Filtrado de Respuestas Parciales en Arquitecturas de Recuperación de Información Distribuidas

Juan Francisco Puentes Calvo

Tecnología de la Información y las Comunicaciones
Universidad da Coruña

FILTRADO DE RESPUESTAS PARCIALES EN ARQUITECTURAS DE RECUPERACIÓN DE INFORMACIÓN DISTRIBUIDAS

POR

JUAN FRANCISCO PUENTES CALVO

DIRECTOR DE TESIS:
VICTOR CARNEIRO

TESIS PROPUESTA PARA EL GRADO DE DOCTOR EN

TECNOLOGÍA DE DE LA INFORMACIÓN Y LAS COMUNICACIONES

UNIVERSIDAD DE A CORUÑA

2006

FILTRADO DE RESPUESTAS PARCIALES EN ARQUITECTURAS DE RECUPERACIÓN DE INFORMACIÓN DISTRIBUIDAS

Resumen: El gran incremento del volumen de información disponible en línea desde hace unas décadas hace necesarias, cada vez más, técnicas de recuperación de información con el objetivo de gestionar, recuperar y filtrar la información disponible por estos medios. Las líneas de investigación actuales han descubierto la existencia de un cuello de botella en el canal de respuesta de las arquitecturas de recuperación de información distribuida, debido, principalmente, al gran número de respuestas parciales y su consiguiente influencia sobre las arquitecturas de comunicaciones y sobre los *brokers* de usuario. Por un lado, si disminuimos el número de respuestas parciales generados por cada servidor de consulta, disminuimos la precisión de la respuesta final; por el otro lado, si diseñamos una arquitectura de comunicaciones que soporte el tráfico generado obtenemos un claro cuello de botella al ordenar los resultados parciales en los *brokers* de usuario con el fin de seleccionar la respuesta final.

La solución que propone este trabajo consiste en una arquitectura filtrada de comunicaciones implementada mediante nodos programables, los cuales forman una subred que posee la capacidad de procesar de forma transparente el tráfico que la atraviesa, con el objetivo de reducir la cardinalidad de las respuestas parciales. Así pues la principal ventaja de usar nodos programables en vez de, por ejemplo, brokers es la transparencia; lo que permite construir infraestructuras de canales de respuesta altamente flexibles y fácilmente modificables en tiempo de ejecución.

Este trabajo analiza, diseña, realiza, mide y discute un conjunto de algoritmos de filtrado de respuestas parciales (aquellos tradicionalmente usados en ámbitos de recuperación de información y la familia de algoritmos BGFAST – *Buffered and Grooved Filter with Adaptative and Scaled Threshold*), además de una arquitectura de nodo programable (VAIN – *Value Added Independent Node*), los cuales conjuntamente permiten reducir de forma previsible y sensiblemente la cardinalidad de las respuestas parciales, alcanzando mejoras, según nuestros cálculos, del 64% respecto a un canal de respuesta centralizado y no filtrado.

Agradecimientos

"*With a little help from my friends*"

Lennon / McCartney

Esta tesis no sería una realidad si no fuese por el apoyo y ayuda de mucha gente:

A nivel personal quiero darle las gracias a Celia, la cual ahora conoce todos los sufrimientos y alegrías que un trabajo de esta categoría puede acarrear; le agradezco sobre todo su ánimo en los malos momentos y su sincera alegría en los buenos, los cuales han formado, sin lugar a dudas, el apoyo emocional básico necesario y siempre suficiente. A mi hermano por estar ahí, creyendo y animando en los peores momentos. A mi madre, fallecida mucho antes de iniciar este trabajo, por enseñarme las mejores lecciones sobre la vida y darme el ejemplo de cómo vivirla, hasta el final. A mi padre, fallecido durante la realización de este tesis, y con quien me hubiera gustado compartir sus resultados. Finalmente, y en general, a mi familia, tanto carnal como política, y a mis amigos, por su ilusión que superaba incluso, a veces con creces, la mía propia.

A nivel profesional he de reconocer, en primer lugar, el trabajo de mi director de tesis, Victor, cuyas indicaciones y guías fueron de gran utilidad durante la realización de este trabajo, restándole tiempo al suyo. En segundo lugar, pero no menos importante, me gustaría reconocer el esfuerzo y la paciencia de Fidel, sin los cuales, sin duda alguna, este trabajo no sería el mismo.

