases encargadas de generar el tráfico han sido desarrolladas de forma independiente de la implementación de los diferentes elementos de conmutación. De forma que el mismo software puede ser reutilizado en otro simulador, los cambios de los elementos de conmutación no afectan a la generación del tráfico y los nuevos tipos de fuentes pueden ser fácilmente añadidas, inherentemente las estadísticas se calcularían de forma automática.

La estructura de dichas clases viene reflejada en el siguiente diagrama estático de clases.

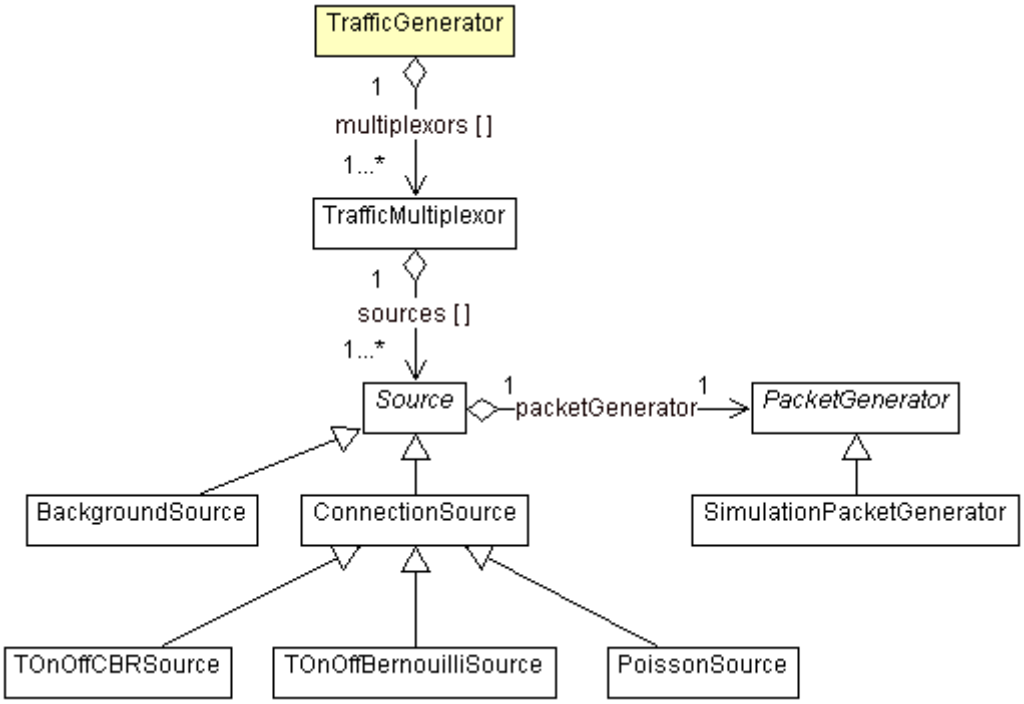


**Figura 3.3 Diagrama UML estático de todas las clases generadoras de tráfico**

Para poder llevar a cabo la simulación se deben realizar las conexiones necesarias con la red de conmutación, crear las fuentes generadoras de tráfico para cada puerto de entrada y unir dichas fuentes al objeto multiplexor de tráfico apropiado. De ello se encarga el módulo principal generador de tráfico (*TrafficGenerator*) que agrupa todos los multiplexores, y establece todas las conexiones necesarias (entradas y salidas) de la red multietapa. Además transmite la señal de reloj a los multiplexores y calcula diversas estadísticas a nivel global mediante las estadísticas parciales que le proporcionan los multiplexores tales como el número total de paquetes introducidos en el último tic, el retardo global de los paquetes o el tráfico (ofrecido, cursado y perdido) total acumulado.

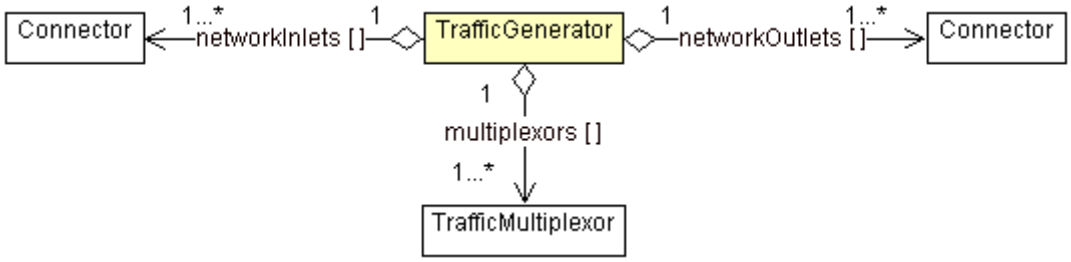


**Figura 3.4 Diagrama de clases del generador de tráfico (*TrafficGenerator*)**

Los multiplexores agrupan varias fuentes generadoras de tráfico. Se encargan de transmitirles la señal de reloj a dichas fuentes y de introducir los paquetes que ellas generan en la red de conmutación Para ello poseen una cola con los paquetes a introducir. Por último calculan diversas estadísticas con la información que les

proporcionas todas sus fuentes tales como el retardo de los paquetes en el multiplexor o el tráfico (ofrecido, cursado y perdido) acumulado del multiplexor.

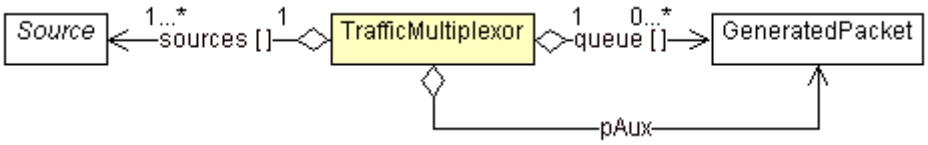


**Figura 3.5 Diagrama de clases de los multiplexores (*TrafficMultiplexor*)**

Las fuentes se encargan de la generación de los paquetes según el comportamiento que tienen definido y del cálculo de varias estadísticas. Dichas fuentes usan la interfaz *packetGenerator* (que proporciona su interfaz *source*) a la hora de crear un paquete. El diagrama de clases siguiente muestra los 2 tipos principales de fuentes diseñadas, que pueden ser combinadas para construir el escenario de tráfico deseado para la simulación:

- *Background source:* Su objetivo es simular un aumento del tráfico de múltiples fuentes con diferentes puertos de salida como destino. Estas fuentes se caracterizan por el tiempo que transcurre entre paquete y paquete y la distribución que sigue para decidir el puerto de salida. Las fuentes de este tipo realizan las estadísticas (probabilidad de pérdida de paquete, tráfico y estadísticas de retardo en media) de forma automática.
- *Connection source:* El objetivo de este tipo de fuente es simular un tráfico con un puerto de origen y longitud de onda fijado, así como un puerto de destino fijado, de forma que tendrán que tenerse en cuenta el desorden de los paquetes y las características del enrutado. Pueden crearse varios tipos de *conecction sources: TOnOffCBRSource* (con tráfico generado a ráfagas y espacio entre celdas constante), *TOnOffBernouilliSource* (con tráfico generado a ráfagas y espacio entre celdas aleatorio) y *PoissonSource* (en la que el tráfico se genera de forma aleatoria).

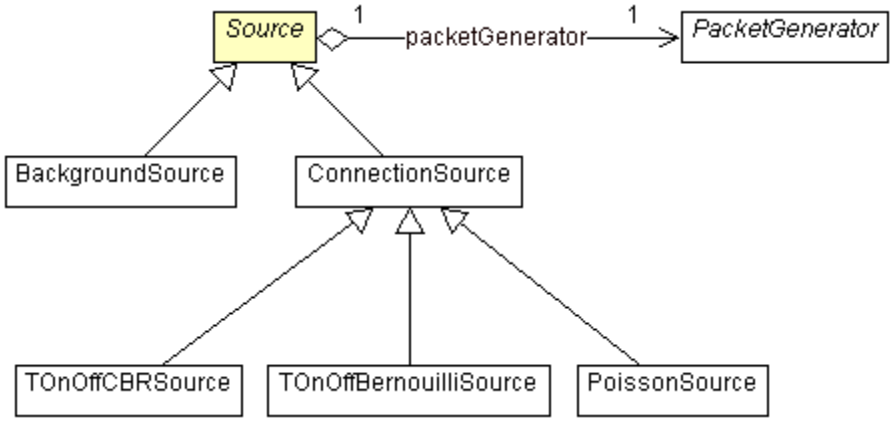


**Figura 3.6 Diagrama de clases de las fuentes (*Source*)**

El objeto encargado en última instancia de crear los paquetes es el generador de paquetes para la simulación (*SimulationPacketGenerator*) a través de su interfaz (*PacketGenerator*). Este objeto se encuentra vinculado a cada una de las fuentes. Es decir es la fuente la que establece la política o los criterios para generar los paquetes y es el generador de paquetes el que los crea y los devuelve a la fuente para que los transmita.

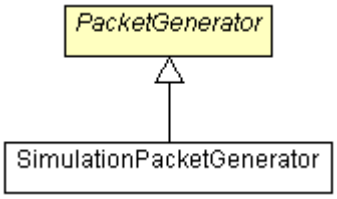


**Figura 3.7 Diagrama de clases del generador de paquetes (*PacketGenerator*)**

El generador de paquetes genera un tipo de paquete denominado "*GeneratedPacket*" del cual hereda "*SwitchedPacket*" que es el tipo de paquete que luego atraviesa el sistema de conmutación.

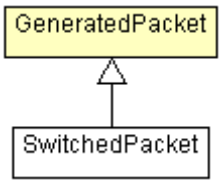


**Figura 3.8 Diagrama de clases de los paquetes**

## 3.2.3 Modelado de la red de conmutación

La arquitectura global del sistema de conmutación consiste en un conjunto de objetos interconectados que manejan de forma independiente la entrega de paquetes. Se ha optado por una implementación software donde los conceptos físicos fuesen claros. Aunque el diseño fue realizado teniendo en cuenta el multiplexado óptico por longitud de onda, se pueden construir y evaluar un sistema de conmutación como un modelo con una sola longitud de onda. Es importante mencionar que el propósito del simulador *PASS* no es la evaluación de las diferentes partes que componen un conmutador óptico, lo que se pretende es que mediante diferentes políticas de interconexión se defina la arquitectura interna del conmutador y con las medidas parciales de los elementos que lo componen tener una visión del funcionamiento del sistema global. La estructura de las clases que forman el sistema de conmutación es viene representada en el diagrama 3.9:

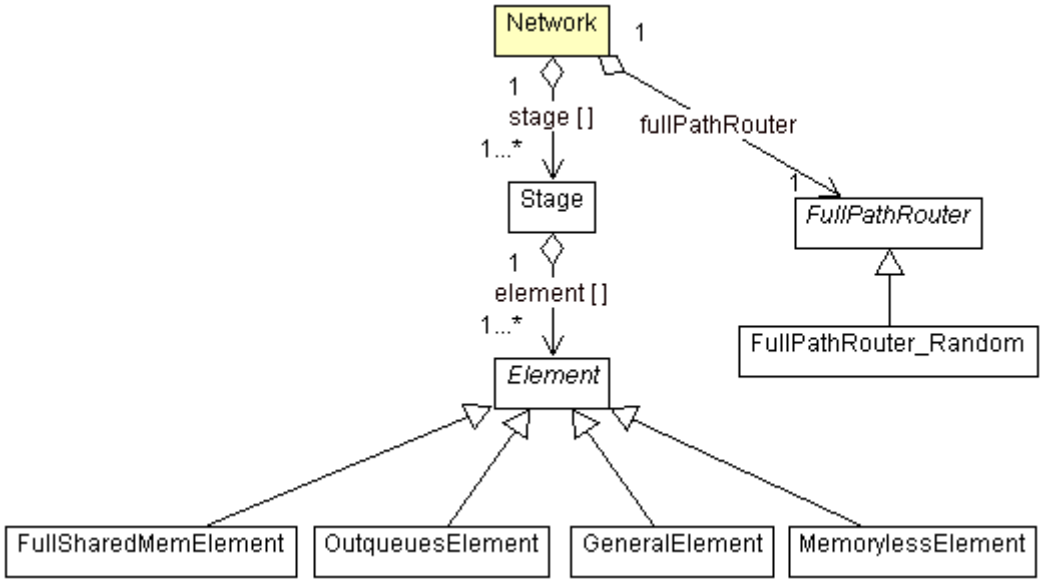


**Figura 3.9 Diagrama de clases de todo el sistema de conmutación**

El simulador *PASS* fue diseñado para conmutadores ranurados, que actúan de una forma sincronizada. Así pues la acción de conmutar debe ser gobernada por algún objeto que invoque el método de sincronización apropiado (tic o señal de reloj) en los objetos de la red. Para ello se creó la clase *Network*, encargada de modelar la red de conmutación. Esta clase además de gobernar la acción de conmutación agrupa todas las etapas de la red (y sus respectivos elementos de conmutación) así como el módulo encargado del encaminamiento (*FullPathRouter*).

