CAPÍTULO 12

SIMULADOR DE TAP Y ANÁLISIS DE RESULTADOS

12.1. INTRODUCCIÓN

Podemos considerar que un protocolo es un módulo software que implementa una serie de especificaciones de comunicación para ofrecer un conjunto de servicios a las aplicaciones que los requieran. Desde el punto de vista más tradicional los podemos ver como un conjunto de normas o reglas que rigen la comunicación dentro de una tecnología de comunicaciones, generalmente estratificados en un conjunto de capas, que acaban dando lugar a una pila de protocolos que es atravesada por la secuencia de mensajes (datos y control) que genera una sesión de comunicación en la red.

En nuestro caso, TAP aporta una serie de servicios de comunicación que se han implementado usando múltiples algoritmos distribuidos que interactúan entre sí, como ya hemos tenido la ocasión de comprobar en capítulos previos. El diseño de protocolos de comunicación, como sabemos, no es una tarea trivial, pero la implementación robusta con buen rendimiento de la red y con algoritmos interrelacionados en un SMA se convierte en una tarea realmente compleja. Como hemos podido comprobar en el *Capítulo 11*, la implementación del protocolo TAP, como otros tantos protocolos, divide su actividad en dos importantes aspectos, por un lado el flujo de información a través de la pila de protocolos y a lo largo de toda la red, y por otro, todo lo relacionado con el mantenimiento de la información de estado y control en el mantenimiento del propio protocolo.

Muchas son las ventajas que podemos encontrar en los SMA, pero en general podemos destacar las aportadas en [1] para justificar nuestra decisión de proponer un SMA como forma de solventar los problemas identificados en la tecnología ATM: "*La razón por la que los agentes ofrecen un excitante paradigma para la resolución de problemas es debida a las muchas ventajas aportadas por los sistemas basados en agentes. Los agentes ofrecen un paradigma para la programación remota inteligente. Los SMA ofrecen una vía por la que relajar los inconvenientes de la centralización, planificación y control secuencial para ofrecer sistemas que son descentralizados, emergentes y concurrentes. Además, ayudan a reducir tanto los costes software como hardwar,e y ofrecen soluciones rápidas a problemas aprovechando las ventajas del paralelismo.*"

Aunque en los dos o tres últimos años el campo de los SMA ha experimentado bastante actividad, aun hoy puede decirse que los diseñadores e implementadores de sistemas multiagente y, sobre todo en el ámbito de las telecomunicaciones, son bastante escasos. Disponemos por tanto de una interesante excusa para investigar en materia de ingeniería de protocolos basada en agentes software. Existen en la actualidad múltiples SMA ya implementados y diversas herramientas para la construcción de agentes y de SMA, unas basadas en Java, otras en otros lenguajes, pero las propuestas en el ámbito de los protocolos de comunicación son escasas y poco desarrolladas. En el caso de la tecnología ATM la escasez es aún mayor por lo que nos hemos propuesto buscar una solución a los problemas identificados mediante la especificación de un SMA. No obstante, destacamos que el SMA-TAP es la excusa para investigar en este atractivo campo, pero no hemos de perder de vista que nuestro principal objetivo es buscar una solución válida al problema planteado por las congestiones en los conmutadores ATM, sea solventado por los métodos más tradicionales de la

ingeniería de protocolos, o a través de técnicas inspiradas en SMA. Desde este punto de vista llamamos la atención sobre el hecho que el SMA-TAP se propone como un primer prototipo que ha de ser refinado en investigaciones futuras y, como tal, lo identificaremos en este capítulo y en el siguiente donde comentaremos las líneas de continuidad de nuestras investigaciones en esta materia.

En lo que respecta al flujo de la información, los protocolos gestionan el tráfico que fluye en dos sentidos: desde los usuarios hacia la red, y desde ésta hasta los propios usuarios. Esto lo hemos podido comprobar en el *Capítulo 11*, donde el protocolo TAP en los extremos de la comunicación hace de interfaz entre las aplicaciones y las placas de red de los usuarios. Entre TAP y la red se dispone de la pila de protocolos estándar ATM donde también hemos ampliado las características de la capa AAL-5. No obstante, estas capas estándares del modelo arquitectónico ATM realizan todas las labores del estándar, ya sea en cuanto al flujo de información, o en cuanto al mantenimiento de la información de estado, tal como hemos asumido en los algoritmos ya presentados. En cuanto al flujo de información en los AcTMs también hemos podido ver cómo los agentes intercambian información entre ellos y controlan a la vez el flujo de los datos de usuario. Como cada uno de los agentes está jerárquicamente uno sobre otro, esto permite que sean fácilmente configurables. El conjunto de relaciones entre los algoritmos que intervienen en el flujo de información nos permite establecer un mapa de dependencias y relaciones entre los algoritmos que da lugar a lo que llamamos la tabla del protocolo. En esta tabla las filas pueden ser los protocolos o algoritmos y las columnas representan las relaciones de cada algoritmo con sus vecinos. Como cada algoritmo (o agente) puede ser más o menos dependiente de otros algoritmos (o agentes) la tabla representa las conexiones de unos con otros y, por tanto, sus servicios y relaciones para lo que será necesario disponer de un interfaz genérico inter-protocolo, inter-algoritmos o inter-agentes. En nuestro prototipo con Java, es realizado a través de threads de comunicación inter-agentes que se encargan de mantener la comunicación y las relaciones entre cada uno de los algoritmos. Para cada flujo de datos entrante o saliente en un algoritmo, éste debe identificar el siguiente algoritmo y reenviarle al mensaje para que éste lo procese.

En lo que respecta a la información de estado hay que destacar que la implementación de protocolos requiere mantenerla para cada una de las conexiones establecidas. Esta información de estado suele ser de dos tipos: por un lado información global de toda la implementación del protocolo, y por otro lado, información específica que corresponde a cada una de las conexiones particulares establecidas por los usuarios. En el caso de TAP puede considerarse como información global a todo el protocolo, la identificación de los nodos activos que intervienen, los circuitos virtuales establecidos, o los tamaños de DMTE que intervienen en la red. Podemos considerar como información específica de cada comunicación, todas aquellas que se emplean para caracterizar cada fuente en el proceso de establecimiento de la conexión (grado de GoS, ToS, Ton, Toff, etc.) o, por ejemplo, el control de los PDUid que son propios de cada conexión extremo-extremo.

Debemos señalar también por otro lado las informaciones de control propias de la red y que están también relacionadas directamente con TAP, ya que esta información de control es la que, a través de la información de estado (tanto general como específica), permitirá controlar el flujo de la información de datos. A parte de los aspectos propios de la señalización que intervienen en la comunicación, en nuestro caso podemos también considerar como tal el propio mecanismo de recuperación de pérdidas basado en la generación de BRM como ya sabemos.

La *Figura 12.1* muestra un esquema del agente RCA que es uno de los agentes más complejos del SMA-TAP por las interrelaciones que tiene con otros agentes y con las estructuras de datos propias de los nodos AcTMs. Podemos observar cómo el agente dispone de un control de métodos remotos por los cuáles se pueden invocar desde otros agentes determinadas acciones. Por ejemplo, RCA dispone de un método remoto pensado para atender las peticiones de retransmisión que le llegan desde otros agentes RCA situados en otros nodos activos. Cuando este método es invocado el agente encola la petición en la cola de peticiones que, mediante un thread, es atendida siguiendo el ciclo de vida de éste y según las consideraciones que son explicadas en los *Capítulos 10* y *11*. Del mismo modo se incorporan otros métodos remotos para acciones concretas como la atención de retransmisión proveniente desde el algoritmo EPDR que forma parte del agente CCA y que se desencadena cuando la congestión es local al nodo que implementa el propio agente RCA. Una vez que el control de métodos remotos pasa la notificación a la cola de peticiones, se realiza una notificación al agente que solicita la retransmisión como un mecanismo de confirmación de que el método remoto invocado ha sido atendido.

En este capítulo justificaremos la elección del lenguaje Java para la implementación de protocolos de comunicación y de SMA. Las ventajas destacadas son las que nos han animado a elegir este lenguaje como herramienta para el desarrollo de nuestro prototipo de simulador de TAP (TAPS). En el siguiente punto comentamos la metodología y decisiones de diseño tomadas para el desarrollo de TAPS, poniendo especial atención en el SMA. A continuación se realiza la especificación de las clases, métodos y atributos del SMA-TAP,

sin ánimo de ser extensivos para no distraer la atención sobre el verdadero objetivo de esta tesis. Los dos últimos apartados del capítulo tienen por objetivo analizar algunos de los resultados más significativos obtenidos en las simulaciones realizadas. En primer lugar se presentan varios escenarios de simulación donde podemos comprobar las posibilidades de recuperación de PDU congestionadas, y también demostramos que se cumple la idea intuitiva de aprovechas los estados de inactividad de los enlaces para abordar las retransmisiones. Después se presta atención a la gestión de las colas de entrada, por ser uno de los aspectos de mayor complejidad. El capítulo termina con un apartado de conclusiones.

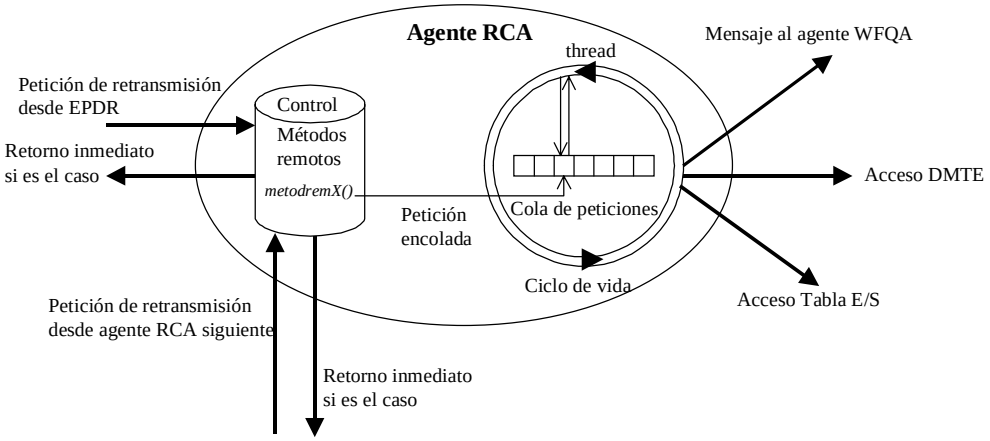

Figura 12.1. *Arquitectura del agente RCA y método remoto para la gestión de mensajes del agente*

12.2. IMPLEMENTACIÓN DE PROTOCOLOS Y SMA CON JAVA

En los últimos años se han propuesto varios lenguajes de programación para la implementación de agentes móviles. Los lenguajes más extendidos realizan diferentes decisiones de diseño como son la determinación de qué componentes de estado de un programa pueden migrar de un nodo a otro. En Java [2], sólo puede migrar el código del programa y el estado no puede ser transportado con el programa. Por otro lado, *Obliq* [3], los valores de función de *first-class* (*closures*) pueden migrar; los *closures* consisten en el código de programa, junto con el entorno que liga variables a valores o posiciones de memoria. Por último, en *Telescript* [4] las funciones no son valores *first-class*, y este lenguaje ofrece operaciones especiales que permiten migrar a los procesos autónomamente. Estos lenguajes también se diferencian en la forma en que transportan objetos entre agentes. Cuando un proceso o un *closure* migra, puede transportar todos los objetos (datos mutables), o dejar los objetos y transportar por la red las referencias a los objetos. Java no permite hacer esto ya que sólo permite la migración del código de programa. En *Obliq*, los objetos permanecen en el nodo en el que han sido creados, y los *closures* contienen referencias de red a estos objetos. Si se desea la migración del objeto es necesario programarlo explícitamente clonando objetos remotamente y borrando después los objetos originales.

El lenguaje Java fue diseñado específicamente para soportar el desarrollo y la distribución de aplicaciones a través de WWW. Sin embargo, con el paso del tiempo el lenguaje se está revelando como una atractiva, potente y potencial plataforma de sistemas ubicuos [5]. De este modo, Java se ha propuesto también como un lenguaje para el desarrollo de protocolos que mejoren el rendimiento y soporten nuevas aplicaciones en todos los entornos. No hay que perder de vista que el mayor inconveniente para el desarrollo de nuevos protocolos y servicios de comunicaciones lo encontramos en la necesidad de actualizar varios millones de sistemas finales y nodos de interconexión con implementaciones compatibles entre todos ellos. Para vencer esta preponderancia de los protocolos actuales lo que se viene haciendo es diseñar protocolos a medida, incorporando en las capas de aplicación código específico para labores muy concretas, lo que impide una vieja aspiración de la programación que es la reusabilidad.

Java aporta sobre todo una importante característica como es la de ser una plataforma que permite el desarrollo de aplicaciones portables e independiente de la arquitectura de los sistemas. Esto, gracias a la aportación de su *virtual machine* está haciendo que el lenguaje se convierta en una potente herramienta para el desarrollo y expansión de protocolos de comunicación y también para la implementación de SMA por su carácter inherentemente distribuido.

En la actualidad se está empleando Java combinado con WWW como una plataforma para el desarrollo de protocolos de comunicaciones. Aunque nuestro objetivo es ligeramente diferente a esta visión, podemos destacar las siguientes características que hacen de Java una atractiva plataforma para el desarrollo de un

prototipo del protocolo TAP:

- Los protocolos, y el código adicional que puedan necesitar para funcionar, puede ser descargado y ejecutado en la propia red cuando sea necesario. Esto permite la interoperatividad y flexibilidad entre los extremos de la comunicación y los propios nodos de interconexión (AcTMs en nuestro caso). Esta característica hace de Java una de las mejores herramientas para el diseño y desarrollo de los conceptos presentados en el *Capítulo 6* sobre los sistemas multiagente y las redes activas.
- Por otro lado, el problema de la portabilidad del código de los protocolos es reducido con el uso de Java tal como se demuestra en [6], gracias a la posibilidad de aprovechar la máquina virtual de Java que es la que realmente se porta en lugar de los protocolos.
- Las características de la orientación a objetos como es la herencia, la especialización y el polimorfismo aportan nuevas posibilidades a los programadores y facilitan la reusabilidad del código.
- No obstante, estas ventajas tienen en la actualidad un coste debido a que la máquina virtual es emulada en software y, mientras no se emplee soportada en hardware, esto implica una apreciable caída en el rendimiento de la red, lo que es importante en el caso del desarrollo de protocolos.