Tabla de contenidos

ILUSTRACIONES

Tablas

1 Introducción

En los últimos años hemos asistido a un incremento espectacular en el volumen de información disponible en línea a través de, especialmente, Internet. Hoy en día este gran conjunto de datos sigue incrementándose, haciéndose cada vez más necesario técnicas de recuperación de información (IR) con el objetivo de gestionar, recuperar y filtrar la información disponible por estos medios.

Existen diferentes estrategias para implantar un sistema IR, dos de las más utilizadas son una aproximación basada en índices invertidos locales (*local inverted files*) y plataformas de recuperación distribuidas (DIR) (*Distributed Information Retrieval*) [Hawking, 99] sobre todo en ámbitos de gran carga de consultas o usuarios. Estas estrategias consisten en la distribución sobre un conjunto de servidores de consultas (QS) del conjunto total del índice invertido en subconjuntos disjuntos, de forma que cada QS es responsable de procesar la consulta sobre su subconjunto. Como resultado de dicha consulta, cada QS emite una serie de pares (*identificador*, *peso*) indicando el documento que cumple la pregunta y el peso relativo de este en el contexto de la respuesta. Esta es procesada por uno o más agentes intermediarios (*brokers*) para seleccionar los mejores documentos (teniendo en cuenta sólo sus pesos) y generar así la respuesta definitiva.

Los trabajos de Cacheda *et al.* [Cacheda, 04] [Cacheda, 05] han identificado dos principales cuellos de botella en estas arquitecturas: los *brokers* (dado el alto número de resultados parciales que deben ser ordenados) y la misma subred de interconexión (en un sistema DIR, dado el alto número de QS y el continuo intercambio de datos con los *brokers*).

El modelo analítico que determina el proceso de consulta de un sistema DIR [Cacheda, 05] muestra que el tiempo total necesario para procesar la consulta *i* es:

$$t_i = \max(t_{i,j}) + \max(ra_{i,j}) + tc(\sum_j tr_{i,j}).$$

Donde $t_{i,j}$ es el tiempo que invierte el QS j para procesar la consulta i ; $ra_{i,j}$ es el tiempo empleado para enviar la respuesta parcial de la consulta i desde el QS j; $tr_{i,j}$ es el número de respuestas parciales producidas por el QS j para la consulta i; finalmente $tc(n)$ es el tiempo empleado por los *brokers* para combinar y ordenar n elementos.

1.1 Objetivos de la tesis

El principal objetivo de esta tesis es minimizar el valor de t_i anteriormente descrito, mediante la reducción de los cuellos de botella representados por los términos $ra_{i,j}$, $tr_{i,j}$ y $tc(n)$.

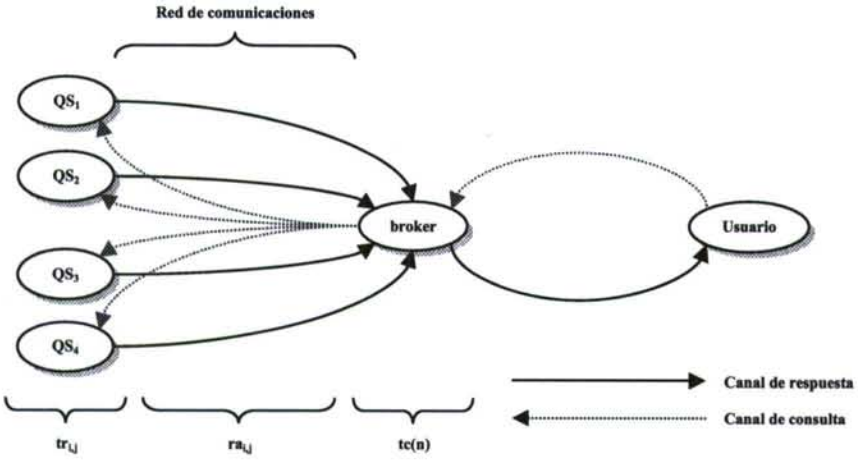

Ilustración 1: Canales de consulta y respuesta en una arquitectura DIR

En la Ilustración 1 podemos observar un modelo orientado a las comunicaciones de una arquitectura DIR. En ella establecemos dos canales: uno de consulta, por donde fluyen las preguntas de los usuarios al sistema, y otro de respuesta, en donde los QS envían sus respuestas parciales hacia un *broker* (de usuario) encargado de ordenar los documentos seleccionados y disminuir la cardinalidad de la respuesta en aras de obtener las T mejores referencias a la consulta realizada. Nuestro esfuerzo de investigación se centrará en el canal de respuesta, disminuyendo el término $tc(\Sigma tr_{i,j})$ mediante la reducción del parámetro $tr_{i,j}$ y, consecuentemente, incrementando el ancho de banda disponible para el canal de respuesta ($ra_{i,j}$) de los QS.