En el siguiente diagrama *UML* estático de la clase *Network* podemos observar desde las etapas (*stages*) que forman la red de conmutación hasta el *FullPathRouter* o los conectores que son las entradas (*inlets*) y salidas (*outlets*) de la red multietapa:

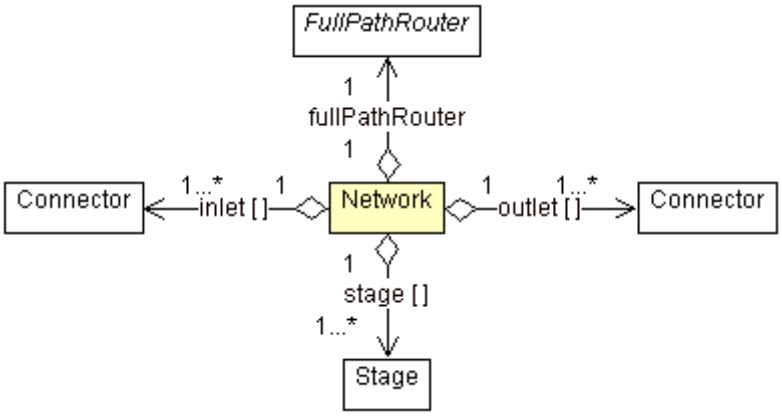


**Figura 3.10 Diagrama de clases de la red de conmutación (*Network*)**

Bajo la interfaz *FullPathRouter* se encuentran todos los objetos (actualmente solo *FullPathRouter_Random*) encargados de tomar las decisiones de encaminamiento de forma global. Los paquetes son tratados de forma independiente, y bajo esta interfaz se pueden implementar tanto estrategias de enrutado estáticas como dinámicas. Potencialmente cualquier tipo de conexión esta permitida, aunque la

implementación actual de *PASS* esta limitada a la interconexión total entre etapas, en un futuro el numero de etapas, el número de elementos por etapa y las longitudes de onda permitidas serán parámetros configurables.

La única implementación de dicha interfaz desarrollada por el momento la constituye la clase *FullPathRouter_Random*, objeto que genera la ruta que seguirá el paquete a través de las diversas etapas del conmutador de forma aleatoria y la almacena en la cabecera del paquete. Las clases mencionadas mantienen las siguientes relaciones:

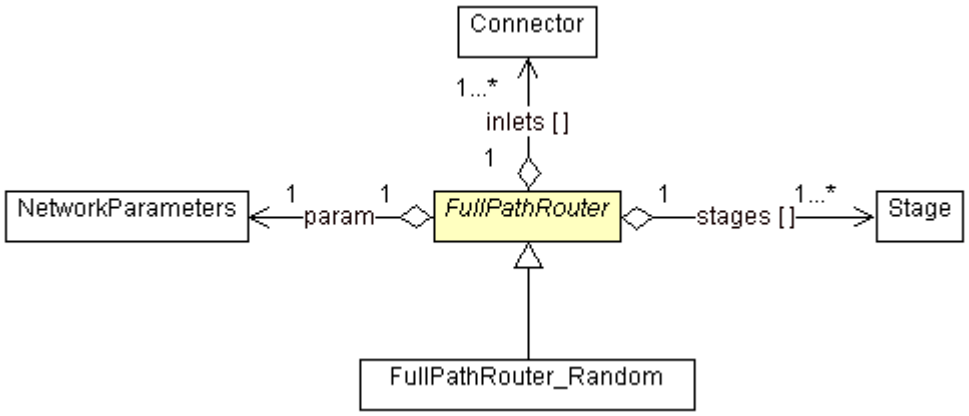


**Figura 3.11 Diagrama de clases (*FullPathRouter*)**

Tal y como podemos ver en el diagrama de clases que presentamos a continuación *Stage* es el objeto que modela una etapa de la red de conmutación, agrupando todos los elementos de conmutación pertenecientes a dicha etapa. Además cuando recibe la señal de sincronización marca el puerto de salida en los paquetes para la siguiente etapa y transmite la señal de reloj a sus elementos de conmutación.

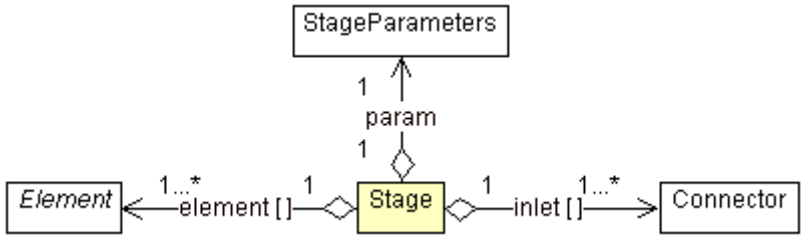


**Figura 3.12 Diagrama de clases de las etapas (*Stage*)**

Los elementos de conmutación se encuentran modelados balo la interfaz *Element*. Estos interconexionan N con M objetos de tipo *Connector*. Cuando llega la señal de sincronización la implementación del objeto busca los paquetes en sus N conectores de entrada, llevando a cabo la función para la que fue programado y escribiendo en sus M salidas. Además cada una de las implementaciones de *Element* puede calcular sus propias estadísticas parciales, que serán posteriormente recolectadas por el sistema global. De nuevo existe la posibilidad de que futuras implementaciones puedan ser añadidas al sistema usando la interfaz *Element*.

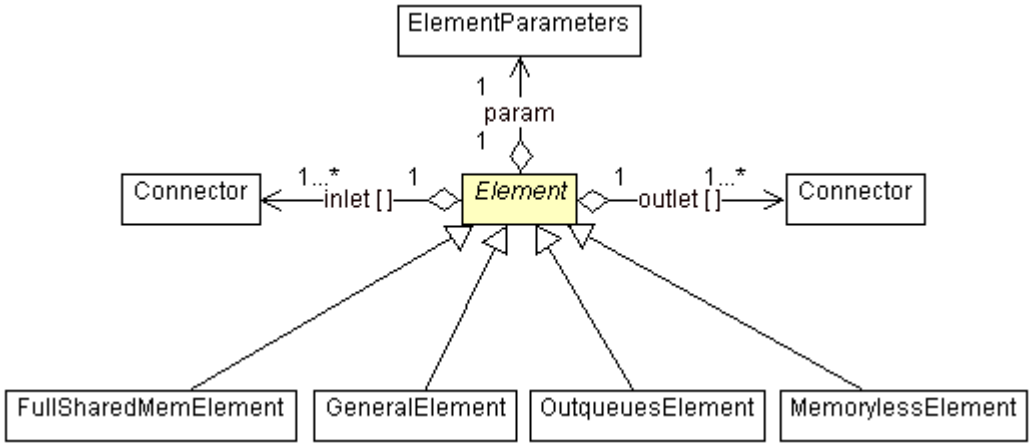


**Figura 3.13 Diagrama de clases de los elementos de conmutación (*Element*)**

En el diagrama anterior pueden apreciarse las diferentes tipos de elementos de conmutación que hay implementados actualmente:

- *GeneralElement:* Modela un elemento de conmutación general, es decir con acceso a la memoria compartida y con una cierta cantidad de memoria propia segregada en colas de salida. Calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida
- *FullSharedMemElement:* Modela un elemento de conmutación con acceso exclusivo a la memoria compartida (no posee memoria propia) para mantener las colas de salida. También calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida.
- *MemorylessElement:* Modela un elemento de conmutación sin memoria. Por lo tanto no tiene mucho sentido hablar de colas de salida o retardo estimado en cola.
- *OutqueuesElement:* Modela un elemento de conmutación con una cierta cantidad de memoria propia segregada en colas de salida. También calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida.

Por último analizaremos el modelado de los conectores encargados de formar la red de interconexión. Dichos conectores forman a su vez las entradas y salidas de la red multietapa y en un futuro deberán ser capaces de manejar varios paquetes a la vez si se pretende que el simulador incluya *WDM*. Bajo la interfaz *Connector* se agrupan los diferentes tipos de conectores:

- *BufferedConnector:* Objeto que modela un conector con cola de salida (con memoria).
- *UnbufferedConnector:* Objeto que modela un conector sin cola de salida (sin memoria).
- *NetworkOutletConnector:* Objeto que modela un conector a la salida de la red multietapa (a el llegan los paquetes conmutados con éxito).

Tal y como puede verse en el diagrama siguiente:

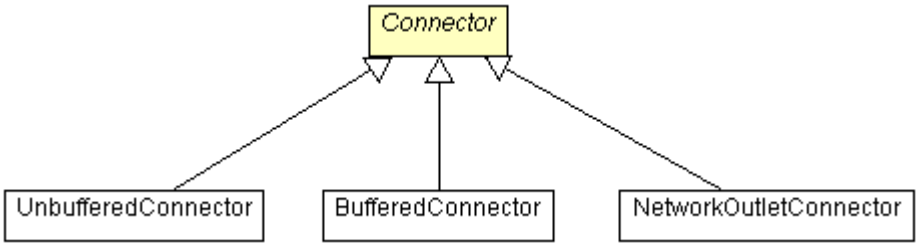


**Figura 3.14 Diagrama de clases de los conectores (*Connector*)**

# Capítulo 4
# Modelado dinámico del simulador

En este apartado trataremos la parte correspondiente al modelado dinámico de la herramienta, que muestra cuales son los procesos a ejecutar, cómo se ejecutan, el orden y la información que fluye entre éstos. Siguiendo la metodología *UML* esta última parte se describiría mediante diagramas de estado, diagramas de actividades y diagramas de interacción. Sin embargo debido a que la naturaleza de un proceso de simulación dirigida por eventos, es de carácter totalmente secuencial, los diagramas de estado no son muy representativos ya que las clases modeladas no poseen una gran cantidad de estados (se usan mas en sistemas concurrentes y/o interactivos). En cambio, si se a optado por usar diagramas de actividades que nos representarán los pasos y tareas que se van ejecutando secuencialmente durante el proceso de simulación y algunos diagramas de secuencia para indicar las clases que intervienen y los datos que se manejan con más detalle en una actividad concreta.

## 4.1 La secuencia global de simulación

Comenzaremos con una primera representación de éste proceso de simulación con un diagrama de actividades global donde se detallan los pasos y tareas a realizar.

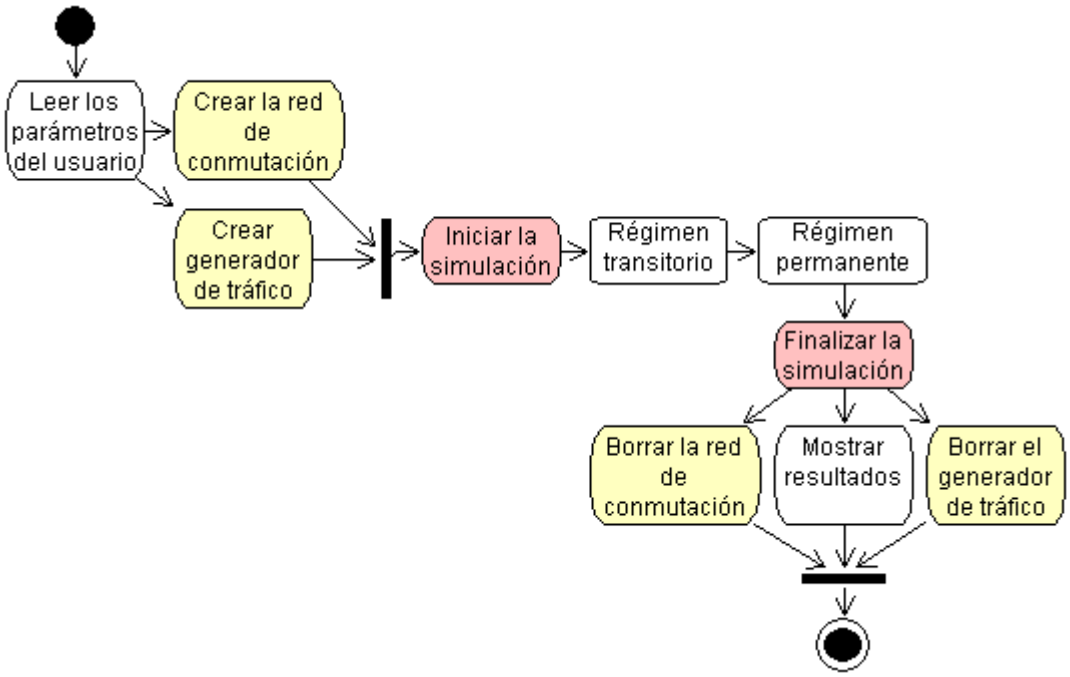


**Figura 4.1 Diagrama de actividades global**

A continuación agruparemos estas actividades en 3 grupos: actividades a realizar previo inicio de la simulación, actividades a realizar durante la simulación y actividades a realizar posteriormente a la misma.