En la actualidad Java está formada por dos entidades separadas: un lenguaje de programación [7] y por debajo de éste una especificación de máquina llamada JVM (*Java Virtual Machine*) [8]. Estas dos partes juntas pueden entenderse análogamente a lo que podemos encontrarnos al combinar el lenguaje de programación C y la arquitectura Sparc. El lenguaje de programación es, por sí mismo, orientado a objetos, y más simple que otros como C++ y Smalltalk.

JVM interpreta los bytecodes Java que son instrucciones ensambladas por una arquitectura de CPU abstracta. Cuando el código escrito en Java es compilado en formato bytecode interpretado por la JVM, los programas Java podrán ser ejecutados en una amplia gama de arquitecturas de computadores sin necesidad de recompilación previa, lo que aporta esa característica tan importante de portabilidad de Java. La mayor ventaja del formato en código binario es que es independiente de las arquitecturas hardware, de los interfaces de sistemas operativos y de los entornos de GUI (ventanas). Por tanto, el compilador de Java no genera código máquina relativo al juego de instrucciones del hardware nativo, sino que genera bytecodes independiente de la máquina real, para una máquina hipotética que es implementado por el intérprete Java y el sistema *run-time*.

El lenguaje de programación Java dispone de una serie de librerías de clases predefinidas para la manipulación de interfaces gráficos, flujos básicos de E/S, utilidades como la gestión de tablas de hashing y vectores, y también relativos al networking. El soporte estándar, en cuanto al networking se refiere, está limitado a los servicios que usan los protocolos TCP y UDP y algunas otras funciones relacionadas. El lenguaje también aporta clases para la comunicación entre servidores de web y todos los aspectos relacionados con estas funciones.

Sin embargo, Java permite la extensión de nuevas clases que pueden ser cargadas dinámicamente en la JVM cuando son necesarias. Esto nos va a permitir la implementación de nuevas clases que constituyan los diferentes subsistemas del protocolo TAP. Como puede observarse en la *Figura 12.2*, los programas Java son aplicaciones que se ejecutan como intérpretes independientes y autónomos [1]. En la Figura puede observarse el cargador de clases, desde el que la JVM puede ofrecer la posibilidad de cargar y añadir dinámicamente nuevas clases en el sistema. El cargador de clases recibe las peticiones de nuevas clases y después las carga desde ficheros (si el intérprete que usa la clase se está ejecutando como una aplicación independiente) o desde la red. En nuestra propuesta de prototipo de TAP se emplea la capacidad para cargar y ejecutar protocolos (algoritmos) “en el aire”, o que son aportados por el cargador de clases.

En los protocolos desarrollados en lenguajes de programación tradicionales como C se emplean de forma habitual definiciones comunes a varios programas recurriendo a ficheros de cabecera (ficheros *.h*) y también constantes globales y preprocesado de símbolos. Pero Java no soporta ninguna de estas posibilidades ampliamente usadas en C, por lo que los ficheros cabecera con definiciones, constantes y máscaras de bits comunes no pueden usarse. Pero en la práctica este tipo de limitaciones no es ninguna restricción ya que el programador puede declarar constantes y definiciones como clases de variables en aquellas clases que sean compartidas. Por ejemplo, en el caso de TAP empleamos constantes como los VPI/VCI y también los nombres de los agentes en una clase, de forma que otros protocolos o algoritmos puedan referenciarlos.

[1] Si en la *Figura 12.2* se cambian las aplicaciones por *applets* y se añade un navegador de web entre el SO y la JVM tendremos aplicaciones Java ejecutándose restringidas a un intérprete que estará contenido en el navegador de web específico que esté instalado.

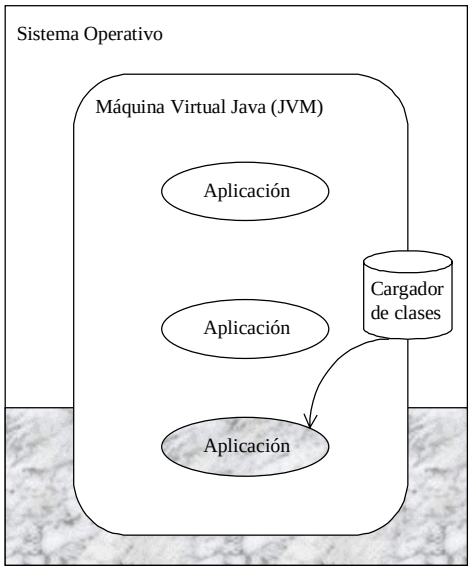

Figura 12.2. *Ejemplo del modelo de arquitectura de la JVM como sistema independiente*

Colocando definiciones comunes, máscaras de bits y constantes en una clase Java dispondremos de varias ventajas [9]. Primero, si nada cambia, los protocolos que ya hayan sido desarrollados en otros lenguajes de programación deberán ser completamente compilados y redistribuidos a los sistemas afectados. En los protocolos basados en Java, cambiando constantes, máscaras de bits o definiciones comunes sólo se vinculan cambiando la clase apropiada y descargando las versiones antiguas de todos los sistemas. Los protocolos que dependen de esta clase sólo necesitan actualizar el cambio la próxima vez que referencian a la clase. Segundo, los protocolos que emplean este planteamiento para compartir información de configuración, pueden ver modificado su comportamiento de forma más sencilla desde otros protocolos, algoritmos o desde las propias aplicaciones de los usuarios. Esta es precisamente una de las más atractivas características de Java para diseñar redes programables bajo las premisas de los agentes software o de las redes activas.

La implementación de la estructura de un protocolo identifica cómo éste interactúa con otros protocolos y con el resto del sistema. Por esto define los interfaces y métodos para construir la tabla del protocolo que comentamos anteriormente, y la forma en que se ejecuta el protocolo. Así, la estructura del protocolo definida para el uso con protocolos basados en Java es similar a la implementación tradicional donde se definían las entradas, salidas y los métodos de control. Sin embargo, como se destaca en [5], la estructura de protocolos basados en Java difiere de los métodos tradicionales en varios aspectos y en nuestro caso particular hemos de destacar los siguientes aspectos de interés que nos afectan más directamente:

- Los protocolos basados en Java se implementan como clases que son instanciadas por los usuarios o por otros protocolos que requieran de sus servicios.
- Las clases de los protocolos heredan su estructura de protocolo de una clase de protocolo base. El uso de la herencia para imponer la estructura del protocolo permite a éstos aprovechar el soporte del lenguaje para la programación estructurada, permitiendo al programador ampliar sus funcionalidades a través de la especialización, mientras se soporta la reusabilidad del código del protocolo.
- La estructura del protocolo puede definir métodos explícitos para la composición de protocolos en tablas. Por ejemplo, en el caso del SMA-TAP usamos estos métodos para identificar protocolos vecinos superiores e inferiores a un protocolo dado para identificar la jerarquía del SMA completo dentro de un AcTMs. Así, *protprev* y *protsig* identifican en la tabla de protocolos al protocolo vecino anterior y posterior respectivamente. En la estructura del protocolo pueden emplearse también métodos explícitos como *FinConex()* para cerrar una conexión concreta ya que Java no dispone de memoria explícita, ni permite el uso de punteros ni de desasignación de objetos.
- Por último, disponemos de la ventaja del uso del multithreading, de forma que se pueda emplear un hilo por objeto de protocolo. El soporte para la construcción de hilos es una de las herramientas más potentes de Java para la ejecución de eventos concurrentemente. El multithreading es la forma de conseguir rapidez y concurrencia de una forma sencilla con un solo espacio de proceso. Java aporta en su librería la clase *Thread* con un completo conjunto de métodos que permite que un thread sea: arrancado, parado, ejecutado, chequeado, etc. Además, puede hacerse uso de primitivas de

sincronización basadas en el paradigma monitor y condición variable. En el caso de TAP el uso de los hilos es una de sus principales características ya que, cada uno de los agentes del SMA dispone de hilos de comunicación con los agentes con los que interactúa, de forma que estos hilos se emplean para el intercambio de información de comunicación (conversación) entre los agentes, y otros para la transmisión de información de datos (células y PDU) que intercambian entre sí. Los hilos se emplean para el estudio del comportamiento periódico; para el procesamiento específico de cada protocolo como la creación y mantenimiento de las tablas de rutas en el establecimiento de la conexión; para la asignación de VPI/VCI; para el mantenimiento de las colas de entrada, de la DMTE, de las tablas de E/S y también para el mecanismo de solicitud de retransmisiones mediante las células BRM. Este tipo de procesamiento periódico del rendimiento del protocolo simplifica su código y evita el uso de los clásicos *timeouts* en el diseño de protocolos. Los hilos aportan un importante paradigma a la programación con Java porque cada hilo posee su propia pila de llamadas, por lo que pueden ser activados múltiples hilos con cada protocolo o algoritmo de forma simultánea sin interferir con ninguno de los otros. Cuando un paquete concreto no puede ser procesado (cola llena, tablas llenas, sincronización rota, etc.) el hilo puede mantener el paquete autosuspendiéndose (con el consiguiente efecto sobre el rendimiento final) hasta que el procesado del paquete pueda continuar en lugar de encolar el paquete, activar un *timer* y terminar. En resumen, el soporte de los threads es una de las herramientas más poderosas de Java, no sólo para aportar la interacción entre aplicaciones gráficas, sino por poder ejecutar concurrentemente múltiples eventos. El soporte de los threads de Java incluye un sofisticado conjunto de primitivas de sincronización basadas en un potente y ampliamente usado paradigma monitor y variable condición. Para que un programa software sea autónomo, éste debe ser un proceso separado o un thread. En una aplicación multiagente, un agente puede ser un thread de control separado y lo normal es que cada agente tenga su propio thread. Cuando usamos threading la ventaja clave es que podemos conseguir ejecuciones simultáneas. En realidad, el multithreading se usa por cada entidad funcional en un agente como son comunicaciones entrantes y salientes.

Para una aplicación multithreading como es un SMA, es necesaria la sincronización entre los threads de agentes diferentes. En Java se usan monitores para coordinar múltiples threads usando los métodos *wait()* y *notify()* disponibles en cada objeto Java. Un thread espera por la ocurrencia de alguna condición o evento, y notifica un *thread wait* cuando se produce una condición o evento. Los threads son coordinados usando el conocido concepto de variable condición que es un estado lógico que debe mantenerse a *cierto* para que un thread pueda comenzar a actuar. Si la condición no se mantiene a *cierto*, el thread debe esperar hasta que la condición se convierta en *cierta* antes de continuar. En Java este comportamiento podría expresarse con el código presentado a continuación y usado en el agente RCA para que éste se mantenga a la espera mientras no se produzca una notificación de congestión en el buffer.

```
While (NoNotificacionCongestion) {
        wait();
}
```

Los métodos *wait()* y *notify()* deben ser invocados desde un método sincronizado o desde un estado sincronizado. Estos dos métodos simplifican la tarea de coordinación de múltiples threads en un programa concurrente escrito en Java. El siguiente fragmento de código presenta un ejemplo de cómo puede ser usada la variable de control en el agente DPA para esperar por una decisión hecha sobre el estado de la DMTE donde se invoca a una llamada a método remoto el cual llama al método *EstadoDecision()*. De este modo, DPA espera mientras no se reciba una notificación de retransmisión desde la DMTE.

```
Public sincro void SincEstadoDecision(int estado)  {
        EstadoDecision = estado;
        notify();
}
sincro(preparado) {
        if (EstadoDecision = = NoPreparado)
        wait();
        }
}
```

- Otra atractiva ventaja del uso de Java para la programación de agentes y protocolos es su posibilidad de emplear RMI (Remote Method Invocation) para crear aplicaciones Java-to-Java distribuidas, en las cuales los métodos de los objetos Java remotos puedan ser invocados desde otra máquina virtual que esté situada en otro nodo. En nuestro caso no hacemos uso de RMI, ya que el TAPS se ejecuta en su actual versión únicamente en un nodo, pero ésta es una de nuestras investigaciones futuras en un intento por conseguir un escenario real y plenamente distribuido del protocolo.

- El carácter orientado a objetos es otra de las características de Java para obtener código reusable y para encapsular todos los datos en objetos. Esta es una ventaja que empleamos ampliamente en el simulador con múltiples objetos que se van a usar en las transferencias y conversaciones.

- Además, Java mediante métodos nativos permite llamadas a código ya compilado de otros lenguajes (código nativo) optimizando la velocidad del código que implementa un agente concreto. El interface Java Native Method Interface permite a los programas que ejecutan la Java Virtual Machine (JVM) llamar a programas de interfaz escritos en C y C++. Esto permite que los programas escritos en otros lenguajes puedan arrancar la JVM y ejecutar programas Java.

- Los protocolos basados en Java pueden elegir si las entidades como mensajes, capas, conexiones, buffers, colas, tablas y tiempos son limitados a la capa de planificación. En bastantes casos se han implementado los paquetes acompañados a través de los protocolos mediante hilos no apropiativos, lo que quiere decir que cada protocolo dispone de su propio hilo para el envío de los flujos de datos hasta el siguiente protocolo o la apropiada estructura de datos. Es decir, cuando una aplicación, o uno de los algoritmos del conjunto de protocolos, necesita enviar datos, lo que se hace es crear un buffer al que se asigna un hilo de comunicación para acompañar los datos hasta la tabla de protocolos, de forma similar a como se hacía tradicionalmente con otros lenguajes de programación.