La solución propuesta (Ilustración 2) consiste en dotar a la red de comunicación, entre los QS y el *broker* de usuario, de la capacidad de procesamiento necesaria como para eliminar

lo más tempranamente posible documentos dentro de las respuestas parciales, que sabemos que no formarán parte de la respuesta final de la consulta. Con ello pretendemos reducir el tráfico entre los actores del sistema y disminuir el número de elementos que el *broker* de usuario necesita ordenar.

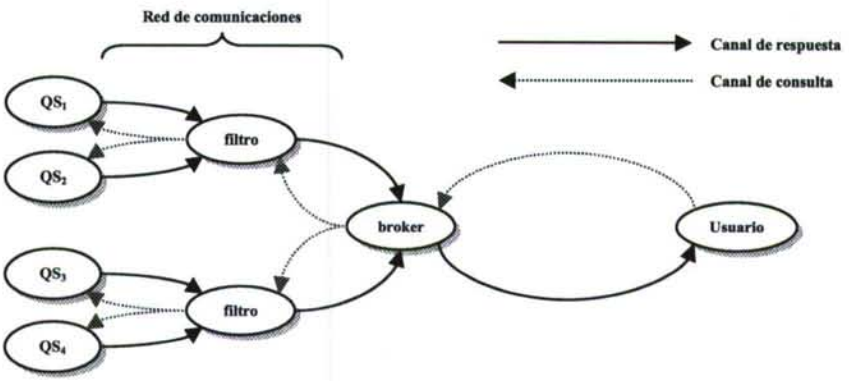

Ilustración 2: Canales de consulta y respuesta en una arquitectura DIR filtrada

Las entidades que en la Ilustración 2 hemos etiquetado como *filtro* son dispositivos activos de la capa de red (pensemos en ellos como *routers* segmentando la red de comunicaciones) con la capacidad embebida de procesar el tráfico que los atraviesa. Cada filtro es el encargado de recoger las respuestas de un subconjunto del total de QS del sistema y generar, a su vez, una respuesta parcial de cardinalidad inferior a la suma de las cardinalidades de las respuestas parciales que recibe.

Las redes programables son el resultado del trabajo de una gran corriente de investigación y desarrollo enfocada en simplificar el desarrollo de nuevos servicios, delegando en las redes de comunicaciones la capacidad de soportar explícitamente el proceso de creación y distribución de nuevos procesos en su interior. De entre los trabajos existentes al respecto, destacamos XBind [Chan, 96], Smart Packets [Kulkarni, 98], ANTS [Wetherall, 98], FAIN [Galis, 00] entre otros.

Estos resultados han podido realizarse gracias a la construcción de un nodo de valor añadido embebido (VAIN) según los diseños mostrados en [Puentes, 04b]. El carácter programable en tiempo de ejecución de esta herramienta ha permitido probar exhaustivamente los diseños detallados en este documento y dado que se trata de una herramienta de propósito general, puede ser usada para extender los estudios experimentales sobre el tema o abordar otras investigaciones donde se requiera el uso de componentes activos en la subred.

La principal ventaja al usar nodos programables como la infraestructura de filtrado es la transparencia. Debido a que estos dispositivos operan como filtros (capturando el tráfico dirigido a los brokers finales, procesándolo y reenviando el resultado parcial obtenido) ofrecen una funcionalidad extra más allá de la funcionalidad propia de los dispositivos de conmutación de paquetes tradicionales y no programables. En el capítulo 6 discutiremos y analizaremos más en profundidad las implicaciones que esta ventaja posee.

1.2 Trabajos relacionados

El trabajo en este documento está relacionado con el análisis del rendimiento de las arquitecturas IR distribuidas, especialmente enfocadas en el filtrado de las respuestas parciales producidas por los servidores de consultas, como una solución complementaria a las diferentes estrategias planteadas en términos de arquitecturas de comunicaciones. Si bien no hemos encontrado soluciones parecidos en el ámbito investigador, si existen trabajos anteriores relacionados con el análisis del rendimiento especialmente enfocado en la arquitectura de red.