## 4.1.1 Modelado dinámico previo a la simulación

Tras leer los parámetros de simulación proporcionados por el usuario una de las tareas a realizar antes del inicio de la simulación es la creación de la red de conmutación. Representaremos este proceso con un diagrama de secuencia:

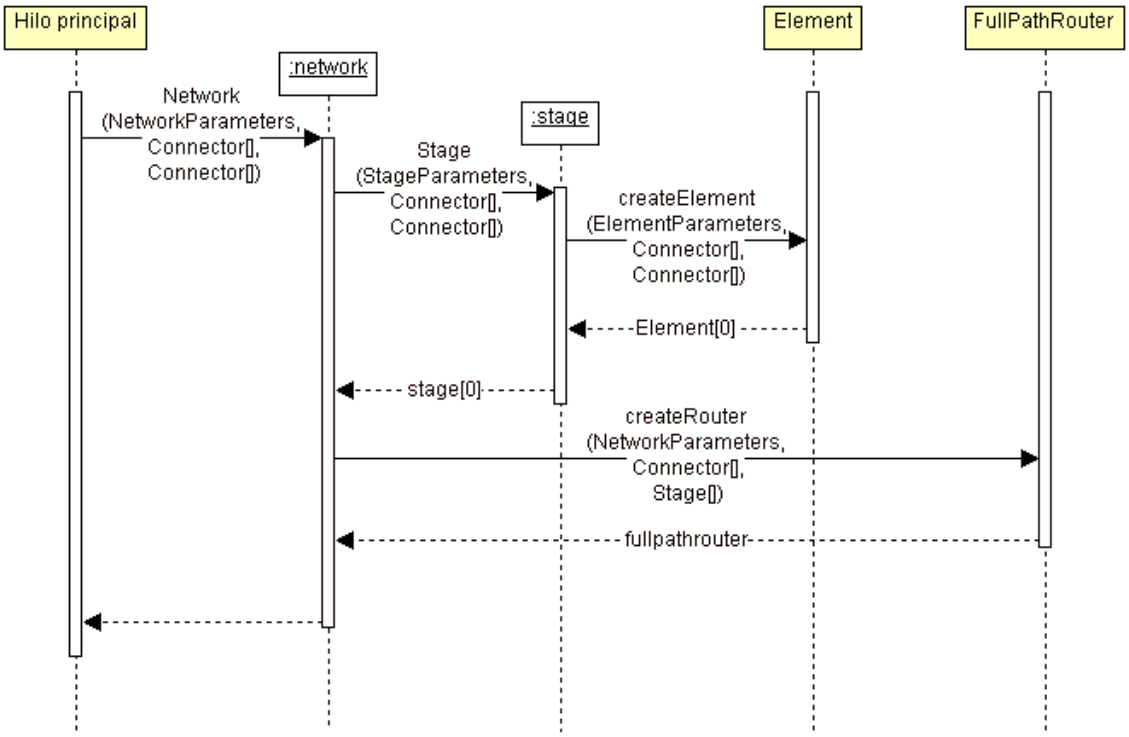


**Figura 4.2 Diagrama de secuencia para crear la red conmutación**

Es importante mencionar que la clase *element* crearía una instancia de una de sus clases hijas (pueden verse en la figura 3.13) que es la que devolvería al objeto :*stage*

De la misma forma la clase *FullPathRouter* crearía un objeto de una de sus clases hijas, por ejemplo una instancia de *FullPathRouter_Random* que es la que devolvería al objeto :*network*

Otra de las tareas a realizar es la creación del generador de tráfico, representaremos este proceso en el siguiente diagrama de secuencia:

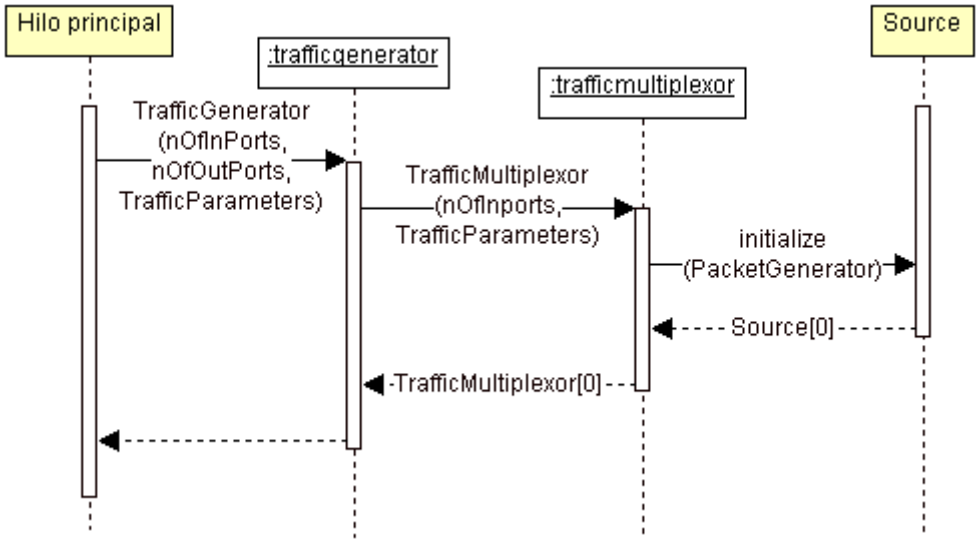


**Figura 4.3 Diagrama de secuencia para crear el generador de tráfico**

En donde la clase *Source* devolvería una instancia de una de sus clases hijas (pueden verse en la figura 3.6).

NOTA: Con el fin de simplificar los diagramas de secuencia de este apartado solo hemos representado la creación del 1º elemento de cada uno de los *arrays*.

## 4.1.2 Modelado dinámico durante la simulación

Una vez la red de conmutación y el generador de paquetes han sido creados, puede empezar la parte central de la simulación, la conmutación propiamente dicha.

Para entender dicho algoritmo diremos que este posee dos estados: el régimen transitorio y el régimen permanente. Cada uno de los cuales posee un número de celdas temporales diferente (parámetro proporcionado por el usuario que sirve como medida de tiempo durante la simulación). En cada una de estas celdas temporales se produce un tic de reloj en el generador de tráfico y en la red de conmutación. A continuación mediante un par de diagramas de secuencia representaremos este proceso:

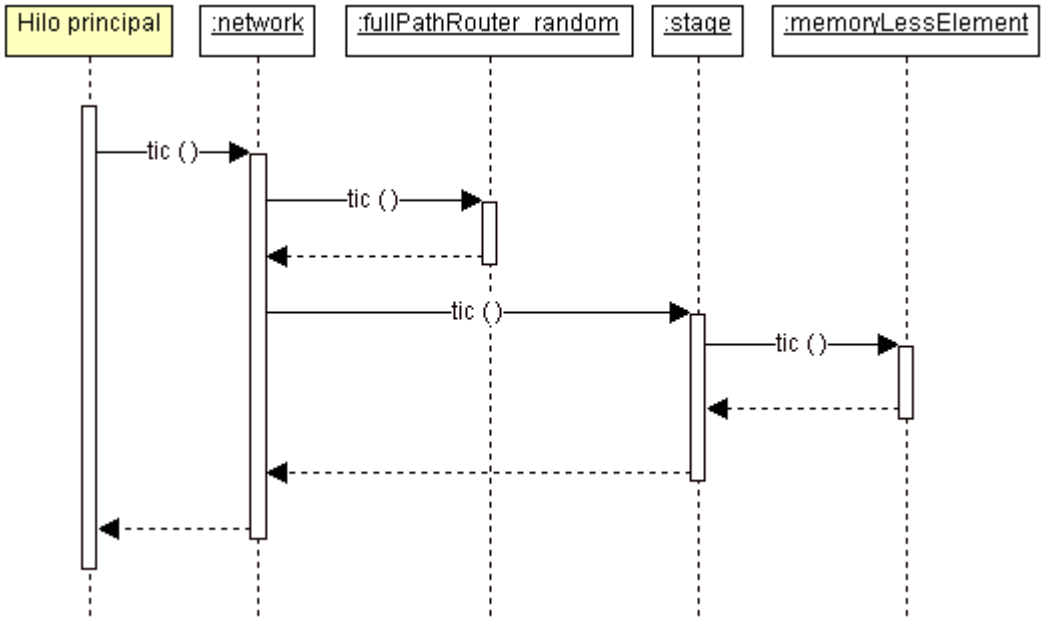


**Figura 4.4 Diagrama de secuencia de un tic en la red de conmutación**

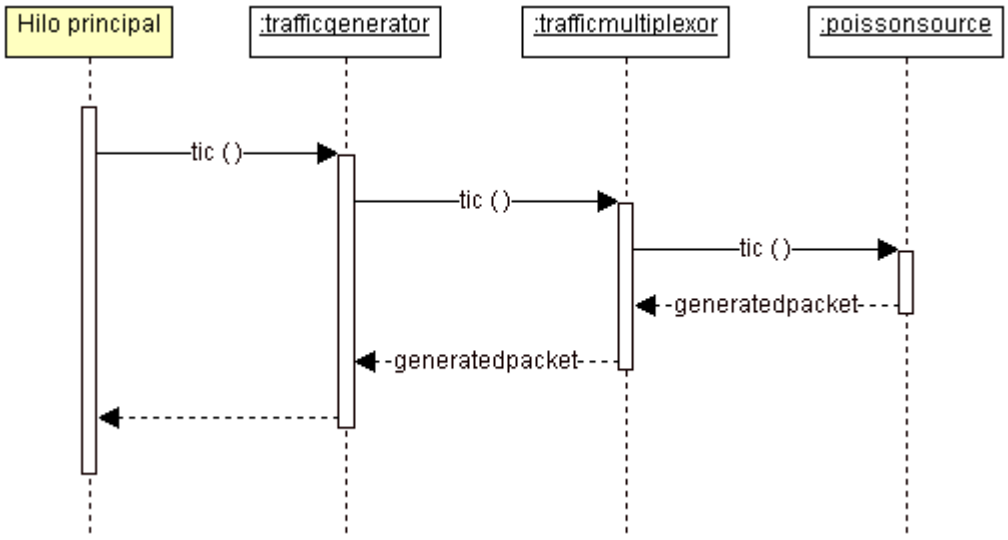


**Figura 4.5 Diagrama de secuencia de un tic en el generador de tráfico**

Para terminar trataremos de modelar lo que sería el ciclo de vida de un paquete que atraviesa el conmutador (describiendo de esta forma el algoritmo de conmutación).

La simulación que hace *PASS* del ciclo de vida del paquete se describe en el siguiente diagrama de actividad de *UML*, dicho diagrama viene determinado por la interacción entre objetos que implementan las interfaces *C++* previamente descritas. El objeto que genera los paquetes define el puerto de entrada y salida. Esta información es usada por el *fullpathrouter* para marcar el camino que seguirá el paquete, el elemento de conmutación que atravesará en cada etapa. Esta información se calcula y se graba en la cabecera del paquete si el *Fullpathrouter* esta programado para ello.

Cada elemento de conmutación (ya sea con *buffer* o sin memoria) debe encargarse de realizar su función, transfiriendo los paquetes de entrada a los puertos de salida (con o sin cola) y siguiendo la estrategia de conmutación para la que fue programado. Si se produce una contención que puede resolverse, o se desborda un *buffer*, los paquetes involucrados se pierden. De forma que el objeto fuente es el único que actualiza las estadísticas globales y destruye los paquetes perdidos (liberándolos de la memoria). En caso de que no se pierda el paquete pasa a la siguiente etapa (dependiendo de la topología de la red). Después de atravesar la última etapa el paquete podemos decir que el paquete se conmutó correctamente. De forma que el paquete es procesado por el objeto fuente o generador que actualiza las estadísticas y al haber abandonado la simulación se libera de la memoria.

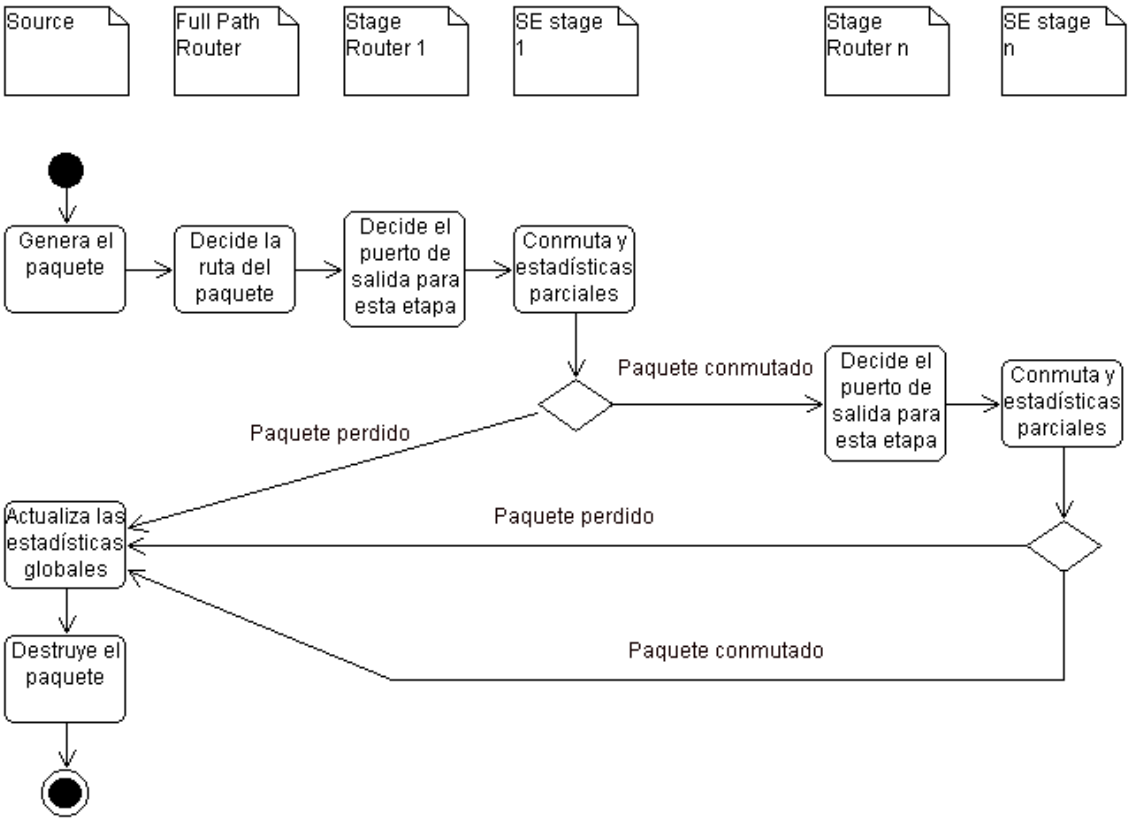


**Figura 4.6 Diagrama de actividad *UML* (ciclo de vida del paquete)**

## 4.1.3 Modelado dinámico posterior a la simulación

Por último después de haber realizado la simulación se borraría la red de conmutación y el generador de tráfico y se presentarían los resultados de la simulación.