- Tanto para la implementación de protocolos como de SMA, Java aporta otra ventaja adicional que es la de poder usar una clase pensada para los sockets de datagramas multicast (*MulticastSocket*) que permite la formación de grupos multicast. Del mismo modo se puede emplear también para el envío de mensajes de difusión broadcast que pueden ser muy útiles para la difusión de acciones concretas a un grupo determinado de agentes. En la versión actual del SMA-TAP no se aprovecha esta posibilidad, pero es otra de las mejoras prevista para futuras versiones.

- La seguridad es otra de las cuestiones en el ámbito de los SMA y de los protocolos de comunicación, y en este sentido Java ha sido mejorado para aportar fiabilidad a las comunicaciones, incluso a través de técnicas de cifrado.

En las arquitecturas de protocolos basadas en Java, suelen usarse clases *servicio* especiales que construyen dinámicamente las tablas de protocolos en tiempo de ejecución cuando las aplicaciones necesitan servicios concretos de comunicación. Por ejemplo, si una aplicación necesita el servicio GoS únicamente tendrá que crear una instancia de su correspondiente clase *servicio*. El código de la clase servicio se encarga de asignar e inicializar las capas de protocolos e interconectarlas a través de los métodos ya comentados *protprev* y *protsig*. De este modo la clase *servicio* puede entenderse como un *daemon* del estilo de Unix que se encarga de la construcción y mantenimiento de una tabla particular de protocolo [6].

El cargador de clases de Java se emplea habitualmente en la construcción de la tabla de protocolos porque ofrece la posibilidad de incorporar nuevas clases de protocolos (tal como se hace con las clases ordinarias de Java) en la JVM. Las clases *servicio* pueden referenciar explícitamente a una clase de un protocolo concreto referenciando directamente su nombre. Si la clase no está ya cargada en la JVM, el cargador de clases del sistema lo hará de forma transparente al usuario y a los protocolos (cargando la clase directamente de un fichero) sin la intervención de la propia clase *servicio*.

Las clases *servicio* y otros protocolos basados en Java pueden cargar también explícitamente clases Java ellas mismas. Por ejemplo, envían a las otras clases código de clase Java como datos transmitidos vía conexión de transporte normal. Estos datos pueden ser convertidos en una clase Java ejecutable usando una de las facilidades de construcción del cargador de clases. Esta característica extensible de los protocolos con Java permite la introducción o reemplazamiento “en el aire” de nuevo código de protocolo, lo que podemos emplear para la implementación de los agentes programables en el caso de TAP.

Por otro lado, la tabla de cada protocolo puede ser colocada como una clase de información del protocolo o dentro de la información de estado de cada instanciación individual de esa clase. Si se hace en la clase fija esta tabla de configuración de protocolo se usa para cada instanciación de objetos del protocolo. Si se sitúa la tabla de configuración del protocolo en el objeto permite a cada sesión individual determinar y controlar su propia tabla de protocolo. En el caso de TAP, y con la intención de que cada algoritmo y conexión pueda

elegir los servicios que requiere de una forma flexible en sus comunicaciones, hemos decidido situar la configuración de los protocolos en la información de estado de cada sesión de los objetos, en lugar de hacerlo en una clase propia para ello. Esto lo hacemos a través de las operaciones de sincronización especificadas en los algoritmos que presentamos en el *Capítulo 11*. De este modo, se permite la existencia de múltiples tablas de protocolos conteniendo los mismos protocolos pertenecientes a agentes del SMA.

En lo referente al rendimiento de los protocolos implementados en Java cabe destacar que el hecho que la JVM sea implementada en software y que el propio lenguaje sea orientado a objeto, hace pensar que el rendimiento de los protocolos desarrollados en Java y compilados en bytecode Java pague un importante precio en cuanto a su rendimiento para disponer de la portabilidad y extensibilidad del lenguaje. Estos costes han sido evaluados en [6] comparativamente con lenguajes más tradicionales como C y Streams, en función de varios aspectos como el tiempo de procesamiento de los paquetes, el overhead debido a la recolección de restos de memoria libre (*garbage collector*), y a las técnicas de compilación no demasiado depuradas. Cabe destacar que la gestión de memoria en los programas escritos en Java es bastante más simple que otros lenguajes de programación, porque la memoria no necesita ser desasignada explícitamente debido a que el recolector de *garbage* de la máquina virtual se encarga de reclamar la memoria libre. Esta forma de actuar permite simplificar el código programado por poder obviar las operaciones de asignación y liberación de memoria típica en otros lenguajes; sin embargo, acaba pagándose también un precio en rendimiento por esta simplificación para el programador, ya que la máquina virtual debe suspender periódicamente la ejecución de los programas para localizar la memoria libre no utilizada. Esta forma de actuar acaba afectando por tanto al rendimiento de los protocolos y, sobre todo, a los que deben soportar servicios en tiempo real, ya que el recolector de *garbage* es invocado sólo cuando la máquina virtual está baja de memoria, y esta condición ocurre, desgraciadamente, de una forma no determinística.

Por tanto, la mayor parte de estudios de rendimiento de Java se centran en el intérprete de la JVM y en el propio lenguaje Java. No obstante, este tipo de estudios es realmente difícil de realizar debido a que Java se encuentra aún en su primera etapa como lenguaje, comparado con otros, y también debido al comportamiento de las diferentes plataformas sobre las que se puede ejecutar. Los resultados mostrados en [6] indican que la latencia introducida por Java en el procesamiento de un paquete con respecto a otros lenguajes nativos como C/BSD, Streams y x-Kernel (basados en código nativo compilado y ejecutable directamente), incrementa en un factor de 10 en el envío, y de 17 en la recepción del paquete. El resultado del estudio de rendimiento debe mejorar a medida que evolucione el propio lenguaje Java y la tecnología de su intérprete en la JVM (basada en técnicas de compilación *just-in-time*), y de hecho los estudios realizados demuestran que el rendimiento de JDK v. 1.1.1 ha doblado el rendimiento respecto a su versión previa. En realidad, la gran desventaja en cuanto a rendimiento está en la diferencia entre el código Java interpretado y el código nativo de otros lenguajes que es compilado y ejecutable directamente. De hecho, los estudios de rendimiento del uso de Java en el desarrollo de protocolos demuestran que, sobre una misma plataforma hardware, la compilación JIT (*Just-In-Time*) aporta mejoras sobre versiones como JDK (Java Development Kit), ya que JIT realiza compilación en tiempo de ejecución del bytecode de Java que es directamente usable por el sistema operativo que está debajo, y esto permite reducir parte de los problemas de rendimiento.

Por todas estas razones generales y por otras más particulares que podrían ser entendidas más puramente dentro del contexto del desarrollo de protocolos y de sistemas multiagente, se decidió adoptar el lenguaje Java para la implementación del simulador de protocolo TAP y, por tanto, el sistema multiagente que propuesto en esta tesis. Aunque Java es una de las posibles opciones, en la actualidad es el líder de los estándares de facto y el que han usado gran parte de los fabricantes de *networking* como Cisco ó 3Com, aunque tenga algunos aspectos mejorables como la portabilidad, seguridad y sus características de rendimiento en comunicaciones. Sin embargo, el lenguaje Java se ha demostrado que puede aportar atractivas ventajas para la construcción de sistemas multiagente complejos.

12.3. METODOLOGÍA Y DECISIONES DE DISEÑO DE TAPS

Hemos podido ver ya las ventajas más importantes del uso del lenguaje Java para el desarrollo, tanto de protocolos, como de SMA. Con este punto de partida, en este apartado se discute la metodología y las decisiones de diseño seguidas para el desarrollo del SMA-TAP que es la principal parte del TAPS. Así debemos de conseguir la comunicación, coordinación y cooperación entre los agentes para lograr una solución coherente.

Precisamente uno de los principios de diseño deseados es el de huir de la excesiva centralización y, tal como hemos podido comprobar en el capítulo anterior y en los precedentes, en el caso de TAP no se dispone de un control central en el sistema, aunque partimos de una tecnología inherentemente orientada a la conexión, por lo que se depende de ciertos aspectos como es el control de los extremos de la comunicación

con algunas características que han de mantenerse en toda la conexión. Este objetivo de conseguir un sistema lo más distribuido posible nos ha llevado a emplear los métodos de comunicación directa entre los agentes. El método de federación es más apropiado para SMA con un gran número de agentes y el método *blackboard* es claramente centralizado. Con el modo de comunicación directa aportamos a TAP las posibilidades básicas en todo SMA. Como se ilustra en los algoritmos del *Capítulo 11*, en algunos casos se necesita comunicar con varios agentes a la vez por lo que el modo de comunicación por difusión es el adecuado, pero esta posibilidad la resolvemos con comunicación directa ya que en todos los casos se trata de comunicación entre pares de agentes por lo que el método directo es el más apropiado.

La metodología de diseño del SMA-TAP que hemos empleado está basada en [1,10] y se divide en las siguientes etapas:

- Identificación de los agentes: En esta etapa realizamos el análisis del sistema y de los objetivos que perseguimos y que, en esencia, ha sido ya descrito en los *Capítulos 6*, *10* y *11* donde hemos identificado todos los agentes que intervienen en el sistema. Tenemos también ya una lista de las funciones particulares que desempeñan cada uno de los agentes y del objetivo general del sistema completo. De este modo tenemos definido un conjunto de entidades que interactúan unas con otras para lograr ese objetivo general, tal como se ha descrito en el *Capítulo 10*. Estas entidades identificadas son las que van a constituir los agentes del sistema, en los que se han aclarado los objetivos de cada agente y también los servicios que aportan al SMA. Además, hemos dividido ya esos agentes en grupos según sus categorías o clases como agentes programables (CoSA y CCA), agentes autónomos (DPA) y colaborativos (CoSA, RCA y WFQA).

- Identificación de las conversaciones entre agentes. El problema de la coordinación puede ser entendido conociendo los procesos de interacción entre los agentes del SMA. Pero este conocimiento ha de ser sobre la capacidad para resolver el problema completo desde el punto de vista de todo el SMA y no desde el de un agente concreto. Puede verse el sistema como un conjunto de elementos, de modo que la interacción entre esos elementos puede entenderse como un modo de conversación. O más concretamente, una conversación puede entenderse como la planificación de un agente para conseguir un objetivo, basándose para ello en la interacción con otro agente [1]. Este tipo de conversaciones suele modelarse mediante autómatas y, en nuestro caso, hemos decidido explicarlo sin excesivos formalismos en el *Capitulo 10*, donde vieron las posibles conversaciones que cada uno de los agentes puede emprender. Así se tienen las diferentes clases de conversaciones del sistema y las clases de aplicaciones específicas que puede usar cada conversación.

- Identificación de las reglas de conversación, que permite fijar las normas de establecimiento de conversaciones entre agentes; es decir, una determinada acción produce un estado en la conversación, y el estado actual de la conversación puede influir en el siguiente estado del agente. Esta etapa puede formalizarse también mediante autómatas, aunque nosotros hemos especificado este tipo de precedencias entre estados y conversaciones de los agentes literalmente en el *Capítulo 10* y en forma de algoritmo en el *Capítulo 11*.

- Análisis del modelo de conversación ya especificado en la etapa 2, que intenta encontrar incoherencias en el sistema. Analizando la coherencia de cada agente se llega a garantizarla en todo el sistema. Cuando se detectan inconsistencias en el sistema se vuelve a la etapa 2 para redefinir las conversaciones. Para todo este análisis suelen emplearse máquinas de estados finitos y redes de Petri y, en nuestro caso, hemos podido comprobar este análisis mediante trazas del funcionamiento del sistema que han sido presentadas en el *Capítulo 10*.

- Implementación del SMA que es el último punto de esta metodología, donde debe elegirse un lenguaje o una herramienta de implementación del SMA que sea capaz de asegurar la comunicación, interoperación y coordinación entre todas las entidades del sistema. Los lenguajes (o herramientas de elaboración de SMA) deben permitir soportar el tipo de la comunicación deseada (directa, broadcast, federación o *blackboard*), la creación de agentes y de sus hilos, los mensajes de comunicación, la iniciación de la conversación de cada agente en su hilo y el desarrollo de reglas de conversación. Los lenguajes como Java pueden verse como este tipo de herramientas que aportan las características que hemos podido comprobar en el apartado anterior para la implementación de los SMA. No obstante pueden emplearse también herramientas para el desarrollo más sencillo de SMA como JAFMAS [1], JATLite[10] y otras tantas que suelen aportar un conjunto de clases Java ya implementadas que pueden ser usadas para la implementación de SMA.

La capacidad de comunicación es la que permite a los agentes de un SMA coordinar sus acciones y cooperar con otros agentes. Generalmente la comunicación puede establecerse para el intercambio de

mensajes de forma directa, o bien a través de un sistema federado, mediante mensajes tipo broadcast o a través de un mecanismo de almacenamiento de datos compartidos donde se almacenan y recuperan los mensajes. Como hemos indicado antes, en nuestro caso implementamos el SMA basado en un sistema de comunicación directa sin tener en consideración los aspectos de enlaces físicos directos ya que el SMA se implementa en un solo ordenador en el que se simula el comportamiento de TAP. No obstante, en el mecanismo de inicialización de la comunicación podemos considerar que empleamos el sistema de comunicación de mensajes broadcast ya que, como se vió en el *Capítulo 11* (*Figura 11.1*), al ejecutarse el algoritmo TAP en el terminal emisor se realiza la inicialización de todo el SMA mediante el establecimiento de un mecanismo de sincronización con cada uno de los agentes que intervienen en la comunicación. El mecanismo de comunicación empleado a menudo depende de las dependencias del sistema y de las circunstancias en las cuales los agentes intercambian información. Existen tres mecanismos básicos de comunicación: llamada a procedimiento, *callback* y *mailbox*. En TAP se emplean llamadas a procedimiento.

Puede usarse también un tipo de mecanismo *mailbox*, un modelo cola y servidor. En un entorno de red con sus retardos inherentes, los agentes no pueden ofrecer esperas en torno a algo que vayan a necesitar en un punto futuro. De esta forma, los mensajes entrantes son colocados en las correspondientes colas. El servidor chequea cada mensaje de la cola por turnos o acordando una prioridad predefinida, por ejemplo, manipulando solicitando manipulación de conexiones antes de la configuración de la conexión. Este es el tipo empleado por el agente WFQA para recibir las notificaciones de retransmisión que le llegan desde el agente RCA.