Tomasic y García-Molina [Tomasic, 93] han estudiado el rendimiento de varias estrategias de procesamiento de consultas en paralelo usando varias opciones para la organización del los índices invertidos, simulando un pequeño grupo de máquinas interconectadas a través de una red de área local.

Ribeiro-Neto y Barbosa [Ribeiro-Neto, 98] han usado un modelo analítico simple emparejado con un pequeño simulador con el objeto de estudiar cómo el rendimiento de las consultas era afectado por diferentes parámetros (por ejemplo la velocidad de la red) en librerías digitales distribuidas.

Cahoon y McKinley [Cahoon, 96] describen los resultados de los experimentos simulados realizados sobre la arquitectura distribuida INQUERY. Estos autores usaron el comportamiento observado en un servidor aislado con el objetivo de estimar el rendimiento de una implementación distribuida.

Lu y McKinley [Lu, 00] usaron el mismo simulador que los autores anteriormente citados con el objeto de observar y analizar los efectos de una replicación parcial de la información sobre una colección de 1TB de datos.

Burkowski [Burkowski, 90] ha descrito un estudio simulado el cual mide el rendimiento de la recuperación de información en un sistema IR distribuido, usando una pequeña colección de documentos. En sus experimentos se usó una red LAN de alta capacidad para conectar

un grupo de estaciones de trabajo con un *cluster* de servidores, si bien en el modelo de simulación los tiempos de transmisión no fueron considerados.

Coevreur *et al.* [Coevreur, 94] analizaron el desempeño de la búsqueda en grandes colecciones de texto (más de 100GB) en sistemas paralelos. Estos autores usaron modelos de simulación con el objeto de investigar tres diferentes arquitecturas hardware (un sistema *mainframe*, un conjunto de procesadores RISC interconectados mediante una red de área local y una arquitectura de propósito especial) y algoritmos de búsqueda.

Hawking [Hawking, 97] diseño e implementó un sistema IR paralelo sobre un conjunto de estaciones de trabajo, llevando a cabo experimentos con un máximo de 64 máquinas. Para este propósito usó una *Ethernet* conmutada 10BaseT.

Por último, Cacheda, Plachouras y Ounis [Cacheda, 04] [Cacheda, 05] estudiaron el rendimiento de un sistema en *cluster*, replicado y distribuido, sobre una gran colección de documentos Web (SPIRIT [Jones, 02]). La red simulada representaba una LAN compartida operando a 100Mbps y más tarde una red conmutada, siguiendo el modelo de [Tomasic, 93]. Como hemos indicado anteriormente los autores identifican en estos trabajos los dos principales cuellos de botella.

Así pues, aunque podemos citar trabajos relacionados con el rendimiento de los sistemas IR distribuidos, todos ellos se centran en el rendimiento basándose en la organización de los índices locales, el número y distribución de los servidores de consultas y de los *brokers*, o en las arquitecturas de redes de comunicaciones empleadas. Por el contrario, nuestro trabajo representa el estudio y análisis de los aspectos semánticos de las respuestas parciales, su significado, y como a partir de esta información puede inferirse una arquitectura filtrada distribuida de recuperación de información usando redes programables.

1.3 Estructura del documento

Para llevar a cabo el objetivo planteado, este documento se estructura de la siguiente manera: En primer lugar (capítulo 2) presentaremos la arquitectura de nodo programable VAIN (*Value Added Independent Node*), un modelo de nodo de red independiente de la infraestructura subyacente, programable y configurable en tiempo de ejecución. Este capítulo presenta el punto de vista conceptual de la arquitectura así como de los antecedentes de los que hereda parte de su funcionalidad. Seguidamente, en el capítulo 3, desarrollaremos un modelo de filtrado en entornos de IR, describiendo las propuestas de varios algoritmos y estudiando analíticamente sus respuestas como filtros. Estos modelos serán puestos a prueba mediante juegos de ensayo en prototipos cuya implementación será detallada en el capítulo 4 y cuyos resultados pueden encontrarse en el capítulo 5 junto con

el análisis de los mismos. Finalmente, en el capítulo 6, podremos encontrar las conclusiones y un estudio pormenorizado del trabajo futuro a desarrollar en este ámbito.

Como colofón a este trabajo, el capítulo 7 está dedicado a describir las herramientas creadas originalmente para esta tesis, en forma de manual de usuario, y descritas anteriormente en los capítulos 2 y 4. El objetivo de esta sección es de servir de guía a futuros usuarios de la implementación de nuestra herramienta VAIN (*Value Added Independent Node*), tanto para su uso o estudio como para el desarrollo de nuevas características. Este capítulo muestra un recorrido por la funcionalidad externa del nodo, su configuración y programación, e induce a reconocer sus características más importantes así como sus limitaciones.