# Capítulo 5
# Implementación actual del simulador

## 5.1 Decisiones de implementación

Tras la obtención de un buen modelo que representa el software, siguiendo la metodología *UML* sería el momento de tomar las decisiones de implementación. Al tratarse de un simulador ya desarrollado lo que haremos será explicar las decisiones que se tomaron para su desarrollo. Además siempre que se quisiera añadir determinada funcionalidad al simulador (como la posibilidad de modelar redes *WDM*, multietapa o de determinar la interconexión interna de los elementos) es posible que se presenten determinadas decisiones de cara al desarrollo de dichas ampliaciones para las cuales se tomara como punto de partida el modelado realizado en los capítulos anteriores (metodología *UML*).

Para implementar la herramienta si se hubiera seguido el proceso de modelado *UML* se debería haber realizado un estudio de las diferentes herramientas y lenguajes de programación existentes, y determinar así cual nos ofrecería unos resultados más próximos a los deseados. En este análisis se establecería el uso de un lenguaje de programación orientado a objetos, ya que al haber realizado previamente el modelado orientado a objetos, la representación de ese modelo en uno de estos lenguajes sería directa. Actualmente, los lenguajes de programación orientados a objetos con mayor aceptación son *JAVA* y *C++,* para los cuales existen gran cantidad de herramientas *CAD* para ayuda a la programación de estos modelos. De hecho las figuras mostradas en los capítulos anteriores, han sido obtenidas con las herramientas *UMLStudio* y *RationalRose*. Finalmente la implementación se decantó por el uso del lenguaje *C++,* debido a la necesidad de obtener una herramienta de alta velocidad y fácilmente modificable. Se enumeran los motivos que llevaron a tomar esta decisión:

- *C++* ofrece mayor flexibilidad frente a la robustez y el fuerte tipado de *JAVA*.
- *JAVA* funciona sobre una máquina virtual y esto influye en la velocidad de ejecución del código, pudiéndose considerar *C++* cómo más rápido al usar código nativo de cada máquina sobre la que funciona.
- Es posible combinar código *C++* con código *C* y en la actualidad existen gran cantidad de librerías de utilidades que generan de forma automatizada código *C*, capaz de resolver ciertas tareas extremadamente complejas, cómo son los análisis léxicos, sintácticos, semánticos de ficheros en entrada.
- Una ventaja adicional en el uso de *C++,* es la existencia de una librería estándar denominada *STL* (*Standart Template Library*) [SCHI99] que proporciona tipos de datos usados frecuentemente en todo software como son listas, colas, árboles, vectores, algoritmos, etc. Esta librería se

incluye como parte de todo compilador de C++, siendo así posible la compilación de cualquier programa que las use en diferentes plataformas con soporte C++.

El resultado es una implementación de *PASS* en código *C++* distribuido en 172 ficheros fuente y con un total de 1044 *Kbytes*.

## 5.2 Análisis de la implementación actual

A lo largo de este apartado trataremos de dar una visión global de la implementación actual de *PASS* analizando en detalle las clases presentadas en los apartados anteriores. Debido a la extensión del código hemos optado por diseñar unas fichas donde se analicen las características de dichas clases en profundidad. El formato de las fichas sería el siguiente:

- El nombre de la clase.
- El tipo general de clase (abstracta, derivada, etc).
- Descripción de la misma
- Las variables y objetos pertenecientes a la clase, de los que se muestra:
  - Nombre del objeto o variable.
  - Su modificador de acceso (privado, público, protegido)
  - El tipo de variable (*integer, double, long*) o clase a la que pertenece el objeto
  - Breve descripción de su función
- Métodos principales que posee la clase, de los que se muestra:
  - Nombre del método
  - Su modificador de acceso (privado, público, protegido) o tipo (abstracto, estático)
  - Parámetros que recibe
  - Parámetros que genera
  - Breve descripción de su función

Además se a intentado seguir un orden lógico entre las clases analizadas (tal y como se hizo en los apartados anteriores) agrupándolas según la función que desempeñan.

| **Nombre:** | Simulation |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Objeto encargado de modelar la simulación, es decir, leer los parámetros del usuario, construir la red de conmutación y el generador de tráfico a partir de dichos parámetros, realizar la simulación y por ultimo presentar los resultados. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| parametersFile | *char | Puntero usado para apuntar al fichero de parámetros proporcionado por el usuario |
| resultsFileName | *char | Puntero usado para apuntar al fichero donde imprimiremos los resultados |
| delaysFileName | *char | Puntero usado para apuntar al archivo donde guardar los retardos |
| threeStageNetwork | *Network | Objeto que modela la red de conmutación |
| userDefinedParameters | *UserDefinedParameters | Objeto que modela los parámetros introducidos por el usuario |
| graphicGenerator | *GraphicGenerator | Objeto usado para generar gráficos con los resultados de la simulación (No usado aún) |
| Transactions | long | Numero de celdas totales tratadas en la simulación (régimen transitorio y permanente) |
| discardCell | long | Numero de celdas descartadas en la simulación (régimen transitorio) |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| Simulation | char *fileRoot | | Constructor que inicializa los punteros parametersFile y resultsFileName para apuntar a los ficheros de parámetros y de resultados respectivamente. Además con ellos crea los objetos userDefinedParameters y resultsFile(de tipo ofstream, que usaremos para escribir en el archivo de resultados) |
| ~Simulation | | | Destructor que cierra el fichero de resultados |
| Run | char *directory | void | Crea las variables para controlar la semilla usada para generar los números aleatorios, para mantener los puntos de simulación y para guardar el número de celdas que intentan entrar en el conmutador, además crea el objeto delaysFileName así como el generador de tráfico y la red de conmutación. Luego pone el reloj y el contador de celdas a cero y comienza respectivamente el régimen transitorio y el permanente (cada uno con una duración establecida como parámetro en número de celdas y durante los cuales envía al generador de tráfico y la red de conmutación la señal de reloj en cada celda e incrementa el reloj). Una vez finalizada la simulación borra el generador de tráfico y la red de conmutación y cierra el archivo encargado de guardar los retardos. |
| appendPointResults ToFile | int point | void | Guarda en el fichero de resultados el punto de simulación correspondiente. Así como el estado del la red y del generador de paquetes |

| **Nombre:** | TrafficGenerator |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Módulo principal generador de tráfico que agrupa todos los multiplexores, establece las conexiones necesarias (entradas y salidas) de la red de conmutación. Se encarga de transmitir el tic de reloj a los multiplexores los cuales le devuelven el número de paquetes que cada uno de ellos ha introducido en la red de conmutación en ese tic. Con esta información y mediante las estadísticas que le proporcionan los multiplexores el generador de paquetes calcula diversas estadísticas a nivel global tales como el número total de paquetes introducidos en el último tic, el retardo global de los paquetes o el tráfico (ofrecido, cursado y perdido) total acumulado. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| networkInlets | vector <Connector*> | Entradas de la red multietapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) |
| networkOutlets | vector <Connector*> | Salidas de la red multietapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo NetworkOutletConnector) |
| multiplexors | vector <TrafficMultiplexor*> | Multiplexores de la red multietapa (Vector de punteros a objetos de tipo TrafficMultiplexor) |
| nOfPacketsSentInThisTic | int | Número de paquetes enviados en este tic |
| contAux | int | Contador auxiliar |
| pAux | *SwitchedPacket | Puntero auxiliar para apuntar al paquete conmutado |
| simulationPacketGenerator | *SimulationPacketGenerator | Puntero para apuntar al generador de paquetes |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| TrafficGenerator | int nOfInPorts,<br>int nOfOutPorts,<br>TrafficParameters *trafficParameters | | Constructor que crea las entradas, las salidas y los multiplexores de la red multietapa. Así como el generador de paquetes y además inicializa una fuente (source) asignándole el generador de paquetes. |
| ~TrafficGenerator | | | Destructor que borra las entradas, las salidas, los multiplexores y el generador de paquetes de la red multietapa. |
| tic | | int | Envía el tic a cada uno de los multiplexores, cada uno de los cuales puede generar un paquete (SwitchedPacket) en cuyo caso se introduciría en una de las entradas. Por ultimo devuelve el número de paquetes enviados en este tic. |
| getInlets | | *vector <Connector*> | Devuelve las entradas de la red multietapa |
| getOutlets | | *vector <Connector*> | Devuelve las salidas de la red multietapa |
| getMultiplexers AcumulatePacket Counter | | *PacketCounter | Acumula en un contador todos los contadores de paquetes parciales de cada multiplexor y devuelve el contador acumulado. |
| getMultiplexers WeigthedDelay | | double | Devuelve el retardo global de los paquetes (teniendo en cuenta todos los multiplexores) |

| **Nombre:** | TrafficMultiplexor |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Módulo multiplexor de tráfico que agrupa varias fuentes generadoras de tráfico. Se encarga de transmitirles el tic de reloj a dichas fuentes y de introducir los paquetes que ellas generan en la red de conmutación. Para ello posee una cola con los paquetes a introducir (sacando un paquete de la cola en cada tic). Lleva un contador con el número de paquetes enviados en este tic (uno o ninguno) y además calcula diversas estadísticas con la información que le proporcionas todas sus fuentes tales como el retardo de los paquetes en el multiplexor o el tráfico (ofrecido, cursado y perdido) acumulado del multiplexor. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| inPort | int | Puerto de entrada |
| sources | vector <Source*> | Fuentes generadoras del tráfico (vector de punteros a objetos de tipo fuente) |
| queue | deque <GeneratedPacket*> | Cola con los paquetes a enviar, sacando uno por tic (la cola es un vector de punteros a objetos de tipo paquete) |
| pAux | *GeneratedPacket | Puntero auxiliar para manejar paquetes |
| contAux | int | Contador auxiliar |
| firstSource | int | Marca la posición de la primera fuente del vector de fuentes (cambia en cada tic) |
| nOfPacketsSentInThisTic | int | Contador con los paquetes enviados en este tic (uno o ninguno) |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| TrafficMultiplexor | int inPort , TrafficParameters *trafficParameters | | Constructor que inicializa la variable firstSource, y crea las fuentes (una BackgroundSource y tantas ConnectionSource como conexiones indiquen los parámetros invocando el método createConnection) |
| ~TrafficMultiplexor | | | Destructor que borra las fuentes y la cola de los paquetes |
| tic | | *GeneratedPacket | Envía un tic a cada una de las fuentes asociadas de forma secuencial. Además la primera fuente invocada (firstSource) cambia en cada tic de reloj. Los paquetes creados (si hay alguno) son encolados. Entonces si la cola tiene algún paquete para transmitir de este o de anteriores tics, el primer paquete de la cola se devuelve, para ser introducido en la red de conmutación. |
| show | | void | No implementado aún |
| getSourcesWeightedDelay | | double | Devuelve el retardo global de los paquetes (teniendo en cuenta todos los sources de este multiplexor) |
| getAccumlatedPacketCounter | | *PacketCounter | Acumula en un contador todos los contadores de paquetes parciales de cada source y devuelve el contador acumulado. |

| **Nombre:** | | Source | | |
|---|---|---|---|---|
| **Tipo:** | | Clase abstracta (Interfaz) | | |
| **Descripción:** | | Interfaz de las diversas fuentes que se encargan de la generación de los paquetes según el comportamiento que tienen definido y del cálculo de varias estadísticas. Dichas fuentes usan el objeto *packetGenerator* que proporciona esta interfaz a la hora de crear un paquete. | | |
| **Variables Protegidas:** | | **Tipo/Clase:** | | **Descripción:** |
| *packetGenerator | | PacketGenerator | | Puntero al objeto generador de paquetes (verdaderamente a su interfaz) |
| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
| ~source | virtual | | | Invoca el destructor de la clase hija apropiada |
| tic | virtual | | *GeneratedPacket | Invoca el método tic de la clase hija apropiada |
| passedPacket | virtual | *GeneratedPacket | void | Invoca el método passedPacket de la clase hija apropiada |
| lostPacket | virtual | *GeneratedPacket | void | Invoca el método lostPacket de la clase hija apropiada |
| getPacketCounter | virtual | | *PacketCounter | Invoca el método getPacketCounter de la clase hija apropiada |
| getDelay | virtual | | *AverageEstimation | Invoca el método getDelay de la clase hija apropiada |
| initialize | static | *PacketGenerator | void | Inicializa la fuente asignándole el generador de paquetes |