La cooperación entre agentes permite crear una comunidad de agentes especializados que aúnan sus recursos y capacidades con la intención de solventar problemas complejos, pero con el coste adicional del overhead en los aspectos de comunicación. La comunicación activa los agentes en un sistema multiagente para intercambiar información bajo las bases de que cada agente coordina sus acciones y coopera con otros. Cuando los agentes necesitan conversar con otro agente, pueden hacerlo de varias formas. Pueden hablar directamente con otro agente, hablando el mismo lenguaje. O pueden hablar a través de un facilitador, ofreciendo una forma para comunicarse con él, y éste puede hablar con el otro agente. El SMA no emplea facilitadores, ya que cada par de agentes conversa de forma directa.

La interacción es otra implementación importante en un sistema multiagente, como el proceso de interacción que permite a varios agentes combinar sus recursos y acciones para alcanzar sus objetivos basándose en un concepto puramente distribuido. Sin embargo, la naturaleza distribuida y heterogénea de los sistemas multiagente hace que la implementación de la interacción entre agentes sea realmente compleja. Precisamente, trabajando en un SMA de agentes distribuidos lo más complejo es diseñar un mecanismo de comunicación o lenguaje que permita a un agente comunicar con los demás intercambiando mensajes y esto [1] es lo que diferencia precisamente a los agentes de los objetos.

El comportamiento de un agente puede caracterizarse como una máquina de estados finitos, la cual describe la interacción desde el punto de vista de un agente individual con todas sus posibles interacciones, en las cuales el agente puede establecer contactos con otros agentes. Sin embargo, los agentes son desconectados de otros teniendo un thread especial y poniendo las solicitudes en la correspondiente cola de mensajes. Cada cola puede intercambiar mensajes con conversaciones estructuradas con otros agentes, cambia de estado y realiza acciones.

La coherencia y la coordinación son dos más de los aspectos básicos en los SMA, mediante los cuales se establece en el caso de la coherencia el comportamiento completo del sistema al resolver problemas concretos para los que ha sido diseñado; y en de la coordinación la posibilidad de que un conjunto de agentes puedan interactuar para realizar acciones colectivas como las que se realizan entre RCA, CCA y WFQA para ser capaces de recuperar las PDU que se pierden en las congestiones.

12.4. ESPECIFICACIÓN DE LAS CLASES DEL SIMULADOR TAPS

El punto anterior ha presentado el diseño del SMA-TAP. El simulador de TAP (TAPS) se ha desarrollado en JDK (Java Development Kit) 1.1. En este apartado vamos a especificar (obviando los procesos de formalización de todos estos procesos) y centrándonos principalmente en identificar las clases propuestas y sus procesos de comunicación y coordinación y obviando los aspectos lingüísticos y semánticos.

Las clases de TAPS soportan agentes con múltiples hilos (uno para el propio agente, y uno por cada una de las conexiones que mantiene el agente con otras entidades).

A continuación se presenta una relación de las clases de mayor interés empleadas para la codificación del simulador. Estas clases son añadidas al cargador de clases de la JVM de forma que podrán ser empleadas por cada una de las entidades que necesiten usarlas para su ejecución. Se han reinterpretado algunas de las clases

propuestas en [1] y [10] y, a la vez, hemos incluido en el diseño nuestras propias clases que conforman la mayor parte del SMA. Se destaca en algunas de las clases propuestas sus atributos y métodos para clarificar su instanciación.

- *TAP*: es la clase principal que llama a los procedimientos de inicialización del resto de las clases que se presentan a continuación.
- *CrearAgente:* Esta clase permite la creación de los agentes del sistema en el proceso de inicialización del SMA-TAP. Es decir, cuando se arranca el protocolo TAP, la primera operación que realiza es la creación de los agentes del SMA. Para ello esta clase dispone del método *CrearAgente(),* que emplea el nombre de cada agente y sus atributos y, a continuación, se crea y se inicia (mediante el método *inicia*) arrancando sus correspondientes hilos de comunicación.
- *Mensajes:* Los agentes del SMA emplean esta clase para su comunicación entre ellos. Pueden observarse los atributos y métodos de la clase en la *Figura 12.3*, donde puede verse una variable *acción* que es usada en cada instanciación de la clase. Esta variable se usa para definir la acción interactiva que se va a establecer como *aceptar* y *rechazar*. Los métodos *SincEmisor()* y *SincReceptor()* se emplean para especificar los emisores y receptores de los mensajes. Por otro lado, el método *SincIntencion(),* se emplea para que el emisor pueda notificar la intención o el propósito de envío de mensaje, y el agente receptor para poder recibir esta intención. Pueden enviarse y recibirse mensajes de dos tipos: de *sincronización* y de *contenido*. Los de *sincronización* se emplean para establecer y mantener las comunicaciones entre los agentes con las acciones y estado de éstas. Los mensajes de *contenido* donde se indica la información que se envía a través de los mensajes. La variable *contenido* permite a los agentes especificar el contenido de sus mensajes, de forma que un mensaje *contenido* llevará la información que un agente va a enviar y el tipo de objeto a que pertenece y, para ello, emplea el método de sincronización *SincContenido().*

Clase Mensaje	
Variables	**Métodos**
accion *emisor* *receptor* *contenido*	*Mensaje(accion)* *Mensaje(mensaje_recibido)* *SincReceptor* *SincContenido* *SincIntencion* *TomarAccion* *TomarEmisor* *TomarReceptor* *TomarIntencion* *TomarContenido*

Figura 12.3. *Clase Mensaje del SMA-TAP*

- *ColaMensaj:* El SMA-TAP emplea esta clase para mantener una cola de mensajes cruzados entre los agentes. Esta clase dispone de métodos para mantener la cola y para realizar búsquedas: *AñadMensaj(), BorrMensaj(), BuscPorEmisor(), EsVacia(), Tamaño(),* etc.
- *PDU*: es una clases genérica de la cual heredarán todos los tipos de paquetes empleados en la aplicación. Una PDU es una estructura del tipo matriz en la que se van insertando datos de tipo entero, short, string, etc. Entre otros cuenta con el método *siguientePDUid().*
- *CélulaATM*: clase que cuelga de PDU, con las funciones y campos de las células estándares ATM.
- *General*: es una clase creada para contener todas las constantes de la aplicación y todos los procedimientos útiles, como la conversión de datos.
- *EAAL5***:** Clase que representa la capa EAAL-5 en los nodos emisor y receptor.
- *ATM:* Clase que representa la capa ATM en los nodos emisor y receptor.
- *Fisica:* Clase que representa la capa Física en los nodos emisor y receptor.
- *Fichero*: emplea la clase *File* de Java, y añade funciones específicas de la aplicación para leer y escribir ficheros de texto (leer línea, leer cadena, leer número, etc.).

- *Ventana*: representa el contenido de la ventana de datos genéricos.
- *VentanaAcTMs*: representa el contenido de la ventana de datos concretos del nodo activo.
- *Config*: clase para la ventana de edición de los ficheros de configuración.
- *Comunicacion*: Como se ha comentado, el SMA emplea el método de comunicación directa para lo que se dispone de esta clase que se emplea para enviar y recibir mensajes, tanto en los procesos de inicialización del SMA cuando se establece una comunicación con GoS, como entre los propios agentes que forman parte del SMA-TAP. Cada uno de los agentes dispone de una instanciación de esta clase que realiza la función de enviar y recibir mensajes directos. Para ello la clase dispone del método *SincMensajRcbd()* para recibir mensajes de otros agentes que, además, es usado por los agentes del SMA para enviar mensajes al agente al que pertenece el objeto asociado *Comunicacion*. Esta clase dispone también del método *EnvMensaj()* para enviar mensajes directos desde el agente. Los agentes envían mensajes, instanciando la clase *Mensaje*, como un argumento del método *EnvMensaj()*. Asociado a esta clase existe un hilo de comunicación, extendiendo la clase *Thread* incluida en el lenguaje Java de forma que cada objeto *Comunicacion* dispone de un hilo de ejecución separada, de modo que los agentes tendrán un objeto hilo *Comunicacion* separado.
- *ReglasConv*: Es la clase que se emplea para establecer las reglas de conversación entre los agentes que describen las acciones a realizar cuando una conversación entre agentes se encuentra en un estado concreto. Cada regla tendrá un estado en cada momento y un estado siguiente al que se llega en la ejecución. Esta clase se encarga de activar el mensaje recibido y el que va a ser transmitido a través de los métodos *SincRcbdMensaj()* y *SincTrmtdMensaj()*. El método *ControlRegla()* se ejecuta continuamente sobre la lista de reglas a través de un atributo *regla* y, cuando coincide con el atributo del estado actual de una regla, se usa el método *Invocar()* de la clase para ejecutar esa regla. Cuando esto ocurre se emplea el método *Transmitir()* para enviar un mensaje, que puede ir precedido y/o seguido de otras acciones (a través de los métodos *Antesde()* y *Despuésde()*) en función del mensaje de que se trate. Por ejemplo, si estamos ante el mensaje de notificación desde RCA hasta WFQA de una petición de retransmisión, se esperará una confirmación desde éste para generar una célula RM.
- *Gráfico*: representa a la ventana de simulación de la red. Esta clase usa los objetos gráficos que representan a los nodos y a sus enlaces. Esta clase incluye además la siguiente clase:
  - ✓ *DibujoNodo*: incluye todos los elementos necesarios para representar a un nodo de la red con sus propios enlaces.
- *NodoAcTMs*: es la clase más amplia, ya que representa a un nodo activo de la VPN. Contiene las variables locales y los procedimientos necesarios para simular el comportamiento de un nodo activo que implementa TAP. Dentro de esta clase existen las siguientes clases:
  - ✓ *TablaRouting*: representa la tabla de routing del nodo activo empleada para mantener los datos de cada conexión identificada por un VPI/VCI. Esta tabla dispone de métodos para la consulta y el mantenimiento de la tabla.
  - ✓ *MonitorBuffer*: clase empleada en todas las entidades que envían o reciben datos. Representa la estructura de datos buffer de los conmutadores activos. Los buffers almacenan PDU y células ATM genéricas de forma que pueden instanciarse para tratar, tanto células de fuentes no privilegiadas, como PDU pertenecientes a conexiones garantizadas.
  - ✓ *MonitorColas*: clase empleada también por todas las entidades que envían o reciben datos. Representan las estructuras de datos colas de entrada del tráfico a los conmutadores. Pueden almacenar, tanto PDU, como células ATM de forma que puedan instanciarse para tratar, tanto datos en forma de PDU, como en forma de células.
  - ✓ *MonitorDMTE*: clase creada para uso de las entidades que acceden a la memoria dinámica, tanto para leer, como para copiar unidades de PDU. Se emplea por tanto para representar la estructura de datos DMTE en la que se almacenan las PDU de las conexiones con GoS.
  - ✓ *MonitorTablaE/S*: clase pensada para la estructura de datos de las tablas de E/S asociadas a los puertos de salida. Esta clase es también empleada por todas las entidades que envían o reciben datos y representan los enlaces entre unas entidades y otras para el flujo de la información.
  - ✓ *AgenteCoSA***:** Clase que representa al agente de Clase de Servicio.
  - ✓ *AgenteWFQA***:** Clase que representa al agente que aplica el Weighted Fair Queueing.

- ✓ *AgenteCCA*: Clase que representa al agente de Control de Congestión.
- ✓ *AgenteRCA*: Clase que representa al agente de Control de Retransmisiones.
- ✓ *AgenteDPA*: Clase que representa al agente Despachador de las PDU a los puertos de salida.
- ✓ *CélulaBRM*: clase que cuelga de PDU, con los campos específicos de las células backward resource management, que empleamos para la solicitud de retransmisiones.

- *NodoATM*: Clase que representa los nodos estándares ATM que no implementan TAP por lo que no se les considera nodos activos. Como los nodos no activos sólo reenvían el tráfico de datos hacia el siguiente nodo en dirección al destino, no necesitan las estructuras de datos de la clase *NodoAcTMs* ni tampoco las clases de los agentes software. Como se encargan de la conmutación de las células sí disponen de la clase *TablaRouting*. Además, como estos conmutadores reenvían las células BRM que reciben en dirección al nodo emisor también emplean la clase *CelulaBRM* de la clase *NodoAcTMs*.

Cada una de estas clases se implementa como un módulo independiente que contiene la descripción de los procedimientos que componen cada clase. En los casos en que una clase incluya a su vez otras clases, cada una de estas dispone a su vez de sus propios procedimientos que implementan la clase de que se trate.

Un SMA se compone de diferentes clases de agentes, de forma que una vez que se han especificado los aspectos relativos a la comunicación, coordinación y cooperación entre los agentes, lo que queda es disponer de los agentes del SMA. Se ha visto ya que disponemos de clases para la creación de los agentes en el SMA-TAP y también clases específicas para cada uno de los agentes. A continuación realizaremos la especificación de una clase de agente general que permita, mediante instancias, el desarrollo de los diversos agentes del sistema y a la vez nos sirva para explicar la forma en que operan, así como conocer su arquitectura interna y los métodos y atributos que lo forman.

Cada uno de los agentes estará compuesto de las clases relativas a los aspectos de comunicación así como a los de modelo social (coordinación y cooperación). La *Figura 12.4* presenta el aspecto interno de este agente general en el que pueden observarse las clases comentadas interactuando entre sí para conseguir que un agente pueda comunicarse, coordinarse y cooperar con otros agentes del sistema.