2 Redes de valor añadido embebido

2.1 Introducción

La arquitectura que estamos describiendo está basada en una topología de red jerárquica donde algunos de sus miembros son nodos programables implementando un algoritmo de filtrado (de entre una familia de algoritmos cuya eficiencia será descrita y analizada en los capítulos 3 y 5) con la finalidad fundamental de reducir el juego de resultados parciales a ordenar por parte de los *brokers* finales o de usuario y generados por los QS. Este capítulo describe conceptualmente la arquitectura de nodo programable que hemos creado ex proceso para esta investigación con los objetivos principales de eficiencia y flexibilidad tanto de gestión como de funcionalidad.

En 1948 Claude E. Shannon dio a conocer el artículo titulado "*A Mathematical Theory of Communication*" [Shannon, 48], el cual es considerado hoy en día como el inicio de una nueva era de comunicaciones digitales. En su introducción el autor escribe:

> *"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. Frequently the messages have meaning; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem."*

En dicho título, C. E. Shannon argumenta que el objetivo fundamental de las comunicaciones es reproducir fielmente[1] en un destino dado, el mensaje transmitido por el emisor. También reconoce que los mensajes poseen significado, sin embargo deja fuera de la teoría de la comunicación cualquier aspecto semántico que estos pudieran tener.

Los mensajes, en efecto, tienen un significado, definido como el "*contenido semántico de cualquier tipo de signo, condicionado por el sistema y por el contexto*" [RAE, 01]. Así pues, el significado o semántica de un mensaje transmitido esta de hecho impuesto por los extremos de la comunicación, los cuales han de estar de acuerdo previamente (usando el mismo canal de comunicación u otro previamente establecido) en el significado de la información a transmitir.

La propia evolución tecnológica [Kleinrock, 61] que siguió al nacimiento de las comunicaciones digitales propició la aparición de dos estándares (o arquitecturas) principales de comunicaciones entre ordenadores. Por un lado el modelo *de jure* OSI (*Open Systems Interconnection* ISO, ITU-TS X.2000; hoy en día tomado como modelo de referencia), y por el otro el modelo *de facto*, la arquitectura TCP/IP; este último como una instancia o implantación particular del primero. Sin embargo, en el contexto que nos ocupa, ambos diseños se caracterizan por relegar el significado de los datos transmitidos a los extremos de la comunicación (y por tanto también su procesamiento). Según ambos modelos, la subred es la encargada de transportar el mensaje desde el origen hacia el destino y, por tanto, esta implementa como mucho la capa de red del modelo OSI (capa IP en la arquitectura TCP/IP)[2].

Desde entonces ha habido multitud desarrollos en el ámbito de las comunicaciones entre ordenadores. De entre ellos destacamos las redes de valor añadido (VAN), las cuales son una especialización de la redes ASP (*Application Service Provider*), que a su vez entregan servicios a sus clientes por medio de una red de comunicaciones. El modelo ASP consiste en un software de aplicación que reside en la infraestructura del vendedor y es accesible a sus clientes por medio de un navegador que usa HTTP (*HiperText Transfer Protocol*), o un software de cliente de propósito especial distribuido por el vendedor. El software

[1] En concreto el texto cita "*reproducir en un punto exacta o aproximadamente* ..."; al hablar de "*aproximadamente*" Shannon reconocía implícitamente la imposibilidad de comunicar datos digitales en un canal con ruido y que estos alcancen el destino con la garantía de que no se producirán cambios en el mensaje.

[2] Existen excepciones a esta afirmación. Las infraestructuras modernas de comunicaciones digitales por conmutación de paquetes poseen en su subred capas superiores a las de red según el modelo OSI, pero con un objetivo puntual y sin tener en cuenta el significado del mensaje, solamente algunos campos de la cabecera, con el fin de proteger, filtrar o equilibrar la carga de la red.

propietario del cliente puede acceder a estas redes por medio de un API (*Application Program Interface*) XML. En general las redes ASP usan el paradigma cliente-servidor.

Como hemos dicho, las redes VAN son una especialización de las redes ASP, en donde las primeras actúan como un intermediario entre socios comerciales compartiendo datos o procesos de negocio. Tradicionalmente emplean como formato de datos el estándar EDI (*Electronic Data Interchange*), pero cada vez más a menudo se está empleando el lenguaje de descripción de datos XML (*eXtended Markup Language*). Volveremos a este tipo de redes más adelante.