| **Nombre:** | BackgroundSource |
|---|---|
| **Tipo:** | Clase derivada de Source |
| **Descripción:** | Fuente que simula un aumento del tráfico con diversos puertos de origen y diferentes puertos de salida como destino. Además realiza el cálculo de diversas estadísticas (probabilidad de pérdida de paquete, tráfico y estadísticas de retardo en media) |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| inPort | int | Puerto de entrada |
| bgParam | BackgroundParameters | Parámetros de la fuente |
| *genDecissor | GenDecissor | Puntero al objeto que decide si se genera un paquete (verdaderamente a su interfaz) |
| *outportSelector | OutportSelector | Puntero al objeto que selecciona el puerto de salida (verdaderamente a su interfaz) |
| packetCounter | PacketCounter | Objeto encargado de contar los paquetes (los que entran, los que salen y los que se pierden) y medir el tiempo desde el inicio para estadísticas |
| delay | AverageEstimation | Objeto encargado de calcular el retardo en media |
| contAux | int | Contador auxiliar |
| outPortAux | int | Puerto de salida auxiliar |
| itBgOrderAux | map <int, int>::iterator | Mapa auxiliar para llevar el orden de los paquetes |
| itSeqCheckAux | map <int, DelayVariationWriter*> ::iterator | Mapa auxiliar para llevar la secuencia de los paquetes |
| bgPacketOrder | map <int, int> | Mapa del orden de los paquetes |
| sequenceChecker | map <int, DelayVariationWriter*> | Mapa de la secuencia de los paquetes |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| show | | void | No implementado aún |
| BackgroundSource | int source, Background Parameters param | | Constructor que crea los objetos GenDecissor y el OutportSelector |
| ~BackgroundSource | | | Destructor que borra los objetos GenDecissor y el OutportSelector |
| passedPacket | *GeneratedPacket | void | Invoca al método passedPacket de packetcounter, añade un paquete conmutado a la secuencia y guarda el retardo |
| lostPacket | *GeneratedPacket | void | Invoca al método lostPacket de packetCounter y añade un paquete perdido a la secuencia |
| tic | | *GeneratedPacket | Pregunta a genDecissor si debe generar un paquete y en caso afirmativo le pregunta a outportSelector el puerto de salida y lo genera |
| getPacketCounter | | *PacketCounter | Devuelve el objeto packetCounter |
| getDelay | | *AverageEstimation | Devuelve el objeto delay |

| **Nombre:** | ConnectionSource |
|---|---|
| **Tipo:** | Clase derivada de Source (Interfaz) |
| **Descripción:** | Interfaz de las diversas fuentes que tienen de característica común que simulan un tráfico con un puerto de origen, longitud de onda y puerto de destino fijado, (estableciendo una conexión virtual entre origen y destino). |

| **Variables Protegidas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| inPort | int | Puerto de entrada |
| cnxIndex | int | Identificador de la conexión |
| Param | ConnectionParameters | Parámetros de la fuente |
| cellCounter | int | Contador de celdas |

| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|---|
| ConnectionSource | | int cnxIndex, ConnectionParameters param | | Constructor que inicializa el contador de celdas |
| ~ConnectionSource | virtual | | | Invoca el destructor de la clase hija apropiada |
| show | | | void | No implementado aún |
| createConnection | static | int cnxIndex, ConnectionParameters param | *ConnectionSource | Devuelve el tipo de fuente apropiado (PoissonSource, TOnOffBernouilliSource, TOnOffCBRSource) según el tipo de conexión establecida en param |
| getClosEstad | virtual | | *CellCounter | Invoca al método getClosEstad de la clase hija apropiada (No implementado aún) |
| tic | virtual | | *GeneratedPacket | Invoca el método tic de la clase hija apropiada |
| passedPacket | virtual | *GeneratedPacket | void | Invoca el método passedPacket de la clase hija apropiada |
| lostPacket | virtual | *GeneratedPacket | void | Invoca el método lostPacket de la clase hija apropiada |

| **Nombre:** | TOnOffCBRSource |
|---|---|
| **Tipo:** | Clase derivada de ConnectionSource |
| **Descripción:** | Fuente que simula un tráfico con un puerto de origen, longitud de onda y puerto de destino fijado (estableciendo una conexión virtual entre origen y destino). El tráfico se genera a ráfagas, de forma que la fuente puede atravesar dos estados (*on* y *off*) cada uno de los cuales dura un número de celdas determinado por una distribución geométrica cuya media es un parámetro proporcionado por el usuario. El espacio entre celdas es constante y también es un parámetro proporcionado por el usuario. Además realiza el cálculo de diversas estadísticas (probabilidad de pérdida de paquete, tráfico y estadísticas de retardo en media). |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| nextCell | *GeneratedPacket | Puntero que se usa para devolver el paquete generado (si procede) o NULL en caso contrario |
| currentSendingBurst | *Burst | Puntero a la ráfaga que se está enviando actualmente |
| cellsUntilStateChange | int | Celdas restantes para cambiar de estado |
| weAreInTonState | bool | Estado actual (encendida o apagada) |
| cell_closEstad | CellCounter | Objeto contador de celdas (no usado todavía) |
| packetDisorderMeter | PacketDisorderMeter | Objeto que mantiene el orden de los paquetes |
| burstDisorderMeter | BurstDisorderMeter | Objeto que mantiene el orden de las ráfagas |
| packetCounter | PacketCounter | Objeto encargado de contar los paquetes (los que entran, los que salen y los que se pierden) y medir el tiempo desde el inicio para estadísticas |
| delay | AverageEstimation | Objeto encargado de calcular el retardo en media |
| cellDisorderMeter | PacketDisorderMeter | Objeto que mantiene el orden de las celdas |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| TOnOffCBR Source | int cnxIndex, Connection Parameters param | | Constructor que inicializa los punteros nextCell, currentSendingBurst y las variables cellsUntilState Change,weAreInTonState |
| ~TOnOffCBR Source | | | Destructor que borra el objeto currentSending Burst |
| getClosEstad | | *CellCounter | Devuelve el objeto cell_closEstad (no implementado todavía) |
| tic | | *Generated Packet | Disminuye el numero de celdas restantes para cambiar de estado (puede ocasionar un cambio del estado). Además si la fuente se encuentra en estado encendida genera un paquete e invoca el método enteredPacket de packetcounter. |
| passedPacket | *Generated Packet | void | Invoca al método passedPacket de packetcounter, guarda el retardo y guarda la posición del paquete conmutado y de la ráfaga. |
| lostPacket | *Generated Packet | void | Invoca al método lostPacket de packetcounter y guarda la posición del paquete perdido y la ráfaga. |
| getPacketCounter | | *PacketCounter | Devuelve el objeto packetCounter |
| getDelay | | *Average Estimation | Devuelve el objeto delay |

| **Nombre:** | TOnOffBernouilliSource |
|---|---|
| **Tipo:** | Clase derivada de ConnectionSource |
| **Descripción:** | Fuente que simula un tráfico con un puerto de origen, longitud de onda y puerto de destino fijado (estableciendo una conexión virtual entre origen y destino). El tráfico se genera a ráfagas, de forma que la fuente puede atravesar dos estados (encendida y apagada) cada uno de los cuales dura un número de celdas determinado por una distribución geométrica cuya media es un parámetro proporcionado por el usuario. El espacio entre celdas es aleatorio (sigue una distribución uniforme). Además realiza el cálculo de diversas estadísticas (probabilidad de pérdida de paquete, tráfico y estadísticas de retardo en media). |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| nextCell | *GeneratedPacket | Puntero que se usa para devolver el paquete generado (si procede) o NULL en caso contrario |
| currentSendingBurst | *Burst | Puntero a la ráfaga que se está enviando actualmente |
| cellsUntilStateChange | int | Celdas restantes para cambiar de estado |
| weAreInTonState | bool | Estado actual (encendida o apagada) |
| cell_closEstad | CellCounter | Objeto contador de celdas (no usado todavía) |
| packetDisorderMeter | PacketDisorderMeter | Objeto que mantiene el orden de los paquetes |
| burstDisorderMeter | BurstDisorderMeter | Objeto que mantiene el orden de las ráfagas |
| packetCounter | PacketCounter | Objeto encargado de contar los paquetes (los que entran, los que salen y los que se pierden) y medir el tiempo desde el inicio para estadísticas |
| delay | AverageEstimation | Objeto encargado de calcular el retardo en media |
| cellDisorderMeter | PacketDisorderMeter | Objeto que mantiene el orden de las celdas (no usado todavía) |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| TOnOffBernouilli Source | int cnxIndex, Connection Parameters param | | Constructor que inicializa los punteros nextCell, currentSendingBurst y las variables cellsUntilStateChange,weAreInTonState |
| ~TOnOffBernouilli Source | | | Destructor que borra el objeto currentSendingBurst |
| getClosEstad | | *CellCounter | Devuelve el objeto cell_closEstad (no implementado todavía) |
| tic | | *GeneratedPacket | Disminuye el numero de celdas restantes para cambiar de estado (puede ocasionar un cambio del estado). Además si la fuente se encuentra en estado encendida genera un paquete e invoca el método enteredPacket de packetcounter. |
| passedPacket | *GeneratedPacket | void | Invoca el método passedPacket de packetcounter, guarda el retardo,la posición del paquete conmutado y la ráfaga. |
| lostPacket | *GeneratedPacket | void | Invoca al método lostPacket de packetcounter y guarda la posición del paquete perdido y de la ráfaga. |
| getPacketCounter | | *PacketCounter | Devuelve el objeto packetCounter |
| getDelay | | *AverageEstimation | Devuelve el objeto delay |

| **Nombre:** | PoissonSource |
|---|---|
| **Tipo:** | Clase derivada de ConnectionSource |
| **Descripción:** | Fuente que simula un tráfico con un puerto de origen, longitud de onda y puerto de destino fijado (estableciendo una conexión virtual entre origen y destino). El tráfico se genera de forma aleatoria siguiendo una distribución uniforme (cuyos parámetros son proporcionados por el usuario). Además realiza el cálculo de diversas estadísticas (probabilidad de pérdida de paquete, tráfico y estadísticas de retardo en media). |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| averageIntercellTime | double | Tiempo entre celdas en media |
| packetDisorderMeter | PacketDisorderMeter | Objeto que mantiene el orden de los paquetes |
| packetCounter | PacketCounter | Objeto encargado de contar los paquetes (los que entran, los que salen y los que se pierden) y medir el tiempo desde el inicio para estadísticas |
| delay | AverageEstimation | Objeto encargado de calcular el retardo en media |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| PoissonSource | int cnxIndex, ConnectionParameters param | | Constructor |
| ~PoissonSource | | | Destructor |
| tic | | *GeneratedPacket | De forma aleatoria genera un paquete e invoca el método enteredPacket de packetcounter |
| passedPacket | *GeneratedPacket | void | Invoca al método passedPacket de packetcounter, y guarda el retardo y la posición del paquete conmutado. |
| lostPacket | *GeneratedPacket | void | Invoca al método lostPacket de packetcounter y guarda la posición del paquete perdido. |
| getPacketCounter | | *PacketCounter | Devuelve el objeto packetCounter |
| getDelay | | *AverageEstimation | Devuelve el objeto delay |

| **Nombre:** | PacketGenerator | | | |
|---|---|---|---|---|
| **Tipo:** | Clase abstracta (Interfaz) | | | |
| **Descripción:** | Interfaz del objeto encargado de generar los paquetes. | | | |
| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
| PacketGenerator | | | | Constructor |
| ~PacketGenerator | virtual | | | Invoca al destructor de la clase hija apropiada |
| newPacket | virtual | int inPort,<br>int outPort,<br>ConnectionSource *source,<br>Burst *belongingBurst,<br>int cnxOrder | *Generated Packet | Invoca al método newPacket apropiado (según los parámetros recibidos) de la clase hija apropiada |
| newPacket | virtual | int inPort,<br>int outPort,<br>ConnectionSource*sourceConnection,<br>int cnxOrder | *Generated Packet | Invoca al método newPacket apropiado (según los parámetros recibidos) de la clase hija apropiada |
| newPacket | virtual | int inPort,<br>int outPort,<br>BackgroundSource *sourceBackground, int bgOrder | *Generated Packet | Invoca al método newPacket apropiado (según los parámetros recibidos) de la clase hija apropiada |

| **Nombre:** | SimulationPacketGenerator | | |
|---|---|---|---|
| **Tipo:** | Clase derivada de PacketGenerator | | |
| **Descripción:** | Objeto encargado de generar los paquetes. Posee diferentes métodos para generar los paquetes según el tipo de fuente que lo invoque (dependiendo de los parámetros). | | |
| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
| Simulation PacketGenerator | | | Constructor |
| ~Simulation PacketGenerator | | | Destructor |
| newPacket | int inPort,<br>int outPort,<br>ConnectionSource *source,<br>Burst *belongingBurst,<br>int cnxOrder | *Generated Packet | Genera un paquete para un tipo de fuente ConnectionSource con tráfico a ráfagas (TOnOffCBRSource, TOnOffBernouilliSource) |
| newPacket | int inPort,<br>int outPort,<br>ConnectionSource*sourceConnection,<br>int cnxOrder | *Generated Packet | Genera un paquete para un tipo de fuente ConnectionSource con tráfico aleatorio (PoissonSource) |
| newPacket | int inPort,<br>int outPort,<br>BackgroundSource *sourceBackground,<br>int bgOrder | *Generated Packet | Genera un paquete para un tipo de fuente BackgroundSource |