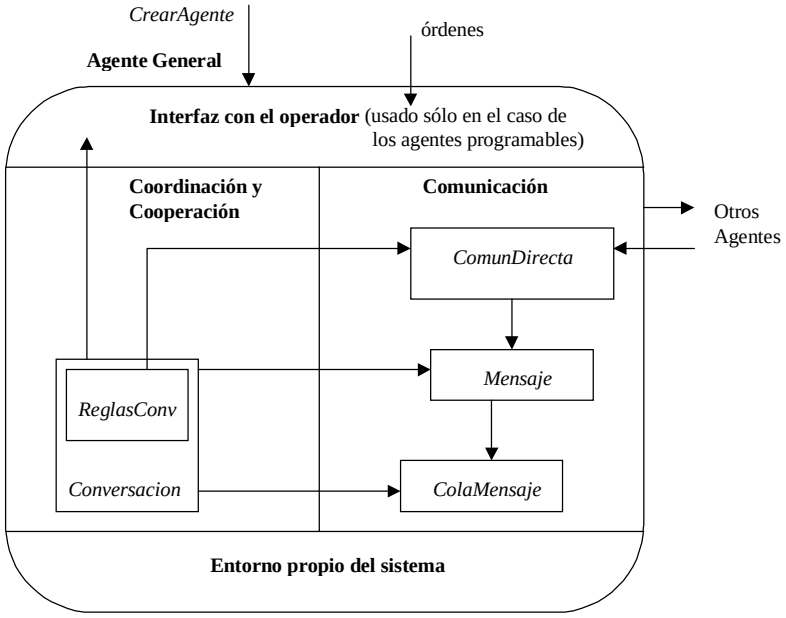

Figura 12.4. *Arquitectura interna de un agente general*

A continuación se presenta la clase *Agente* desde el punto de vista general sin centrarnos en ninguno de los cinco agentes que constituyen el SMA-TAP, pero abarcando todos sus métodos, y sabiendo que los cambios más importantes con que podemos encontrarnos tienen que ver con los agentes con los que se comunica cada agente particular. Así, en la *Figura 12.5* se presentan los atributos y métodos de esta clase *Agente*, que extiende la clase *Threads* de Java, ya que cada agente en el lenguaje Java dispone de su propio hilo de ejecución, lo que permite que varios agentes diferentes puedan ejecutarse concurrentemente aunque hayan sido creados con el mismo espacio de código de programa.

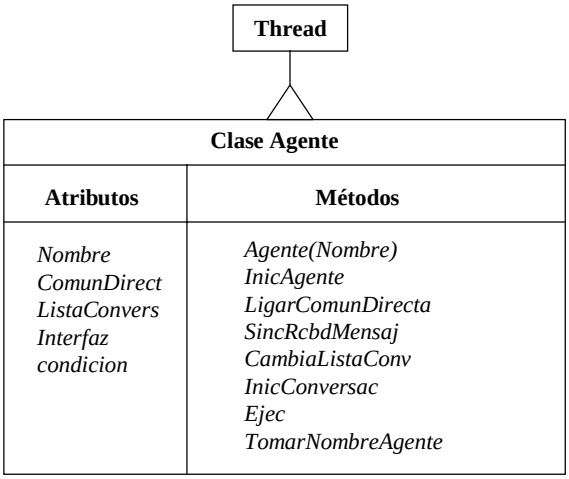

Figura 12.5. *Métodos y atributos de la clase Agente*

Una vez creado, un agente mantendrá su sincronización con los agentes con los que se comunica, por lo que podrá recibir y enviar mensajes a éstos. Pero antes de esto deberá registrar su(s) agentes de comunicación directa para lo que emplea el método *LigarComunDirecta(),* y a continuación se instancia el objeto *ReglaConv* a través del método *InicAgente()* de la clase *Agente*. Cuando esto ha concluido es cuando se inicia la ejecución del agente con el método *Ejec()* y cuando el agente estará listo para enviar y recibir todo tipo de mensajes. Mediante el método *CambListaConv()* se actualiza la lista de conversaciones que cada agente mantiene. Además, se define el método *InicConversac()* para determinar las condiciones bajo las cuáles cada agente va a comenzar una conversación con otro agente (por ejemplo WFQA acepta desde CoSA cuando dispone de colas vacías). La *Figura 12.6* (adaptada de [1]) presenta la secuencia de pasos por los que discurre el ciclo de vida, en este caso del agente CoSA, para describir el funcionamiento de este agente particular aplicando los conceptos comentados en el caso general.

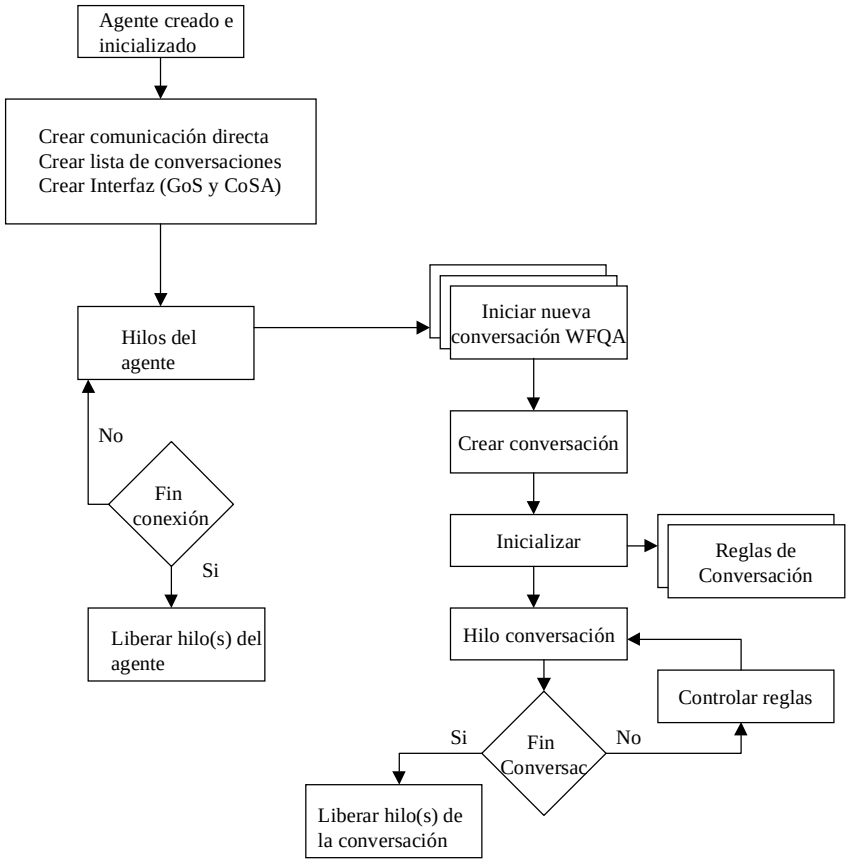

Figura 12.6. *Ciclo de vida de las operaciones del agente CoSA*

12.5. EVALUACIÓN DE RESULTADOS

En los trabajos [11,12] presentamos el buen comportamiento del protocolo RAP (la versión previa del protocolo actual) sobre la arquitectura TAP. Posteriormente implementamos y simulamos la actual versión [13,14] del protocolo confiable incluyendo las técnicas software ya explicadas para introducir las características activas en los conmutadores. Estos mecanismos controlan y gestionan los VCI privilegiados y también ofrecen el mecanismo de recuperación de las PDU congestionadas de los conmutadores activos vecinos.

Como se ha comentado en puntos anteriores se ha implementado en Java un simulador como prototipo de TAP y para evaluar el comportamiento del protocolo. El simulador permite definir la probabilidad de congestión en los emisores, receptores y en los conmutadores ATM. La simulación también permite al usuario introducir valores variables como los parámetros de las fuentes ON/OFF (T_{on} y T_{off}), el número de fuentes emisoras y de receptores del tráfico o el número de conmutadores activos y no activos. El simulador permite al operador de la red la gestión de los agentes programables CoSA y CCA.

Para analizar el comportamiento de la pérdida de células en las simulaciones se usan fuentes de tráfico ON/OFF ya que permiten modelar el tráfico a ráfagas característico del tráfico UBR o ABR para el que TAP es más apropiado. Además, el modelo ON/OFF suele usarse para caracterizar el tráfico ATM por conexiones unidireccionales.

La fuente genera ranuras de tiempo vacías. Se usa en todos los ejemplos una *CSR (Cell Slot Rate*) de 353.208 células/s sobre un modelo de red con enlaces de 155,52 Mbit/s. Cuando la velocidad de llegada de células CAR es menor que CSR, habrá ranuras de tiempo vacías durante el estado activo (ver *Figura 11.12*).

Como se comentó en el *Capítulo 11*, en las simulaciones se emplean valores estadísticos para los tiempos de actividad y de silencio de las fuentes (T_{on} = *0,96* s. y T_{off} = *1,69* s.), aunque también hemos usado otros valores para analizar sus efectos sobre TAP. Se destaca que se han variado algunos de los parámetros para analizar el comportamiento de TAP cuando varía el escenario y los descriptores de las fuentes de tráfico, según se estudia en esta sección.

La *Tabla 12.1* muestra lo descriptores de tráfico con sus valores máximos y mínimos usados en las simulaciones. Utilizamos un proceso que conmuta entre un estado de silencio (*idle*), y el estado activo (*sojourn*) que produce una velocidad media de células (entre 64 Kbits/s y 25 Mbits/s) agrupadas en PDU de 1.500 octetos, aunque el tamaño de las PDU es también un parámetro variable en las simulaciones. Durante los estados ON estos procesos generan células a una velocidad de llegada CAR.

TABLA 12.1
DESCRIPTORES DE FUENTES DE TRÁFICO ON/OFF

Descriptores de tráfico	Parámetros	Mínimo	Máximo
Velocidad Media de las Fuentes	*VMF*	64 kbit./s.	25 Mbit./s.
Cell Arrival Rate	*CAR*	167 cells/s.	65.105 c./s.
Cell inter-Arrival Time	*1/CAR*	6 ms.	15 µs.
Ancho de Banda de Enlaces	*AB*	155,52 Mbps.	622 Mbit/s.
Cell Slot Rate	*CSR*	353.208 cell/s.	1.412.648 cell/s.
Tiempo de servicio por célula	*1/CSR*	2,83 µs.	0,70 µs.
Periodo de tiempo de actividad	T_{on}	0,96 s.	1 s.
Nº medio de células en el estado T_{on}	X_{on}	160 cells	65.105 cells
Periodo de tiempo de inactividad	T_{off}	1,69 s.	2 s.
Nº medio de slots vacíos en el estado T_{off}	X_{off}	596.921 cell slots	2.825.296 cell slots

A continuación se presentan algunos de los resultados más destacables obtenidos en las simulaciones realizadas. Se destaca que todas estas simulaciones son realizadas de forma local sobre un ordenador que simula las topologías de VPN comentadas en el *Capítulo 7*, en el que puede elegirse el número de fuentes, el número de receptores, los conmutadores no activos y los AcTMs que constituyen la VPN. Además puede optarse por los valores de parámetros de tráfico presentados en la *Tabla 12.1* anterior.

Las evaluaciones presentadas se centran en los resultados más significativos y que argumentan las ventajas de TAP como son la recuperación de PDU que se perderían por congestión y además cuidando que todas esas retransmisiones no afecten al rendimiento de la red. Para ello se estudia primero el efecto del CAR sobre el índice de recuperaciones de PDU congestionadas. En segundo lugar se comprueba el efecto del T_{off} sobre el índice de recuperaciones. Por último, se presentan los resultados de las simulaciones aisladas del algoritmo QPWFQ sobre las colas de entrada de los conmutadores activos.

12.5.1. EVALUACIÓN DEL EFECTO DEL CAR SOBRE EL ÍNDICE DE RECUPARACIONES

En este caso, la configuración básica (punto-a-punto) de la VPN estudiada consiste en 3 conmutadores ATM activos con un solo emisor y un receptor.

La *Figura 12.7* muestra el efecto experimentado al variar el CAR entre 86 y 2.667 células por segundo (33.000 bps hasta 1 Mbps respectivamente). En esta simulación se ha fijado la probabilidad de congestión a 10^{-3}. Se usa un buffer de 3.000 octetos y la DMTE almacena 2 PDU de 1.500 bytes para cada conexión.

El valor de PCR es de 64 Kbps (167 cells/s.); T_{on} = 0,96 s.; y T_{off} = 1,69 s, y, sobre el total de las 50 PDU descartadas por congestión, 50 PDU son recuperadas por TAP. También, cuando el CAR=56 Kbps y 33 Kbps, TAP recupera todas las PDU congestionadas. Así, el rendimiento es optimizado (50 PDU recuperadas de 50 PDU congestionadas) porque todas las PDU perdidas son recuperadas y no se producen fallos de DMTE, ya que todas las PDU solicitadas se encuentran en la memoria DMTE del conmutador anterior cuando se solicita la retransmisión.

Como puede observarse en la *Figura 12.7*, cuando la velocidad de llegada de células es baja, el índice de PDU recuperadas es óptima. Cuando el CAR incrementa, 256 Kbps, TAP recupera 48 de las 50 PDU, pero 2 de las PDU perdidas no son solicitadas porque el protocolo detecta insuficiente tiempo de inactividad (T_{off}) para poder realizar la retransmisión y, bajo un escenario en el que se suponen un número de fuentes de *background* que superan la capacidad del enlace. Podemos ver cómo el número de NACK no enviados (PDU no solicitadas) es mayor cuando el valor de CAR incrementa. De este modo, la red no será sobrecargada con retransmisiones inútiles cuando no existe garantía de éxito en la recuperación al no disponer de suficiente tiempo de inactividad agregado.