Así pues, en el diseño tradicional de redes de comunicaciones sólo se han tenido en cuenta la transmisión de unidades discretas de información descontextualizada entre dos puntos finales (aunque posiblemente con un formato estandarizado). Es lógico, por tanto, que se ignorase la semántica de la comunicación y se concentrase en la sintaxis; esto es, en las estructuras físicas y lógicas necesarias para soportar una comunicación fiable.

Sin embargo, desde hace unos años, existe una corriente investigadora que se caracteriza por su interés en dotar a la subred de comunicaciones de la lógica necesaria, no sólo para mantener la estructura lógica de comunicaciones, si no la de procesar el significado de los mensajes que habitualmente se ha relegado a los extremos de la comunicación.

En este capítulo detallaremos las soluciones existentes que incorporan la semántica del mensaje en el interior del proceso de comunicación, las arquitecturas y estándares actuales; y, finalmente, mostraremos la conceptualización de nuestro nodo VAIN (*Value Added Independent Node*).

2.2 La semántica de las comunicaciones

Frecuentemente, cuando se habla de incorporar semántica a la red, los interlocutores suelen pensar erróneamente en tecnologías como agentes, *applets*, redes inteligentes, etc. Sin embargo ninguna de estas soluciones incorpora de forma concluyente alguna semántica a la red propiamente dicha, si no que consiguen crear arquitecturas que entregan, globalmente, más servicios a los clientes que las usan. Por ejemplos, el término agente es usado habitualmente para denotar aquel código móvil que viaja entre clientes y servidores [Tennenhouse, 96]; similarmente los *applets* son código, típicamente escrito en *Java*, que el servidor envía al cliente para que este lo ejecute en un contexto seguro; Por otro lado el concepto de redes inteligentes suele hacer referencia a aquella infraestructura de redes de comunicaciones, usualmente gestionado por un único proveedor, que entrega servicios de valor añadido al cliente que la usa (como por ejemplo servicios especiales, enrutamiento de llamadas, multidestino, gestión delegada de las llamadas, etc.).

El término *middleware*, es un concepto más cercano a lo que intentamos describir: todo aquel software intermediario entre los diferentes componentes de una aplicación. Sin embargo según sea el contexto, el significado del término puede variar sobremanera, y es por esto que habitualmente dicho término es usado con ambigüedad. Actualmente *middleware* se usa para describir sistemas de gestión de bases de datos, servidores *Web*, servidores de aplicación, sistemas de gestión de contenidos, y herramientas similares. En el contexto que nos ocupa y teniendo en cuenta la definición general, usaremos el concepto de *middleware* activo como aquella lógica activa intermediaria entre los puntos finales de una comunicación.

La computación en cluster tiene como principal objetivo aumentar la potencia de cálculo de una infraestructura sin aumentar el gasto proporcional o incluso exponencialmente [Pfister, 98]. Se trata de agrupar dos o más ordenadores de tamaño medio usando una red de comunicaciones y mediante el diseño del problema de tal forma que sea particionable y por ende distribuible concurrentemente entre las unidades de cómputo; de esta manera se consigue una mejor relación potencia/coste que la que obtendríamos con una solución donde sólo hubiera una máquina de gran potencia, típicamente con un alto coste, superior al diseño en cluster. Dado que el problema está particionado y puede ser ejecutable concurrentemente en las diferentes unidades del cluster, podemos afirmar que estas permanecen pobremente acopladas, esto es, sin apenas relación entre ellas excepto en la entrada y salida de cada una. Bajo nuestra perspectiva un cluster representa una arquitectura distribuida cercana a nuestro objetivo de dotar a la red de parte o toda la semántica del problema, sin embargo un cluster tiene normalmente como emisor y receptor a la misma entidad (o persona interesada en el resultado), y un cluster no es propiamente dicho una infraestructura de comunicaciones, si no de cómputo. Respecto a la computación *grid*, esta es la implantación de un cluster colaborativo a través de, habitualmente, Internet [Catlett, 92].