| **Nombre:** | GeneratedPacket |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Objeto que modela el paquete generado. Posee gran cantidad de variables para registrar el paso del paquete por la red multietapa (puerto de entrada, puerto de salida, momento de entrada, momento de salida, momento de entrada en determinada etapa) también para registrar la ráfaga a la que pertenece (si aplicable), la fuente que lo creó o el orden de secuencia dentro de la conexión (aplicable para objetos tipo *ConnectionSource*). Posee diferentes constructores según el tipo de fuente que lo genere (dependiendo de los parámetros que le proporcione el generador de paquetes). |

| **Variables Públicas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| inPort | int | Puerto de entrada de la red multietapa |
| outPort | int | Puerto de salida de la red multietapa |
| genTime | int | Momento en que el paquete entró en la red multietapa |
| outTime | int | Momento en que el paquete salió de la red multietapa (si no se perdió) |
| timeEnteredSecondStage | int | Momento en que el paquete entró en la segunda etapa de la red multietapa |
| timeEnteredThirdStage | int | Momento en que el paquete entró en la tercera etapa de la red multietapa |
| packetOrder | int | Orden de secuencia de este paquete dentro de la conexión |
| belongingBurst | *Burst | Ráfaga a la que pertenece |
| source | *Source | Objeto fuente que creo este paquete |
| enteredTimeInThisQueue | long | Variable que toma en cada etapa el valor del tiempo en el que entró en esa etapa |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| GeneratedPacket | int inPort,<br>int outPort,<br>ConnectionSource *source,<br>Burst *belongingBurst,<br>int cnxOrder | | Constructor usado por fuentes del tipo ConnectionSource con tráfico a ráfagas (TOnOffCBRSource, TOnOffBernouilliSource) |
| GeneratedPacket | int inPort,<br>int outPort,<br>ConnectionSource *sourceConnection,<br>int cnxOrder | | Constructor usado por fuentes del tipo ConnectionSource con tráfico aleatorio (PoissonSource) |
| GeneratedPacket | int inPort,<br>int outPort,<br>BackgroundSource *sourceBackground,<br>int bgOrder | | Constructor usado por fuentes del tipo BackgroundSource |
| passed | | void | Invoca al método passedPacket de source con el mismo paquete como argumento |
| lost | | void | Invoca al método lostPacket de source con el mismo paquete como argumento |

| **Nombre:** | SwitchedPacket |
|---|---|
| **Tipo:** | Clase derivada de GeneratedPacket |
| **Descripción:** | Objeto que modela el paquete conmutado. Mantiene todas las características de *GeneratedPacket* (hereda de él) y añade un array de enteros donde almacena el puerto de salida que atravesó en cada etapa de la red de conmutación. |

| **Variables Públicas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| outPortOfThisStageSE | int | Variable que toma en cada etapa el valor del puerto de salida para el elemento de esa etapa |
| outPorts | int [ ] | Cada uno de los valores del puerto de salida, en el elemento de conmutación que le toque en cada etapa |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| SwitchedPacket | int inPort,<br>int outPort,<br>ConnectionSource *source,<br>Burst *belongingBurst,<br>int cnxOrder | | Constructor usado por fuentes del tipo ConnectionSource con tráfico a ráfagas (TOnOffCBRSource, TOnOffBernouilliSource), llama al constructor de GeneratedPacket |
| SwitchedPacket | int inPort,<br>int outPort,<br>ConnectionSource *sourceConnection,<br>int cnxOrder | | Constructor usado por fuentes del tipo ConnectionSource con tráfico aleatorio (PoissonSource), llama al constructor de GeneratedPacket |
| SwitchedPacket | int inPort,<br>int outPort,<br>BackgroundSource *sourceBackground,<br>int bgOrder | | Constructor usado por fuentes del tipo BackgroundSource, llama al constructor de GeneratedPacket |

| **Nombre:** | Network |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Objeto que modela la red de conmutación. Agrupa todas las etapas (*Stages*) y el objeto encargado del encaminamiento (*FullPathRouter*). La acción de conmutar es gobernada por este objeto, que invoca el método de sincronización apropiado (tic o señal de reloj) en los objetos de la red. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| inlet | vector <Connector*> | Entradas de la red multietapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) |
| outlet | vector <Connector*> | Salidas de la red multietapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo NetworkOutletConnector) |
| stage | vector <Stage*> | Etapas de la red multietapa (Vector de punteros a objetos de tipo Stage) |
| fullPathRouter | *FullPathRouter | Puntero al objeto que toma las decisiones de encaminamiento de manera global (verdaderamente a su interfaz) |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| Network | NetworkParameters param, vector <Connector*> *inlets, vector <Connector *> *outlets | | Constructor que crea las etapas y el fullPathRouter (actualmente solo se permite el tipo de interconexión fullconnect entre etapas) |
| ~Network | | | Destructor que borra las etapas y el fullPathRouter |
| getStages | vector <Stage*> *placeToSaveStages | void | Devuelve un vector con punteros a todas las etapas |
| tic | | void | Comienzo de un nuevo ciclo de conmutación: se envía un tic al fullPathRouter que establece la ruta para los paquetes provenientes de los TrafficMultiplexors. Y luego se envía un tic a cada una de las etapas. |
| showClos | | void | Invoca el método show del fullPathRouter y de cada una de las etapas. (Para debug). |

| **Nombre:** | FullPathRouter |
|---|---|
| **Tipo:** | Clase abstracta (Interfaz) |
| **Descripción:** | Interfaz de los objetos (actualmente solo *FullPathRouter_Random*) encargados de tomar las decisiones de encaminamiento de forma global. Los paquetes son tratados de forma independiente, y bajo esta interfaz se pueden implementar tanto estrategias de enrutado estáticas como dinámicas. Potencialmente cualquier tipo de conexión esta permitida, aunque la implementación actual de PASS esta limitada a la interconexión total entre etapas, donde el numero de etapas, elementos por etapa y longitudes de onda son parámetros configurables. |

| **Variables Protegidas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| param | NetworkParameters | Parámetros de la red |
| inlets | vector <Connector*> | Entradas de la red multietapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) |
| stages | vector <Stage*> | Etapas de la red multietapa (Vector de punteros a objetos de tipo Stage) |

| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|---|
| FullPathRouter | | NetworkParameters param, vector <Connector*> *inlets, vector <Stage*> *stages | | Constructor |
| ~FullPathRouter | virtual | | | Invoca al destructor de la clase hija apropiada |
| tic | virtual | | void | Invoca al método tic de la clase hija apropiada |
| show | virtual | | void | Invoca al método show de la clase hija apropiada |
| createRouter | static | NetworkParameters param, vector <Connector*> *inlets, vector <Stage *> *stages | *FullPathRouter | Crea y devuelve una instancia de una clase hija de FullPathRouter |

| **Nombre:** | FullPathRouter_Random |
|---|---|
| **Tipo:** | Clase derivada de FullPathRouter |
| **Descripción:** | Objeto que genera la ruta que seguirá el paquete a través de las diversas etapas del conmutador de forma aleatoria y la almacena en la cabecera del paquete. Se encarga de tomar las decisiones de encaminamiento de manera global. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| nOfOutPortsOfElements | *int | Puntero a un array de enteros con el número de puertos de salida de los elementos de cada una de las etapas. |
| contAux1 | int | Contador auxiliar |
| contAux2 | int | Contador auxiliar |
| pAux | *SwitchedPacket | Puntero auxiliar para manejar paquetes |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| FullPathRouter_Random | NetworkParameters param, vector <Connector*> *inlets, vector <Stage*> *stages | | Constructor que guarda el número de puertos de salida de los elementos de cada una de las etapas en el array nOfOutPortsOfElement |
| ~FullPathRouter_Random | | | Destructor que borra el array nOfOutPortsOfElements |
| tic | | void | Genera la ruta que seguirá cada uno de los paquetes que entre en la red de conmutación en este tic de forma aleatoria. |
| show | | void | Muestra el estado del FullPathRouter En concreto devuelve por la salida estándar una lista de las entradas y de los respectivos paquetes que pretenden entrar por ellas, así como el número de puertos de salida de los elementos de cada una de las etapas. |

| **Nombre:** | Stage |
|---|---|
| **Tipo:** | Clase |
| **Descripción:** | Objeto que modela una etapa de la red de conmutación, agrupando todos los elementos de conmutación pertenecientes a dicha etapa. Cuando recibe la señal de sincronización marca el puerto de salida en los paquetes para la siguiente etapa y transmite la señal de reloj a sus elementos de conmutación. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| param | StageParameters | Parámetros de la etapa |
| inlet | vector <Connector*> | Entradas de la etapa (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) |
| element | vector <Element*> | Elementos de conmutación de esta etapa (Vector de punteros a objetos de tipo Element) |
| contAux | int | Contador auxiliar |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| Stage | StageParameters param, vector <Connector*>*outlets | | Constructor que crea los elementos pertenecientes a la etapa así como los conectores de entrada de la etapa. Los elementos se conectan siguiendo una topología fullconnect (la única permitida actualmente) |
| Stage | StageParameters param, vector <Connector*>*inlets, vector <Connector*>*outlets | | Constructor que crea los elementos de la etapa. Los elementos se conectan siguiendo una topología fullconnect (la única permitida actualmente) |
| ~Stage | | | Destructor que borra los conectores de entrada de la etapa (UnbufferedConnectors) así como los elementos de la etapa. |
| tic | | Void | Almacena en el campo outPortOfThisStage del paquete el valor del puerto de salida que tendrá en esta etapa (Los elementos de conmutación solo miran ese campo). También se almacena el tiempo de simulación en los campos timeEnteredSecondStage o timeEnteredThirdStage si nos encontramos en la etapa indicada. Por último transmite el tic a todos sus elementos. |
| show | | Void | Devuelve por la salida estándar los parámetros de la etapa, los parámetros de cada uno de sus elementos (Para debug) |
| getInlet | | vector <Connector*>* | Devuelve los conectores de entrada de esta etapa. (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) |

| **Nombre:** | Element | | | |
|---|---|---|---|---|
| **Tipo:** | Clase abstracta (Interfaz) | | | |
| **Descripción:** | Interfaz bajo la que se agrupan los diferentes elementos de conmutación. Interconexiona N con M objetos de tipo *Connector*. Cuando llega la señal de sincronización la implementación del objeto busca los paquetes en sus N conectores de entrada, llevando a cabo la función para la que fue programado y escribiendo en sus M salidas. Además cada una de las implementaciones de *Element* puede calcular sus propias estadísticas parciales, que serán posteriormente recolectadas por el sistema global. De nuevo existe la posibilidad de que futuras implementaciones puedan ser añadidas al sistema usando la interfaz *Element*. | | | |
| **Variables Protegidas:** | **Tipo/Clase:** | **Descripción:** | | |
| param | ElementParameters | Parámetros del elemento de conmutación | | |
| inlet | vector <Connector*> | Entradas del elemento de conmutación (Vector de punteros a objetos de tipo Connector, que luego se usa en concreto para apuntar a objetos de tipo UnbufferedConnector) | | |
| outlet | vector <Connector*> | Salidas del elemento de conmutación (Vector de punteros a objetos de tipo Connector) | | |
| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
| Element | | ElementParameters param,<br>vector <Connector*>*inlets,<br>vector <Connector*>*outlets | | Constructor que comprueba que el número de puertos de entrada o de salida es un valor válido |
| ~Element | virtual | | | Invoca al destructor de la clase hija apropiada |
| createElement | static | ElementParameters param,<br>vector <Connector*>*inlets,<br>vector <Connector*>*outlets | *Element | Crea y devuelve una instancia de una clase hija de Element, en función de los parámetros que se le pasen en param |
| tic | virtual | | void | Invoca al método tic de la clase hija apropiada |
| freeMemory | virtual | | int | Invoca al método freeMemory de la clase hija apropiada |
| estimatedDelay | virtual | int inPort,<br>int outPort | int | Invoca al método estimatedDelay de la clase hija apropiada |
| show | virtual | int stage,<br>int index | void | Invoca al método show de la clase hija apropiada |