Se destaca que a elevados valores de CAR (1 Mbps), el número de PDU recuperadas es de 47 y también en este caso, las 3 PDU no recuperadas no han sido solicitadas para evitar los potenciales fallos de DMTE. De este modo, podemos ver que el *goodput* es también optimizado a elevada velocidad, siempre y cuando el número de fuentes con GoS no exceda la capacidad de servicio CSR (ver *Capítulo 11*). Debe destacarse que se han realizado numerosas simulaciones para estudiar este comportamiento y todas han acabado aportando resultados similares, demostrando que TAP es capaz de recuperar un importante número de PDU que experimentan congestiones en los conmutadores y que acabarían perdiéndose en la red sin la existencia de TAP, ya que los mecanismos estándares de la tecnología ATM no se encargan de solventar los problemas relativos a las pérdidas en la propia red, sino que son detectadas en los receptores y recuperadas mediante retransmisión extremo-extremo con el consiguiente efecto sobre el rendimiento de la red, tal como se argumenta en el *Capítulo 8.*

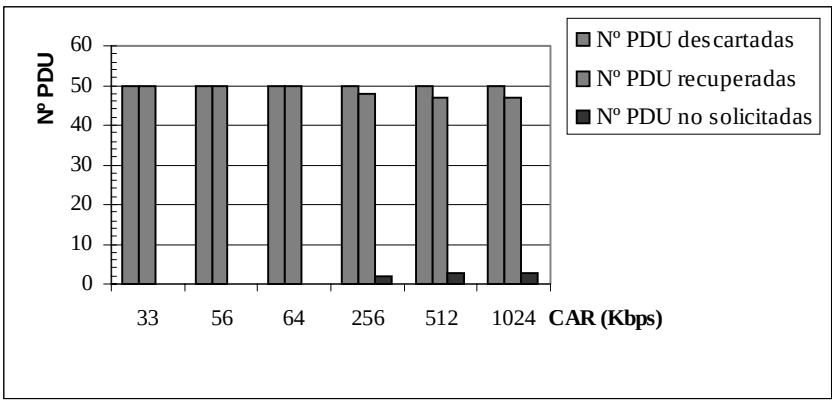

Figura 12.7*. Número de PDU recuperadas en función del CAR*

Se ha estudiado también el índice de recuperaciones sobre otros escenarios consistentes en *1* nodo fuente, *1* conmutador ATM activo, *n* conmutadores no activos y *1* nodo destino. Cuando llega un NACK a un conmutador no activo, éste también transfiere la célula RM al siguiente conmutador. Cuando la RM llega al conmutador activo éste usa la memoria DMTE para retransmitir la PDU solicitada. Este escenario es el mismo que el anterior, sólo que el número de conmutadores no activos varía. En esta configuración hemos simulado el protocolo con varios conmutadores no activos y los resultados obtenidos no cambian respecto a la configuración básica. Sólo varía el retardo en las transmisiones debido a los tiempos de propagación, pero el índice de PDU recuperadas se mantiene como en el caso básico.

Otros escenarios estudiados presentan una configuración punto-multipunto consistente en *1* nodo fuente, *1* conmutador ATM activo, *n* conmutadores no activos y *n* nodos destino. Los resultados obtenidos son similares al escenario básico punto-a-punto, sólo varía el número de nodos destino en las conexiones multipunto, obteniéndose resultados similares a los mostrados en la *Figura 12.7*.

Los resultados de la *Figura 12.7* varían al actuar sobre los diversos parámetros de la simulación como es de esperar. Sin embargo, uno de los aspectos que más afecta al índice de recuperación de pérdidas debidas a congestiones son los valores fijados de T_{on} y T_{off} de forma que, a medida que no se dispone de suficiente tiempo de inactividad agregado, el índice de PDU recuperadas desciende proporcionalmente. Este efecto se estudia en el siguiente apartado.

12.5.2. EFECTO DEL T_{off} SOBRE EL ÍNDICE DE RECUPERACIÓN DE PDU

La *Figura 12.8* muestra los resultados obtenidos al variar el tiempo de inactividad (T_{off}) entre 0,1 y 2 segundos. En este caso se usaron 10 fuentes ON/OFF sobre un enlace con ancho de banda de 25 Mbps. Cada fuente genera 500 PDU con un PCR=1 Mbps. Se fijó el Cell Loss Rate al 5% sobre el número total de las PDU emitidas y, en la figura, se ha mantenido constante el valor de 25 PDU congestionadas. Como puede observarse, cuando el T_{off} agregado es suficiente (0,5 s.), todas las PDU congestionadas son recuperadas (25 recuperaciones de 25 congestiones). Cuando el tiempo T_{off} es menor de 0,5 s. las PDU recuperadas bajan a 12 PDU cuando T_{off}=0,3 s., y a 3 PDU recuperadas cuando T_{off}=0,1 s. De este modo, cuando el tiempo T_{off} es insuficiente, el número de PDU irrecuperables crece, pero TAP garantiza el goodput ya que las PDU irrecuperables no son solicitadas para evitar sobrecargar la red (tal como se ha explicado en el apartado anterior). Por tanto, tal como se comprueba en la *Figura 12.8*, el hecho de no disponer de tiempo de inactividad no afecta negativamente al protocolo ni a la red, ya que en esta situación las PDU no serán solicitadas desde el AcTMs congestionado al conmutador previo. Como ya se ha comentado también en capítulos anteriores, ésta es una de las funciones del agente RCA.

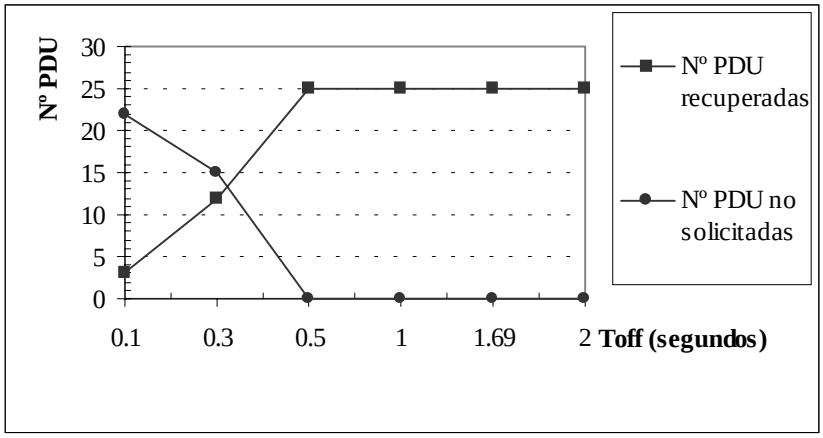

Figura 12. 8. *Efecto de la variación de Toff*

Se ha estudiado también el efecto de la variación del T_{off} sobre otros escenarios donde se alteraron diversos parámetros de simulación como el número de fuentes, el número de destinatarios, el ancho de banda de los enlaces y diversos PCR y nodos AcTMs y, en todas ellas, el efecto de la variación del T_{off} ha terminado dando resultados similares al mostrado en la *Figura 12.8*, demostrándose que la intuitiva idea de atender las retransmisiones aprovechando los tiempos de inactividad de las fuentes y de los enlaces permite aprovechar el goodput en la red.

12.5.3. OTROS ASPECTOS DESTACABLES

Como ya se demostró en el *Capítulo 11*, al incrementar el número de fuentes de background (sin requerimientos de GoS) en las simulaciones se acaba también afectando al índice de recuperación de PDU. Esto es así por el hecho que al aumentar el tráfico en los conmutadores con fuentes de background en realidad lo que se produce es un consumo elevado de la capacidad del enlace y, por tanto, del T_{off} agregado, lo que acaba dando resultados similares a los presentados en la *Figura 12.8*.

Como era de esperar, las simulaciones demuestran también que las variaciones en el tamaño del buffer

afectan al índice de recuperaciones de PDU. Cuanto menor es el tamaño del buffer, mayor es la probabilidad de congestiones en el mismo y, por tanto, incrementa el número de retransmisiones realizadas, aunque el índice de recuperaciones con éxito depende sobre todo de los parámetros estudiados en las *Figuras 12.7* y *12.8*. En el caso de tamaño de PDU de 1.500 octetos, se he comprobado que un tamaño de buffer de 20 Kbytes (427 células) acaba dando resultados aceptables.

El tamaño de PDU usado también afecta a los resultados obtenidos, ya que al emplear PDU de mayor tamaño incrementa también la probabilidad de congestiones en el buffer. Es decir, si se emplea un buffer de 20 Kbytes y PDU de tamaño cercano (o superior a 20 Kbytes) acaba provocando descartes de estas PDU que, como sabemos, pueden tener un tamaño máximo de hasta 65.536 octetos. Por este motivo es importante buscar el punto de equilibrio entre los tamaños de PDU y el tamaño reservado para el buffer de los AcTMs.

Otro aspecto de interés es el efecto del umbral fijado sobre el buffer por el algoritmo EPDR. Aunque el valor del umbral depende directamente también del tamaño del buffer y de las PDU, en la mayor parte de situaciones se obtiene un buen comportamiento cuando el buffer se sitúa en entre el 95% y el 97% del tamaño total del buffer. Actuar sobre este valor permite amortiguar el efecto de la fragmentación de las PDU, pero también aumenta el número de retransmisiones realizadas. Recordamos que la función de ajuste del valor del umbral es responsabilidad del agente programable CCA.

Como se ha destacado en el *Capítulo 10*, otro aspecto que merece comentario es el tamaño de la DMTE empleada. En este caso, como en el del buffer, es necesario buscar un adecuado punto de equilibrio entre el tamaño de la memoria dinámica y el de las PDU. Intuitivamente se ve que el tamaño de la DMTE debe ser mayor cuanto mayor es el tamaño de las PDU. Además, el tamaño de la DMTE también depende del parámetro GoS que se defina para cada fuente; es decir cuando se define una fuente con GoS con valor a 2, esto quiere decir que se almacenan 2 PDU en la DMTE para esa fuente. A medida que se incrementa el parámetro GoS, mayor es la garantía de servicio aportada a las fuentes; sin embargo, mayor es la necesidad de memora DMTE. La misma discusión puede hacerse con respecto al número de fuentes privilegiadas que se definen; es decir, cuantas más fuentes con GoS se empleen, mayor será el requerimiento de memoria. Por tanto, para que el índice de recuperaciones de la DMTE sea aceptable es necesario encontrar una situación de equilibrio entre el tamaño total de DMTE, el tamaño de cada una de las PDU, el parámetro de GoS de cada fuente, y el número de fuentes privilegiadas. Ajustando adecuadamente los valores de todos estos parámetros se logran también aceptables índices de recuperación de PDU.

En resumen, con las simulaciones realizadas se ha comprobado que la arquitectura TAP distribuida y activa aprovecha las ventajas de los conmutadores AcTMs. Así, se ha verificado que es posible recuperar un importante número de PDU con un aceptable tamaño de memoria DMTE y una razonable complejidad añadida en los conmutadores activos soportada por agentes software. Estas simulaciones demuestran también que la idea intuitiva de aprovechar los tiempos de silencio en las fuentes ON/OFF es cierta consiguiendo mejorar el comportamiento y QoS en las redes ATM que soportan fuentes que requieren GoS.

12.5.4. EFECTOS DEL ALGORITMO QPWFQ SOBRE COLAS DE ENTRADA Y BUFFER

En este apartado se presentan los detalles de funcionamiento del algoritmo QPWFQ, implementado como parte del agente WFQ del SMA-TAP. Este algoritmo es explicado en los *Capítulos 4* y *10* y, a continuación, se presenta su entorno de ejecución en el simulador y se comentan algunos de los resultados más significativos que se han obtenido. Se destaca que en este caso vamos a estudiar el comportamiento del algoritmo de forma aislada; es decir, no interviene ningún otro elemento del SMA. El escenario de simulación, por tanto, está formado por tres conmutadores activos y a cada uno de ellos llegan cuatro VPI/VCI, de forma que las cuatro entradas del conmutador A se multiplexan en una de las cuatro entradas del conmutador B. A su vez las cuatro entradas del conmutador B se multiplexan a su salida que es una de las entradas del conmutador C, que dispone también de cuatro entradas y una sola salida. De este modo, se pueden elegir varias fuentes privilegiadas y a su vez simular el efecto de la multiplexación y el de las fuentes de *background*.

Cada uno de los tres AcTMs dispone de sus cuatro colas de entrada, de su correspondiente buffer y de su propia DMTE. Nuestra intención es estudiar las congestiones, el número de retransmisiones y su efecto sobre el rendimiento del cada conmutador. El escenario de simulación puede observarse en la *Figura 12.9*, donde se muestra el aspecto del conmutador A, con sus cuatro fuentes y colas de entrada, con una asignación de pesos en las colas, el buffer y la salida multiplexada hacia el VPI 5.

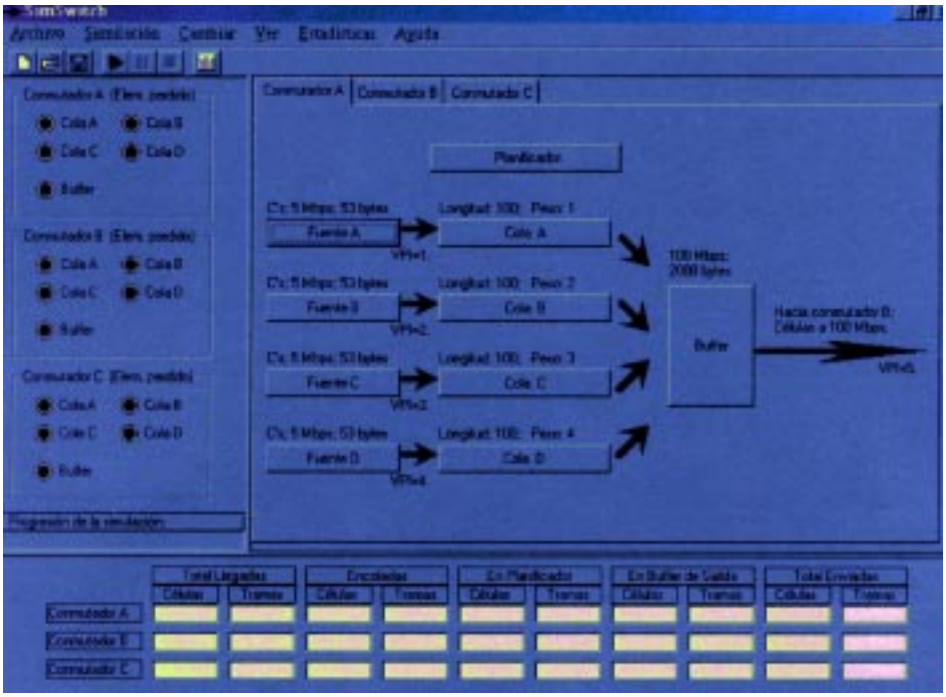

Figura 12.9. *Entorno de simulación del algoritmo QPWFQ sobre un AcTMs*

A continuación se comentan brevemente cada uno de los apartados del entorno de la *Figura 12.9*.