2.2.1 Redes programables.

Hoy en día existe una gran corriente de investigación y desarrollo enfocada en simplificar el desarrollo de nuevos servicios, delegando en las redes de comunicaciones la capacidad de soportar explícitamente el proceso de creación y distribución de nuevos procesos en su interior. Sin embargo han aparecido dos corrientes de pensamiento respecto a como hacer redes programables [Campbell, 98] que se corresponden con los enfoques *hardware* y *software* del problema: La comunidad *Opensig* argumenta que por medio del modelado del hardware de comunicaciones, usando un conjunto de interfaces de red programables de distribución libre, es posible el acceso por parte de terceros a *routers* y *switches*, y de esta

manera habilitar la introducción en el mercado de las telecomunicaciones a proveedores de software externos.

El concepto de red activa (AN) apareció por primera vez en conversaciones entre miembros de la comunidad de investigación DARPA entre los años 1994 y 1995, en el contexto de las futuras direcciones de las arquitecturas de red. La aproximación basada en AN aboga por la distribución dinámica de servicios en tiempo de ejecución, principalmente en los confines de las redes IP existentes en la actualidad. El grado de soporte en tiempo de ejecución de nuevos servicios va más allá de lo planteado por la comunidad *Opensig*. El caso más extremo propuesto por la comunidad AN es la de "*paquetes activos*" los cuales están compuestos por código y datos, pudiendo alterar por completo la funcionalidad del dispositivo o nodo activo. La aproximación basada en redes activas ofrece la máxima flexibilidad a la hora de dar soporte a la creación de nuevos servicios, pero con el coste de añadir más complejidad al paradigma de desarrollo y distribución, así como al modelo de ejecución.

Cualquiera de las dos corrientes de pensamiento persiguen la misma funcionalidad, dotar a la subred de la capacidad de procesar el tráfico que la atraviesa en términos semánticos.

2.2.2 Soluciones existentes.

Gracias a la experimentación y a la observación global de los problemas sabemos que los sistemas diseñados y optimizados para condiciones específicas alcanzan un mejor rendimiento que las soluciones de propósito general. El compromiso entre generalidad y optimización en el contexto, has sido objeto de estudio en muchos campos.

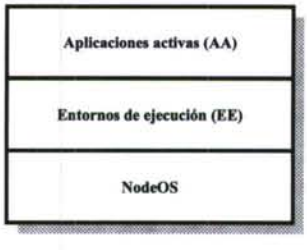

Ilustración 3: Modelo de referencia de un nodo activo

Con el tiempo se ha desarrollado una arquitectura o modelo general de un nodo activo [Tennenhouse, 97] [Campbell, 98] [Peterson, 01]. Esta arquitectura (Ilustración 3) identifica tres capas de código: en el nivel más bajo el sistema operativo del nodo (*NodeOS*) gestiona las comunicaciones, memoria y los recursos entre las diversas entidades o flujos que gobiernan el sistema. En el siguiente nivel uno o más entornos de ejecución

(EE) definen un modelo de programación particular para escribir aplicaciones activas. En el nivel más alto residen las aplicaciones activas (AA) propiamente dichas.

A continuación se mostrarán las soluciones existentes en el contexto de redes programables con el objeto de entender la evolución y el estado del arte de esta disciplina, clasificándolas como pertenecientes a la comunidad *OpenSig*, o, en caso contrario, en alguna de estas tres capas.

2.2.2.1 Xbind (*OpenSig*)

Xbind [Chan, 96] supera las limitaciones que ATM impone en el acceso a sus componentes de señalización debido a su complejidad, aunque esta tecnología, *per nature*, soporta conceptos como admisión de control, reserva de recursos, etc. Lo consigue separando del hardware de telecomunicaciones los algoritmos de control; así pues su énfasis se centra en el diseño de interfaces que habilitan el acceso libre a los recursos y funcionalidad del nodo. Los interfaces son diseñados para soportar la programación de la gestión y el plano de control en redes ATM.

El proyecto *Xbind* ha generado soluciones que incluyen conmutadores ATM usando señalización abierta y creación de servicios, con el objetivo de soportar una amplia variedad de prestaciones de banda ancha, transporte y sistemas de señalización con garantía de QOS.

2.2.2.2 NetScript (*EE*)

El proyecto *NetScript* [Yemini, 96] llevado a cabo en la Universidad de Columbia, usa una aproximación basada en lenguajes funcionales para capturar la semántica de la red. *NetScript* es un lenguaje fuertemente tipado que crea abstracciones universales para implantar las funciones del nodo. A diferencia de otros proyectos basados en lenguajes, *NetScript* se desarrolló para soportar el concepto de redes activas virtuales (VAN) como una abstracción de red programable.