| **Nombre:** | GeneralElement |
|---|---|
| **Tipo:** | Clase derivada de Element |
| **Descripción:** | Objeto que modela un elemento de conmutación general, es decir con acceso a la memoria compartida y con una cierta cantidad de memoria propia segregada en colas de salida. También calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| outQueue | vector <PacketQueue*> | Vector con las colas correspondientes a cada puerto de salida |
| nOfCellsInSegregatedMem | int | Número de celdas en la memoria segregada |
| nOfCellsInSharedMem | int | Número de celdas en la memoria compartida |
| nOfCellsInTotMem | int | Número de celdas en la memoria total |
| firstLookUpPort | int | Puerto de entrada en el que mirar en primer lugar |
| iteration | int | Iterador |
| newPacket | *SwitchedPacket | Puntero para referirse a los nuevos paquetes que pretenden entrar en la correspondiente cola de salida |
| poppedPacket | *SwitchedPacket | Puntero para referirse a los paquetes que salen de la cola de salida correspondiente |
| oPort | int | Variable para referirse a uno de los puertos de salida |
| inPort | int | Variable para referirse a uno de los puertos de entrada |
| counter | int | Contador |
| outportQueueAux | *PacketQueue | Puntero auxiliar para referirse a una de las colas de salida |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| GeneralElement | ElementParameters param, vector <Connector*>*inlets, vector <Connector*>*outlets | | Constructor que inicializa las variables nOfCellsInSegregatedMem, nOfCellsInSharedMem, nOfCellsInTotMem, firstLookUpPort, iteration y el vector outQueue |
| ~GeneralElement | | | Destructor que borra el vector outQueue |
| tic | | void | Examina las entradas en busca de un paquete, si lo encuentra intenta guardarlo en la cola correspondiente a su puerto de salida (si no hay espacio en la memoria segregada ni en la memoria compartida el paquete se pierde). Luego saca por los puertos de salida correspondientes los paquetes que se encuentren en cabeza de cada una de las colas y los libera de la memoria. |
| freeMemory | | int | Devuelve la cantidad de memoria disponible en el elemento. La memoria total (parámetro) - la memoria total usada (nOfCellsInTotMem) |
| estimatedDelay | int inPort, int outPort | int | Devuelve una estimación de lo que un paquete que entrase por inport en ese momento queriendo ir a outport tendría que esperar. (Tamaño de la cola para ese puerto de salida) |
| show | int stage, int index | void | Devuelve por la salida estándar los parámetros del elemento, el estado de la cola e invoca el método show de las entradas (para debug) |

| **Nombre:** | FullSharedMemElement |
|---|---|
| **Tipo:** | Clase derivada de Element |
| **Descripción:** | Objeto que modela un elemento de conmutación con acceso exclusivo a la memoria compartida (no posee memoria propia) para mantener las colas de salida. También calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| outQueue | vector <PacketQueue*> | Vector con las colas correspondientes a cada puerto de salida |
| nOfCellsInSharedMem | int | Número de celdas en la memoria compartida |
| firstLookUpPort | int | Puerto de entrada en el que mirar en primer lugar |
| iteration | int | Iterador |
| newPacket | *SwitchedPacket | Puntero para referirse a los nuevos paquetes que pretenden entrar en la correspondiente cola de salida |
| poppedPacket | *SwitchedPacket | Puntero para referirse a los paquetes que salen de la cola de salida correspondiente |
| oPort | int | Variable para referirse a uno de los puertos de salida |
| inPort | int | Variable para referirse a uno de los puertos de entrada |
| counter | int | Contador |
| outportQueueAux | *PacketQueue | Puntero auxiliar para referirse a una de las colas de salida |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| FullSharedMem Element | ElementParameters param, vector <Connector*>*inlets, vector <Connector*>*outlets | | Constructor que inicializa las variables nOfCellsInSharedMem, firstLookUpPort, iteration y el vector outQueue |
| ~FullSharedMem Element | | | Destructor que borra el vector outQueue |
| tic | | void | Examina las entradas en busca de un paquete, si lo encuentra intenta guardarlo en la cola correspondiente a su puerto de salida (si no hay espacio en la memoria compartida el paquete se pierde). Luego saca por los puertos de salida correspondientes los paquetes que se encuentren en cabeza de cada una de las colas y los libera de la memoria compartida. |
| freeMemory | | int | Devuelve la cantidad de memoria disponible en el elemento. La memoria total (parámetro) - la memoria compartida usada (nOfCellsInSharedMem) |
| estimatedDelay | int inPort, int outPort | int | Devuelve una estimación de lo que un paquete que entrase por inport en ese momento queriendo ir a outport tendría que esperar. (Tamaño de la cola para ese puerto de salida) |
| show | int stage, int index | void | Devuelve por la salida estándar los parámetros del elemento (para debug) |

| **Nombre:** | MemorylessElement | |
|---|---|---|
| **Tipo:** | Clase derivada de Element | |
| **Descripción:** | Objeto que modela un elemento de conmutación sin memoria. Por lo tanto no tiene mucho sentido hablar de colas de salida o retardo estimado en cola. | |
| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
| freeOutport | vector <bool> | Vector que indica si cada uno de los puertos de salida están libres o ocupados |
| packetCounterPerOutPort | vector <PacketCounter> | Vector con un contador de paquetes por puerto de salida para realizar estadísticas (Paquetes perdidos/conmutados, probabilidad de pérdida, etc). |
| firstLookUpPort | int | Puerto de entrada en el que mirar en primer lugar |
| iteration | int | Iterador |
| newPacket | *SwitchedPacket | Puntero para apuntar a los paquetes que intentan atravesar el elemento de conmutación |
| poppedPacket | *SwitchedPacket | Puntero no usado |
| oPort | int | Variable para referirse a uno de los puertos de salida |
| inPort | int | Variable para referirse a uno de los puertos de entrada |
| counter | int | Contador |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| MemorylessElement | ElementParameters param, vector <Connector*>*inlets, vector <Connector*>*outlets | | Constructor que inicializa las variables firstLookUpPort e iteration. |
| ~Memoryless Element | | | Destructor |
| tic | | void | Examina las entradas en busca de un paquete, si lo encuentra intenta sacarlo por su puerto de salida correspondiente (si el puerto de salida esta ocupado el paquete se pierde). |
| freeMemory | | int | Devuelve un cero (no tiene sentido hablar de la memoria libre en un elemento sin memoria) |
| estimatedDelay | int inPort, int outPort | int | Devuelve un cero (no existe retardo al ser un elemento sin memoria, o se conmuta o se pierde, no hay una cola en los puertos de salida) |
| show | int stage, int index | void | Devuelve por la salida estándar los parámetros del elemento (para debug) |

| **Nombre:** | OutqueuesElement |
|---|---|
| **Tipo:** | Clase derivada de Element |
| **Descripción:** | Objeto que modela un elemento de conmutación con una cierta cantidad de memoria propia segregada en colas de salida. También calcula ciertas estimaciones como el retardo medio en cola para cada puerto de salida. |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| outQueue | vector <PacketQueue*> | Vector con las colas correspondientes a cada puerto de salida |
| nOfCellsInSegregatedMem | int | Número de celdas en la memoria segregada |
| firstLookUpPort | int | Puerto de entrada en el que mirar en primer lugar |
| iteration | int | Iterador |
| newPacket | *SwitchedPacket | Puntero para referirse a los nuevos paquetes que pretenden entrar en la correspondiente cola de salida |
| poppedPacket | *SwitchedPacket | Puntero para referirse a los paquetes que salen de la cola de salida correspondiente |
| oPort | int | Variable para referirse a uno de los puertos de salida |
| inPort | int | Variable para referirse a uno de los puertos de entrada |
| counter | int | Contador |
| outportQueueAux | *PacketQueue | Puntero auxiliar para referirse a una de las colas de salida |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| OutqueuesElement | ElementParameters param, vector <Connector*>*inlets, vector <Connector*>*outlets | | Constructor que inicializa las variables nOfCellsInSegregatedMem, firstLookUpPort, iteration y el vector outQueue |
| ~OutqueuesElement | | | Destructor que borra el vector outQueue |
| tic | | void | Primero saca por los puertos de salida correspondientes los paquetes que se encuentren en cabeza de cada una de las colas y los libera de la memoria segregada. Luego examina las entradas en busca de un paquete, si lo encuentra intenta guardarlo en la cola correspondiente a su puerto de salida (si no hay espacio en la memoria segregada el paquete se pierde). |
| freeMemory | | int | Devuelve la cantidad de memoria disponible en el elemento. La memoria total (parámetro) - la memoria segregada usada (nOfCellsInSegregatedMem) |
| estimatedDelay | int inPort, int outPort | int | Devuelve una estimación de lo que un paquete que entrase por inport en ese momento queriendo ir a outport tendría que esperar. (Tamaño de la cola para ese puerto de salida) |
| show | int stage, int index | void | Devuelve por la salida estándar el estado de la cola y de las entradas (para debug) |

| **Nombre:** | Connector | | | |
|---|---|---|---|---|
| **Tipo:** | Clase abstracta (Interfaz) | | | |
| **Descripción:** | Interfaz de los objetos que modelan la red de interconexión (los conectores) y son las entradas y salidas de la red multietapa. | | | |
| **Métodos Públicos:** | **Tipo:** | **Recibe:** | **Genera:** | **Descripción:** |
| writePacket | virtual | SwitchedPacket *c | void | Invoca al método writePacket de su clase hija apropiada |
| readPacket | virtual | | *SwitchedPacket | Invoca al método readPacket de su clase hija apropiada |
| front | virtual | | *SwitchedPacket | Invoca al método front de su clase hija apropiada |
| nOfPacketsStored | virtual | | int | Invoca al método nOfPacketsStored de su clase hija apropiada |
| show | virtual | int index | void | Invoca al método show de su clase hija apropiada |

| **Nombre:** | BufferedConnector |
|---|---|
| **Tipo:** | Clase derivada de Connector |
| **Descripción:** | Objeto que modela un conector con cola de salida (con memoria) |

| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** |
|---|---|---|
| queue | deque <SwitchedPacket *> | Cola con los paquetes almacenados en el conector |
| pAux | *SwitchedPacket | Puntero auxiliar para manejar los paquetes |

| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
|---|---|---|---|
| BufferedConnector | | | Constructor |
| ~BufferedConnector | | | Destructor que borra los elementos de la cola uno a uno |
| writePacket | SwitchedPacket *c | Void | Escribe un paquete al final de la cola |
| readPacket | | *SwitchedPacket | Devuelve el paquete en cabeza de la cola y avanza una posición (para la próxima lectura), si la cola está vacía devuelve NULL |
| front | | *SwitchedPacket | Devuelve el paquete en cabeza de la cola, si la cola está vacía devuelve NULL |
| nOfPacketsStored | | Int | Devuelve el tamaño de la cola |
| show | int index | Void | No implementado aun |

| **Nombre:** | UnbufferedConnector | | |
|---|---|---|---|
| **Tipo:** | Clase derivada de Connector | | |
| **Descripción:** | Objeto que modela un conector sin cola de salida (sin memoria) | | |
| **Variables Privadas:** | **Tipo/Clase:** | **Descripción:** | |
| p | *SwitchedPacket | Puntero para apuntar al paquete almacenado en el conector | |
| aux | *SwitchedPacket | Puntero auxiliar para manejar los paquetes | |
| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
| UnbufferedConnector | | | Constructor que inicializa p a NULL |
| ~UnbufferedConnector | | | Destructor que borra el paquete almacenado en el conector |
| writePacket | SwitchedPacket *c | Void | Almacena el paquete |
| readPacket | | *SwitchedPacket | Devuelve el paquete almacenado y lo borra |
| front | | *SwitchedPacket | Devuelve el paquete almacenado |
| nOfPacketsStored | | int | Devuelve el número de paquetes almacenados (puede ser cero o uno) |
| show | int index | void | No implementado aun |

| **Nombre:** | NetworkOutletConnector | | |
|---|---|---|---|
| **Tipo:** | Clase derivada de Connector | | |
| **Descripción:** | Objeto que modela un conector a la salida de la red multietapa (a el llegan los paquetes conmutados con éxito) | | |
| **Métodos Públicos:** | **Recibe:** | **Genera:** | **Descripción:** |
| NetworkOutletConnector | | | Constructor que inicializa p a NULL |
| ~NetworkOutletConnector | | | Destructor |
| writePacket | SwitchedPacket *c | void | Invoca el método passed del paquete (Paquete conmutado con éxito) |
| readPacket | | *SwitchedPacket | Devuelve NULL |
| front | | *SwitchedPacket | Devuelve NULL |
| nOfPacketsStored | | int | Devuelve cero |
| show | int index | void | No implementado aun |

# Capítulo 6
# Líneas futuras y conclusión

En este capítulo analizaremos las líneas futuras de desarrollo del proyecto y del simulador. Mostraremos además como deberían abordarse las futuras modificaciones del simulador para modelar redes *WDM*, multietapa o añadir nuevas formas de interconexión. Y por último concluiremos explicando lo aprendido a lo largo del proyecto.

## 6.1 Líneas futuras

En un futuro desarrollo del proyecto podrían incluirse objetivos tales como implementar alguna de las modificaciones del simulador que se propondrán a continuación o desarrollar el simulador en java a partir del modelado *UML* realizado en el proyecto.

Con respecto al desarrollo del simulador se buscaría extender su funcionalidad para modelar conmutadores ópticos multietapa *WDM*. Así pues algunas de las mejoras a realizar serían realizar una conversión *WDM*, una conversión multietapa, permitir nuevos tipos de interconexión entre los elementos internos del conmutador o realizar gráficas con los resultados de la simulación. A continuación analizaremos como abordar dichas modificaciones.

### 6.1.1 Conversión WDM

Esta mejora permitiría evaluar conmutadores o *switchs* ópticos de redes *WDM*. Es importante recordar que la interconexión de conmutadores ópticos posee algunas limitaciones físicas, resultado de los efectos combinados del comportamiento no ideal de los componentes ópticos clave. La degradación de la señal debido a la tasa de atenuación, la acumulación del ruido y la sincronización de señales a la salida son los principales factores que imponen dicha limitación. El análisis de estos efectos no sería el objetivo de nuestro simulador, de forma que los interesados en usar la herramienta deberían previamente determinar la viabilidad de la arquitectura hardware que pretendan evaluar.