- La *barra de menús*, situada en la parte superior de la *Figura 12.9*, aporta seis opciones que permiten actuar sobre el simulador (*Archivo*, *Simulación*, *Cambiar*, *Ver*, *Estadísticas* y *Ayuda*).
- La *barra de botones*, situada bajo la barra de menús, se compone de seis botones cada uno programado con diversas acciones sobre las simulaciones (*Nuevo*, *Abrir configuración*, *Guardar configuración*, *Iniciar simulación*, *Congelar simulación*, *Detener simulación* y *Estadísticas de simulación*).
- *Zona de descripción* de los conmutadores del escenario, que se muestra en la parte central de la *Figura 12.9*. Pueden observarse tres fichas, donde cada una corresponde a cada uno de los conmutadores con sus diferentes componentes que son los siguientes:
  - ✓ *Fuentes* dispuestas para la generación del tráfico que llega a cada conmutador. Cada fuente tiene los siguientes parámetros: *Tipo* de fuente con GoS o no (generan PDU o células); *CAR* del tráfico que puede ser desde 1 a 100 Mbps; *Longitud* de las PDU (desde 61 a 65.535 bytes); y *Estado* de la fuente (activa o inactiva).
  - ✓ *Colas* que representan la llegada de las transferencias desde las fuentes. Tiene como parámetros modificables el *peso* y la *longitud* de la cola.
  - ✓ *Buffer* que representa al buffer de los AcTMs sobre los que confluye el trafico de las colas. Tiene como parámetros modificables el *tamaño* (comprendido entre 100 y 100.000 bytes); la *velocidad* de conmutación (de 1 a 100 Mbps); y la velocidad de salida de las unidades procesadas o capacidad de servicio (de 1 a 100 Mbps).
  - ✓ El *planificador* representa a las cola de turnos del algoritmo.
- *Zona de información de estado del flujo*, situada en la parte inferior de la *Figura 12.9*, presenta, en tiempo de simulación, la información de las unidades de transferencia (tramas o células) procesadas en cada uno de los conmutadores.
- *Zona de progresión de la simulación*, situada en la parte inferior izquierda de la *Figura 12.9*, donde puede seguirse la evolución de las PDU y células a lo largo de las colas o buffers en cada conmutador, informando visualmente cuando se producen pérdidas de PDU por congestión del buffer.

En cuanto a las estadísticas se han implementado tres posibilidades:

- *Peticiones de retransmisión* para obtener una representación gráfica del número de solicitudes de

retransmisión que ha realizado cada uno de los AcTMs hacia el conmutador previo, en función del tiempo de simulación.

- *Retransmisiones atendidas* para obtener una representación gráfica del número de PDU retransmitidas por cada conmutador hacia el conmutador siguiente del escenario y en función del tiempo de simulación.
- *Índice de Productividad* de cada conmutador, que muestra el número de células por crédito (intervalo de tiempo en el que se procesa un tráfico proporcional al tráfico de entrada) que cada conmutador envía a su puerto de salida sobre el que multiplexa todas las entradas y expresado en función del tiempo. Esta estadística muestra, por tanto, el goodput de cada conmutador teniendo en cuenta, tanto las transmisiones, como las retransmisiones procesadas.

En este caso se simulan las congestiones de forma realista, es decir, sólo se realiza una retransmisión de PDU cuando se pierde, y una PDU se puede perder en los dos casos siguientes:

- Se intenta introducir en una cola una PDU cuando la cola está llena en ese momento. Las colas están llenas cuando la velocidad de conmutación en el buffer es menor que la velocidad total de llegada de tráfico a éste.
- Se intenta introducir en el buffer una PDU que no cabe.

Seguidamente se muestran los resultados de algunas de las simulaciones realizadas.

La *Figura 12.10* presenta la productividad de los conmutadores en un escenario en el que no se ha producido ningún tipo de congestión. En este caso los parámetros de la simulación han sido los siguientes: Todas las fuentes generan en CAR de 5 Mbps, los pesos de cada cola se corresponden con el CAR de su correspondiente fuente. En cada cola tienen cabida 100 unidades de transferencia (PDU o células), y los tres buffers, con una capacidad de 2.000 octetos, conmutan a 100 Mbps. Se realizó una simulación de 8,56 ms., de forma que el conmutador A fue capaz de procesar 400 células, el conmutador B 415 células y el C 448 células. Con estos parámetros de tráfico no se acaba produciendo ninguna congestión en los buffers y, como puede observarse en la *Figura 12.10*, el volumen de tráfico es constante a lo largo de todo el tiempo de simulación, salvo al final. Este efecto se produce por el corte de tráfico desde el conmutador A que provoca al final de la simulación una bajada en el tráfico servido a los conmutadores B y C. En este caso las estadísticas de retransmisiones mostrarían gráficas completamente planas al no haber congestiones.

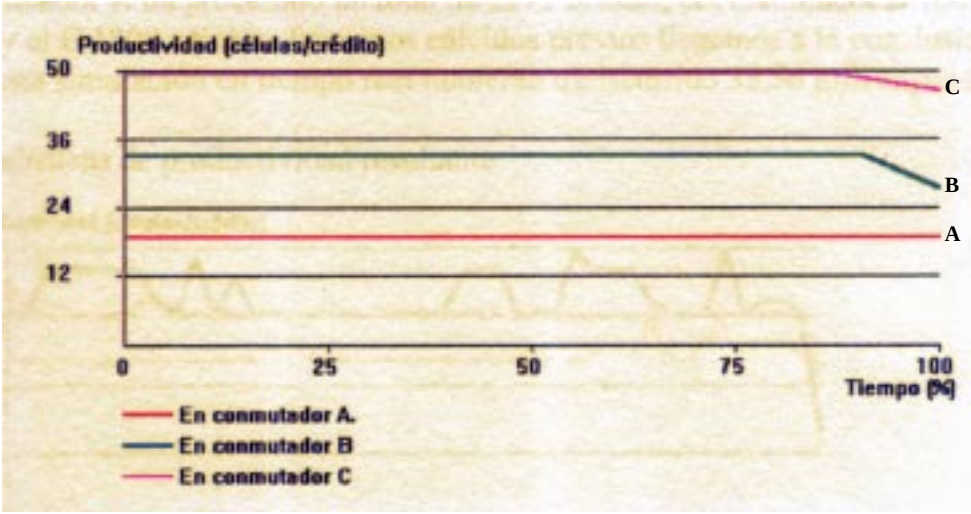

Figura 12.10. *Productividad en un escenario sin congestiones*

La *Figura 12.11* muestra el resultado de una simulación con congestiones en el buffer de los conmutadores. Los parámetros del escenario de esta simulación se explican a continuación.

- Conmutador A: se han introducido tres fuentes de *background* de 5 Mbps cada una, y una fuente con GoS que genera PDU de 530 bytes a 10 Mbps. Las colas tienen una capacidad de almacenamiento de 100 unidades, y pesos que se corresponden con la velocidad de las fuentes. El buffer tiene una capacidad de 20 Kbytes, velocidad de conmutación de 100 Mbps y la velocidad de salida es de 30 Mbps. Con estos datos no se prevén congestiones en el conmutador.

- Conmutador B: con tres fuentes de *brackground* idénticas al conmutador A, lo mismo que las colas. Sin embargo, el buffer es de 15 Kbytes con velocidad de conmutación de 100 Mbps y velocidad de salida o capacidad de servicio de 35 Mbps. En este caso el buffer se llenará progresivamente.
- Conmutador C: las fuentes de *background* y las colas son idénticas a los conmutadores A y B. Pero en este caso el buffer es de 2 Kbytes con una velocidad de conmutación de 100 Mbps y capacidad de servicio de 35 Mbps. Debido a la escasa capacidad del buffer se prevé el llenado progresivo del buffer y una elevada probabilidad de congestiones debido al escaso tamaño del buffer.

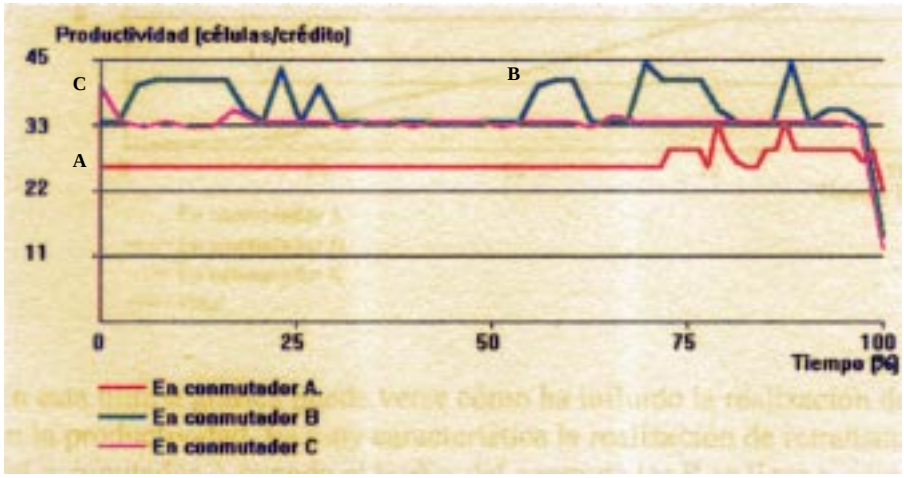

Figura 12.11. *Índice de productividad con buffer congestionado*

En este escenario de simulación el conmutador A procesa un total de 2.273 células, el B 1.645 células y el C 1.300 células, todo ello durante 38,56 ms. Puede observarse un tráfico casi constante en el conmutador C, mientras en el conmutador B aparecen importantes fluctuaciones de productividad, que serán justificadas en gráficas siguientes y que son debidas a las retransmisiones solicitadas desde el conmutador C. En el caso del conmutador A al principio no se produce variación; sin embargo, al final de la simulación el flujo constante experimenta picos debidos a las solicitudes de retransmisión recibidas desde el conmutador B.

La gráfica de la *Figura 12.12* presenta la estadística de petición de retransmisiones realizadas por cada uno de los conmutadores activos. Como puede observarse, en el caso del conmutador C las retransmisiones se producen casi desde el inicio de la simulación, debido al pequeño tamaño de buffer que provoca la pérdida de las PDU. Sin embargo, el conmutador B no realiza retransmisiones hasta el 75% del tiempo de simulación que es cuando comienza a experimentar las congestiones debidas a la sobrecarga de tráfico provocada por las solicitudes de retransmisión recibidas desde el conmutador C. Podemos comprobar que en la simulación se han perdido cinco PDU (se solicitan 22 retransmisiones y se retransmiten 17 PDU). Puede verse que las retransmisiones que se pierden pertenecen al conmutador B que sólo sirve 10 PDU de las 15 que le solicita el conmutador C. El conmutador A no realiza ninguna petición de retransmisión, y si fuese así, esto sería indicativo de pérdidas, ya que el conmutador A no tiene por encima ningún otro conmutador.

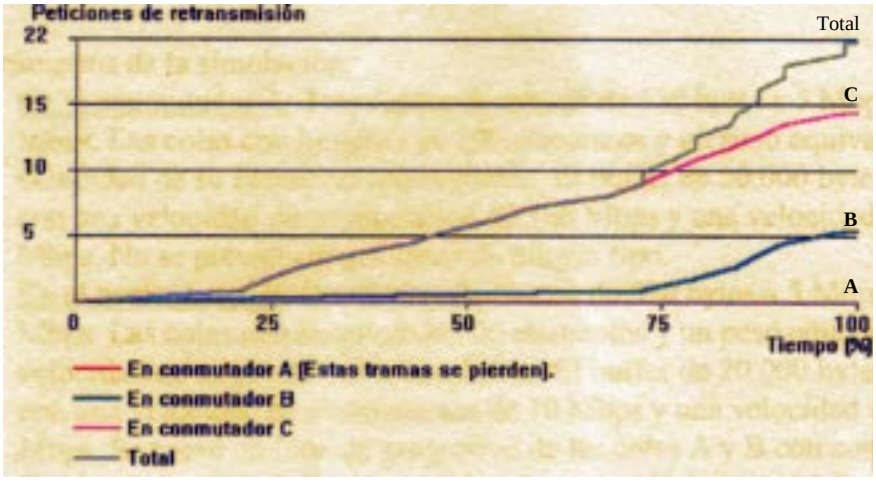

Figura 12.12. *Estadística de solicitudes de retransmisión*

A continuación se estudian las estadísticas relativas a las retransmisiones atendidas por cada uno de los conmutadores. La *Figura 12.13* nos ayuda a entender cómo influyen la atención de las retransmisiones en la productividad representada en la *Figura 12.11*. Es muy significativo el hecho de que en la *Figura 12.13* el conmutador A comienza a atender solicitudes de retransmisión que provienen del conmutador B cuando su buffer se congestiona (debido a las solicitudes de retransmisión provenientes del conmutador C). A medida que se empiezan a atender las retransmisiones en el conmutador A en la *Figura 12.13*, se comienzan a producir las fluctuaciones de productividad en el conmutador A que se observaron en la *Figura 12.11*. En la *Figura 12.13* puede observarse cómo el conmutador C no atiende ninguna solicitud de retransmisión por ser el último de la red. Hemos de destacar que este efecto de sobrecarga sobre los conmutadores que realizan las retransmisiones se produce porque estas simulaciones del algoritmo QPWFQ se han realizado para estudiar su comportamiento aislado del resto de mecanismos de TAP, y por esto no actúa el control de tiempos de inactividad antes de responder a las retransmisiones.