Las modificaciones a realizar en el módulo generador de tráfico serían:

- Crear una clase *WDMGeneratedPacket* parecida a *GeneratedPacket* con una variable para almacenar la longitud de onda de entrada y de salida en la red multietapa. (Al igual que el puerto de entrada y de salida estas variables serán recibidas como parámetro en el constructor)
- Crear una clase *WDMSwitchedPacket*, con las siguientes modificaciones sobre el *SwitchedPacket* original:
    - Además de una variable para registrar el puerto de salida en la etapa actual "*outportForThisStage*", tendrá que tener otra variable para registrar la longitud de onda de salida en la etapa actual

"*outLambdaForThisStage*" en esta variable escribirán los objetos *Stage* para que los *WDMElement* la lean.

  - De igual forma tendrá que tener un *array* "*outLambdas* [ ]" de tamaño el número de etapas de la red de conmutación, donde *WDMFullPathRouter* almacene la longitud de onda de salida para cada etapa. Tal y como ocurría con el *array* "*outPorts* [ ]".

- Crear una interfaz *WDMPacketGenerator* y su correspondiente implementación (la encargada de crear los *WDMGeneratedPacket*) que reciba también como parámetro en el constructor la longitud de onda de entrada y de salida de la red de conmutación.
- Las fuentes, los multiplexores y la clase *TrafficGenerator* no sufren grandes cambios (sólo tendrán que adaptarse al nombre de las nuevas clases). Esto es debido a que las clases encargadas de generar el tráfico han sido desarrolladas de forma independiente de la implementación de los diferentes elementos de conmutación. De forma que los cambios de los elementos de conmutación no afectan mucho a la generación del tráfico.

Las modificaciones a realizar en la red de conmutación serían:

- Cambiar el interfaz *Connector* a *WDMConnector*, que tendrá al menos como parámetro en el constructor, el número de lambdas. (Cuando el número de lambdas sea uno su comportamiento deberá será igual al del la interfaz *Connector* actual)
- Crear una clase *WDMUnbufferedConnector* similar a *UnbufferedConnector*, pero teniendo en cuenta que debe ser capaz de transmitir todos los *WDMSwitchedPacket* que tenga en sus longitudes de onda (antes solo un *SwitchedPacket*) en un solo tic.
- Crear una clase *WDMNetworkOutletConnector* a imagen de *NetworkOutletConnector*, es decir punto de terminación de la red, pero que sea capaz de recibir varios *SwitchedPacket* en un solo tic (y de invocar el método "*passed*" de cada uno de ellos)
- Cambiar la interfaz *FullPathRouter* a *WDMFullPathRouter* (y su correspondiente implementación) teniendo en cuenta que deberá escribir no solo los puertos de salida que atravesará cada *WDMSwitchedPacket* sino también las longitudes de onda de salida para cada etapa (representando así una conversión de longitud de onda). De forma que toda operación de conmutación combina la conmutación espacial con la conmutación por longitud de onda.
- Crear una interfaz *WDMElement*, con un funcionamiento paralelo al de *Element* así como alguna implementación del mismo, por ejemplo:
  - Crear un *WDMGeneralElement* que herede de *WDMElement* similar a *GeneralElement*, pero teniendo en cuenta que las colas de salida ahora serán no solo por puerto de salida, sino también por longitud de onda de salida. (Además el tipo de paquetes encolados ahora serán del tipo *WDMSwitchedPacket*). Las demás implementaciones serán versiones simplificadas de este elemento general. (Tal y como pasa con la versión actual de *PASS*)

    - Crear un *WDMSpatialSwitch* que herede de *WDMElement*, y con el siguiente funcionamiento:
        - En cada tic de reloj, todos los *WDMSwitchedPackets* de un puerto de entrada, deben ir al mismo puerto de salida (el que diga por ejemplo el de frecuencia1 en "*outPortForThisStage*", ya que todos deben decir lo mismo).
        - Se implementen todos los chequeos adecuados (que todos los paquetes en la misma entrada, quieran ir a la misma salida).
- Crear un *WDMStage* similar a *Stage* que al recibir la señal de reloj no sólo marcara el puerto de salida que tendrán los paquetes en esta etapa, sino también la longitud de onda de salida. (Además de transmitir la señal de reloj a todos sus *WDMElement*)
- La clase *Network* tampoco sufrirá grandes cambios (solo tendrá que adaptarse al nombre las nuevas clases).

## 6.1.2 Conversión multietapa

La implementación actual del simulador *Packet Switch Simulator* (*PASS*) está diseñada para evaluar tan solo conmutadores trietapa. Sería interesante también estudiar cuales serían los cambios necesarios a realizar para extenderlo a un número variable de etapas.

Algunas de las modificaciones a realizar serían:

- En la clase *GeneratedPacket* en lugar de tener un par de variables para almacenar el momento de entrada en la segunda etapa ("*TimeEntered SecondStage*") y en la tercera etapa ("*TimeEnteredThirdStage*") debería tener un array donde almacenar el momento de entrada en cada etapa. (El tamaño del *array* debería ser de al menos el número de etapas menos uno, ya que al igual que ocurría antes el tiempo de entrada en la primera etapa coincide con el tiempo de entrada en la red de conmutación, y para ello ya se dispone de una variable)
- En la clase *SwitchedPacket* el tamaño del *array* "*outPorts* [ ]" (y del *array* "*outLambdas* [ ]" si se realizó previamente la conversión WDM) vendrá determinado por el número (variable) de etapas.

En general la declaración de las clases y de las interfaces esta ya preparada para la conversión a un número de etapas variable (si observamos las fichas del capítulo 5 veremos que todas las clases mantienen variables o vectores para manejar las etapas, y no una constante de 3 etapas) lo que habría que cambiar es la implementación de algunos métodos y constructores para adaptarlos a un número variable de etapas.

## 6.1.3 Nuevos formas de Interconexión y algoritmos de enrutado

La implementación actual de PASS sólo permite una interconexión total (*fullconnect*) entre sus elementos, y el módulo encargado de tomar las decisiones de encaminamiento de manera global (*FullPathRouter*) solo tiene una implementación actualmente (*FullPathRouter_Random*) que toma las decisiones de encaminamiento

de forma aleatoria. Otra de las mejoras del simulador sería permitir nuevos tipos de interconexión entre sus elementos o nuevos algoritmos de encaminamiento.

Otras formas de interconexión podrían ser:

- Aleatoria: Las conexiones entre un elemento de conmutación y los elementos de conmutación de la etapa siguiente se establecerían de forma aleatoria. (Se pasaría como parámetro la probabilidad de establecer conexión entre elementos de conmutación, con una probabilidad del 100% estaríamos ante una interconexión total o *fullconnect*). Este algoritmo permitiría generar una red de interconexión parcial rápidamente.
- Determinista: Es el usuario el que establece que conexiones se establecerían y cuales no (como parámetro).

En ambos casos los cambios a realizar se encontrarían en la implementación del constructor de la clase *Network* y en la de la clase *UserDefinedParameters*. La clase *UserDefinedParameters* lee el archivo de parámetros y guarda el tipo de interconexión y la información para realizar las conexiones (en caso de ser determinista) en la estructura *NetworkParameters* (incluida en su estructura *SimulationPointParameters*). Esta estructura mantiene un par de variables que indican el tipo de interconexión y el algoritmo de enrutado, en nuestro caso deberíamos definirnos dos nuevos tipos de interconexión y también un campo para almacenar la información en caso de que la interconexión fuese determinista. Por último el constructor de la clase *Network* recibiría *NetworkParameters*, leería el tipo de interconexión y o bien generaría las conexiones de forma aleatoria o accedería al correspondiente campo de *NetworkParameters* para leer como deben realizarse.

Por otro lado otras implementaciones del módulo encargado de tomar las decisiones de encaminamiento de forma global serían (además del aleatorio):

- Secuencial: Los elementos de conmutación de una etapa transmiten de forma secuencial a los elementos de conmutación de la etapa siguiente.
- Secuencial Adaptativo: Los elementos de conmutación de una etapa tienen en cuenta la carga de los elementos de conmutación de la etapa siguiente, y entre los de menos carga transmiten de forma secuencial.

Estos algoritmos ya se encuentran definidos en el campo "*RoutingPolicy*" de la estructura *NetworkParameters* solo hay que implementar su funcionamiento en una clase bajo la interfaz *FullPathRouter* (al igual que *FullPathRouter_Random*) independientemente del algoritmo elegido al final quedará perfectamente establecida la ruta de cada paquete dentro de la red de conmutación (puerto de salida y lambda de salida si aplicable). Hay que tener en cuenta que para tomar las decisiones de encaminamiento ahora se debe conocer perfectamente las conexiones de la red de conmutación (al haber añadido nuevos tipos de interconexión), por lo que al igual que *FullPathRouter_Random* tienen que poder acceder a la estructura *NetworkParameters*.

### 6.1.4 Realizar gráficas de la simulación

Actualmente los resultados de la simulación se presentan en un archivo de texto. Otra de las mejoras a realizar sería poder mostrar los resultados en una gráfica. Para ello habría que terminar de implementar la clase *GraphicGenerator* y realizar algún pequeño cambio en el método "*main*( )".

## 6.2 Conclusión

En la introducción de este proyecto se plantearon diferentes objetivos, en este apartado se analiza si dichos objetivos han sido cubiertos, extrayendo además conclusiones sobre lo aprendido.

Este proyecto fue iniciado con el objetivo de analizar el simulador *PASS* (*Packet Switch Simulator*) mediante el lenguaje de modelado *UML*. Y de extender su funcionalidad a conmutadores ópticos de redes *WDM*. Se estudiaron otros simuladores así como la situación actual de la conmutación óptica. Posteriormente se realizó el modelado *UML* tanto estático como dinámico de nuestro simulador, lo cual nos sirvió para dar una visión clara de su estructura y funcionamiento. Con este mismo objetivo además realizamos unas fichas a partir de la implementación actual del simulador (donde explicar la función de los métodos y variables de sus clases principales). Por último con la información extraída de la arquitectura y funcionamiento del simulador se estudió como abordar las futuras modificaciones a realizar en el mismo para extender su funcionalidad a conmutadores ópticos de redes *WDM*, conmutadores con número de etapas variable, permitir nuevos tipos de interconexión o realizar gráficas con los resultados de la simulación. (La implementación de dichas modificaciones finalmente no fue objetivo del presente proyecto).

Entre las conclusiones sacadas después de este estudio podemos destacar:

- La aplicación del lenguaje de modelado *UML* (y aplicaciones comerciales como *Rational Rose* o *UMLStudio*) como herramienta para describir, construir o documentar software y como punto de partida para realizar modificaciones en el mismo.
- El aprendizaje de un nuevo lenguaje de programación muy útil *C++* (que se suman a los ya conocidos *C, Java* o ensamblador) teniendo así una visión muy clara de los lenguajes de programación actuales. Además se han comprobado las grandes posibilidades que ofrece la programación orientada a objetos. Este tipo de programación se convierte en la manera ideal de modelar todo tipo de sistemas descomponiendo su arquitectura en una serie de módulos, modelando su funcionamiento mediante clases y volviendo a conectarlos entre sí pasando de una arquitectura física a una arquitectura lógica que permite realizar múltiples simulaciones y pruebas de forma eficiente y rápida y que se aproximan lo más cercano a la realidad. Esto puede ser un objetivo de gran importancia perseguido por empresas de todos los ámbitos que trabajan en proyectos de alta envergadura donde el empleo de simuladores y maquetas para su posterior implementación física se convierten en piezas claves a la hora de aumentar la eficacia y la rapidez con la consiguiente reducción de los costes y mejora del resultado final.

# Referencias


[1] *"Optical Networks: A practical perspective"*
R. Ramaswami y K. N. Sivarajan, Ed. Morgan & Kaufmann, 2nd Edition, 2002

[2] *"Multiwavelength Optical Networks: A layered approach"*
T. Stern y K. Bala, Ed Addison Wesley, 1999.

[3] *"Optical WDM Networks: Principles and Practice"*
Sivalingam K., Subramaniam S. Kluwer Academic Publishers, 2000

[4] *Condor Project Home Page.* http://www.cs.wisc.edu/condor

[5] "*Estudio, configuración y prueba de un entorno de computación Condor"*
Maria Victoria Bueno Delgado, Ed.Universidad Politécnica de Cartagena

[6] *Ptolemy project homepage*. http://ptolemy.eecs.berkeley.edu/

[7] *OMNeT++ Community Site* http://www.omnetpp.org/

[8] *Network simulator Berkeley homepage* http://www.isi.edu/nsnam/ns/

[9] *VINT proyect homepage* http://www.isi.edu/nsnam/vint/

[10] *Sigla homepage:* http://sigla.upc.es/

[11] *VISTA homepage* http://vip.cs.utsa.edu/systems/atm.html

[12] *UML. Unified Modeling Lenguage.* http://www.uml.org/

[13*] OMG. Object Management Group.* http://www.omg.org/

[14] "*C Programming Language"*
Bryan W. Kerninghan & Dennis Ritchie, Ed. Prentice Hall

[15] "*C++ Práctico"*
José Canosa, Ed. Universidad Politécnica de Valencia - series Marcombo

[16] *KDevelop*
http://ftp.cdut.edu.cn/pub2/linux/develop/KDevelop/download.html

[18] *The GNU Project and the Free Software Foundation (FSF)*
http://www.gnu.org