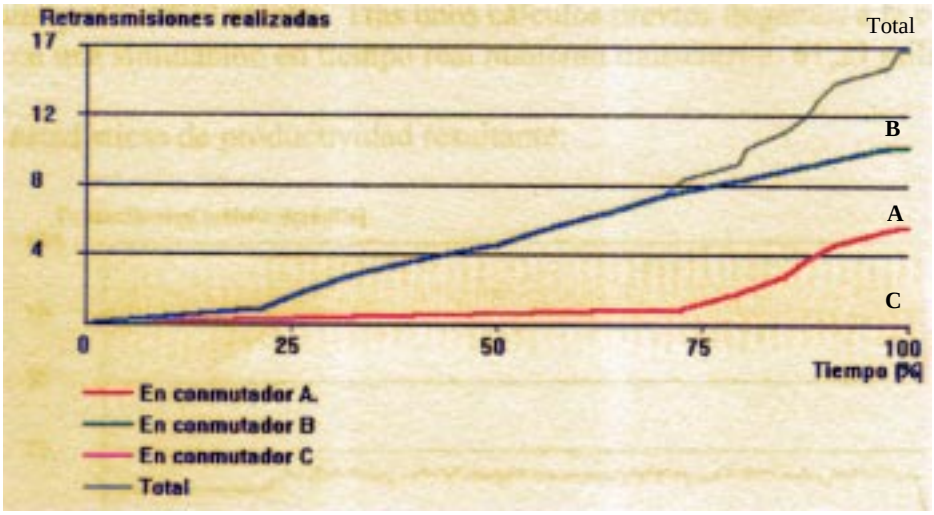

Figura 12.13. *Estadísticas de retransmisiones servidas*

A continuación se presenta un último escenario de simulación donde se estudian las congestiones sobre las colas de entrada. En este caso los parámetros de la simulación son los siguientes:

- Conmutador A: Se emplean tres fuentes con GoS de 530 bytes y un CAR de 5 Mbps. La cuarta fuente es de background a 50 Mbps. Las colas de entrada son de 100 elementos y un peso equivalente a la velocidad de sus correspondientes fuentes. El buffer es de 20 Kbytes de capacidad y una velocidad de conmutación de 100 Mbps y un capacidad de servicio de 100 Mbps. En este caso no se prevén congestiones.
- Conmutador B: Se introducen dos fuentes privilegiadas de 530 octetos y CAR de 5 Mbps. Una tercera fuente es de background con un CAR de 50 Mbps. Las colas son idénticas a las del conmutador A. El buffer es de 20 Kbytes de capacidad, con una velocidad de conmutación de 10 Mbps y una capacidad de servicio de 100 Mbps. En este caso se intuye un llenado progresivo de las colas A y B con congestión que corresponden a las dos fuentes privilegiadas.
- Conmutador C: Se usan dos fuentes privilegiadas de 530 bytes a 5 Mbps, y una tercera de relleno a 10 Mbps. Las colas son iguales a los conmutadores A y B. El buffer es de 2 Kbytes de capacidad y una velocidad de 10 Mbps, con una capacidad de servicio de 100 Mbps. Se prevé un llenado progresivo de las colas A y B y congestiones probables en estas colas debido a la baja velocidad de conmutación del buffer.

Se ha simulado este escenario durante 61,23 milisegundos, de forma que el conmutador A procesó 9.387 células, el conmutador B 1.747 células y el conmutador C 5.570 células.

La *Figura 12.14* presenta la gráfica de productividad de los conmutadores, observándose cómo en el conmutador A queda caracterizado el tráfico a ráfagas propio de la transmisión de las PDU al existir tres fuentes privilegiadas. También se observa que los conmutadores B y C transmiten muy poca cantidad de tráfico debido a la baja velocidad de conmutación que tienen. Puede observarse cómo el volumen de tráfico general se incrementa al llegar al 25 % del tiempo de simulación, debido al inicio de congestiones y a la aparición de retransmisiones como podrá comprobarse en las dos siguientes figuras.

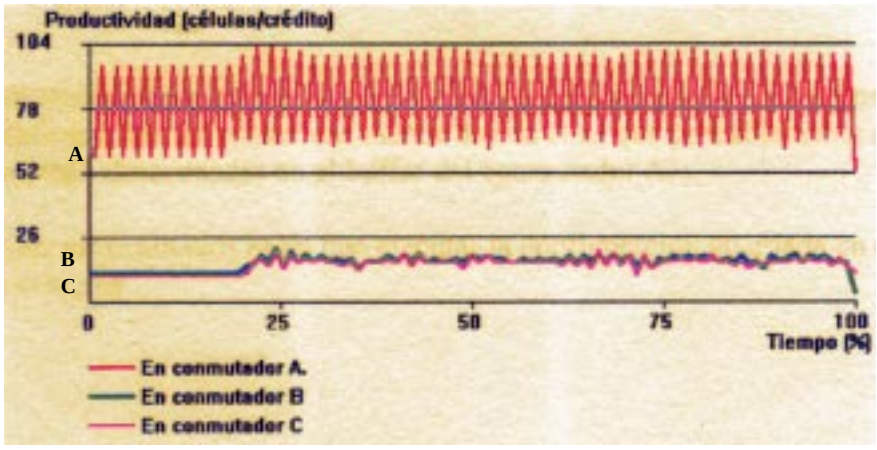

Figura 12.14. *Estadística de productividad en congestiones sobre las colas*

La *Figura 12.15* presenta la estadística relativa a las solicitudes de retransmisión realizadas por los tres conmutadores. Puede observarse cómo sólo el conmutador B realiza solicitudes de retransmisión a partir del 25 % del tiempo de simulación debido a la congestión experimentada en sus colas de entrada y motivada por la baja velocidad de conmutación del buffer y a la elevada fuente de *background* de 50 Mbps introducida.

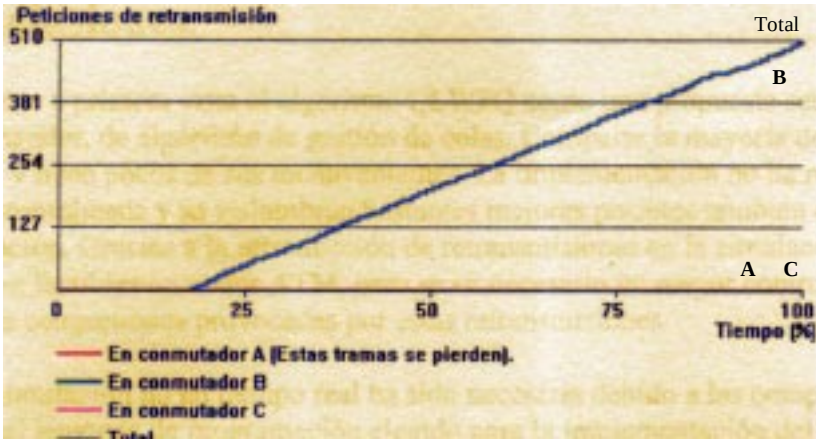

Figura 12.15. *Peticiones de retransmisión realizadas por los conmutadores*

Por último, se estudia en este tercer escenario la estadística de retransmisiones realizadas por los conmutadores. En la *Figura 12.16* puede comprobarse cómo las retransmisiones realizadas siguen el mismo ritmo que las peticiones de retransmisión representadas en la *Figura 12.15*. En la *Figura 12.15* es el conmutador B el que solicita las retransmisiones, mientras en la *Figura 12.16* se observa cómo es el conmutador A el que se encarga de realizar las retransmisiones. Comparando las dos figuras podemos comprobar que se han realizado 510 peticiones de retransmisión desde B, mientras han sido retransmitidas 508 de ellas desde A, debido al corte de la simulación desde el conmutador A.

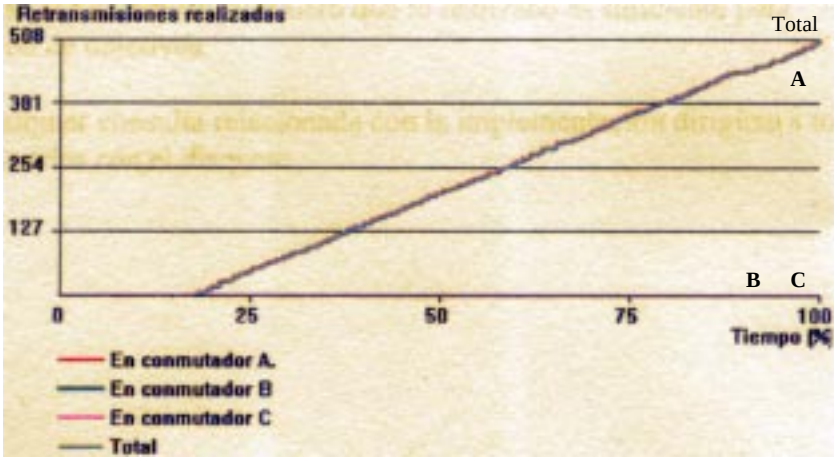

Figura 12.16. *Número de retransmisiones realizadas*

12.6. CONCLUSIONES

El lenguaje Java y su JVM ofrecen atractivas ventajas para el desarrollo de protocolos de comunicaciones portables y extensibles. Además, las propias características del lenguaje permiten también la incorporación en esos protocolos de las premisas necesarias para el desarrollo de sistemas multiagente, ya que la propia extensibilidad aportada por Java permite desarrollar agentes programables. El propio lenguaje Java ha sido identificado como una de las herramientas más adecuadas para implementar agentes, sus interacciones y comunicación con otros agentes en los SMA. El relativo corto periodo de vida del lenguaje no permite tener datos acertados sobre su propia evolución en cuanto al rendimiento aportado por Java con respecto a otros lenguajes en el desarrollo de protocolos; sin embargo, la mejora de sus nuevas versiones permite adelantar que esto beneficiará al rendimiento a medida que evolucionen el propio compilador de Java y la JVM.

Aunque la implementación del protocolo SMA-TAP no es el principal objetivo de esta tesis, hemos presentado la especificación, las decisiones de diseño y algunos de los detalles de implementación del prototipo de simulador de TAP, desarrollado en lenguaje Java con JDK 1.1 y que nos ha servido para evaluar algunos de los aspectos de rendimiento y comportamiento de TAP, a la vez que ha sido la excusa para investigar y experimentar sobre la ingeniería de protocolos de comunicaciones basados en agentes software. Los experimentos realizados han permitido también identificar interesantes vías de investigación futuras para el enriquecimiento e implementación de un simulador más avanzado en el que el SMA sea concluido e incorpore las posibilidades de CORBA y RMI con la intención de poder simular el comportamiento de TAP en una red real sobre la que realizar el estudio de comportamiento del protocolo con tráfico real.

El simulador de TAP se ejecuta sobre un ordenador aislado donde se pueden elegir determinados escenarios con el objetivo de conocer el comportamiento de TAP en un entorno local. De este modo se ha estudiado el comportamiento del algoritmo QPWFQ sobre las colas de entrada en los conmutadores, y se ha comprobado que TAP recupera un importante número de PDU que de otro modo se perderían en situaciones de congestión. También se ha comprobado que se satisface la idea intuitiva de aprovechar los periodos de inactividad de las fuentes y de los enlaces para atender las retransmisiones en los conmutadores activos. Se ha analizado además el comportamiento del algoritmo QPWFQ de forma aislada, comprobándose que sólo con este algoritmo y la DMTE se puede conseguir recuperar un importante número de PDU congestionadas.

REFERENCIAS

[1] Deepika Chauhan, JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation," *ECECS Deparment Thesis, U. Cincinnati,* (1997) *http://www.ececs.uc.edu/~abaker/JAFMAS/*

[2] Sun Microsystems, "Java Whitepaper," *http://java.sun.com/doc/overview/java/index.htm*, (1996).

[3] L. Cardelli, "Obliq: A Language with Distributed Scope", *Technical Report, Digital Equipment Corporation, Systems Research Center*, May. 1995.

[4] General Magic, "Telescript Language Reference", *General Magic*, Oct. 1995.

[5] B. Krupczak, K. L. Calvert, and M. H. Ammar, "Implementing communications protocols in Java," *IEEE Communications Magazine*, pp. 93-99, (1998).

[6] B. Krupczak, K. L. Calvert, and M. Ammar, "Implementing protocols in Java: The price of portability," *Procs. IEEE INFOCOM*, (1998).

[7] J. Gosling, B. Joy, and G. Steele, "The Java languaje specification, v. 1.0," *Sun Microsystems*, (1996).

[8] Sun Microsystems, "The Java Virtual Machine specification, v. 1.0," *Technical Report* (1995).

[9] Jamie Jaworski, "Java 1.2 Al descubierto," *Ed. Prentice Hall* (1999).

[10] H. Jeon, C. Petrie and M. R. Cutkosky, "JATLite: A Java Agent Infraestructure with Message Routing," *IEEE Internet Computing,* (2000). *http://www-cdr.stanford.edu/ProcessLink/papers/JATL.html*

[11] José Luis González-Sánchez and Jordi Domingo-Pascual, "RAP: Protocol for Reliable Transfers in ATM Networks with Active Switches," *Technical Report UPC-DAC-1999-54. http://www.ac.upc.es/recerca/reports/INDEX1999DAC.html* (1.999).

[12] José Luis González-Sánchez and Jordi Domingo-Pascual, "RAP: Protocol for Reliable Transfers in ATM Networks with Active Switches," *International Conference on Communications in Computing (CIC'2000).*

[13] José Luis González-Sánchez and Jordi Domingo-Pascual, "TAP: Architecture for Trusted Transfers in ATM Networks with Active Switches," *ATM'2000 IEEE Conference on High Performance Switching and Routing Joint IEEE ATM Workshop 2000 and 3rd International Conference on ATM (ICATM'2000),* (2000).

[14] José Luis González-Sánchez and Jordi Domingo-Pascual, "Trusted and Active Protocol over a Distributed Architecture in ATM Networks with agents," *IEEE International Conference on Computer Communication and Networks (IEEE IC3N'2000),* (2000).