Un hecho distinguible de *NetScript* es su intención de dotarnos de un lenguaje universal para redes activas de manera similar a *Postscript*, el cual captura las capacidades de programación de las impresoras. Igualmente *NetScript* pretende capturar la capacidad de programación de los nodos activos.

Las abstracciones que *NetScript* recoge, incluye colecciones de nodos y enlaces virtuales que constituyen redes activas virtuales.

2.2.2.3 Mobiware (*OpenSig*)

Mobiware [Angin, 98] está basado en Xbind, extendiendo su modelo de programación a redes móviles basados en paquetes, para la distribución de servicios adaptativos sobre enlaces inalámbricos. Este proyecto incorpora programación orientada a objetos (CORBA) para la programación del plano de control, pero también soporta el transporte de objetos serializados basados en *bytecode* Java.

2.2.2.4 Smart Packets (*EE*)

Desarrollado por la Universidad de Kansas [Kulkarni, 98], está fundamentado en el concepto de especialización de paquetes basado en código implementado en un entorno IP programable. *Smart Packets* (SP) representan código móvil, tanto en-banda como fuera-de-banda, basado en clases Java (serialización de objetos). Los nodos activos que dan soporte SP incorporan: controladores de recursos (entrega interfaces a los recursos del nodo), gestores de nodo (que impone límites estáticos al uso de recursos), gestores de estado (controla la cantidad de información que SP puede dejar reservado en un nodo).

Cada SP es procesado en un único *thread* con alguna cantidad de recursos el nodo. Los nodos activos también incluyen gestores de enrutamiento los cuales soportan los esquemas de enrutado por defecto y los métodos de enrutado alternativos planteados por los SP.

SP ha sido usado para programar servidores HTTP extendidos y servicios SMTP que muestran algún beneficio respecto a sus versiones tradicionales, reduciendo respuestas ACK/NACK en los protocolos.

2.2.2.5 ANTS (*EE*)

Desarrollado en el MIT, ANTS [Wetherall, 98] permite el despliegue no coordinado de múltiples protocolos de comunicaciones en redes activas, confiriendo un conjunto de servicios clave entre los que se encuentran el soporte para el transporte de código móvil y la carga de código bajo demanda y técnicas de *caching*.

ANTS entrega un entorno de programación de red para construir nuevas arquitecturas de red programables basadas en cápsulas (paquetes conteniendo datos y código). Los nodos activos basados en ANTS ejecutan cápsulas y rutinas de cálculo del *siguiente-salto*, mantienen estados locales y soportan servicios de distribución de código con el objetivo de automatizar el despliegue de nuevos servicios. Las cápsulas contienen datos e identificadores en sus cabeceras los cuales hacen referencia a código que necesita ser ejecutado pata procesar el paquete. Si el código no existe en el nodo, este es reclamado al

nodo anterior (salto previo) dado que este habrá sido presumiblemente almacenado al paso de la cápsula. Aquí la propagación del código es llevado a cabo por la llegada de un nuevo paquete y se extiende a lo largo de la trayectoria que atraviesa la topología de la red.

Ejemplos de nuevos servicios creados en entornos ANTS son las extensiones *multicast*, *routing* IP móvil y filtrado a nivel de aplicación.

2.2.2.6 Switchware (*NodeOS*, *EE*)

Desarrollado en la Universidad de Pennsylvania, *Switchware* [Alexander, 98] es un claro intento de equilibrar la flexibilidad de una red programada y los requerimientos de seguridad y estabilidad necesarios en una infraestructura compartida como es Internet. Este proyecto permite a los diseñadores de sistemas construir arquitecturas de redes seguras compensando flexibilidad, estabilidad, seguridad, eficiencia y usabilidad.

Al nivel del sistema operativo del nodo (*NodeOS*) un componentes activo es el encargado de proveer un entorno seguro que garantice la integridad del sistema. Las extensiones activas pueden ser cargadas dinámicamente a través de un conjunto de mecanismos de seguridad que incluyen encriptación, autentificación y verificación de programas.

Las extensiones activas proporcionan interfaces para aumentar la capacidad de programación de la red usando paquetes activos (concepto y funcionalidad similar a las *cápsulas*). El código introducido en los paquetes es escrito en lenguajes funcionales (Caml, PLAN) e invocan servicios residentes en el nodo y soportados por las extensiones activas.

Switchware impone mecanismos de chequeo de seguridad muy fuertes a nivel de extensiones activas, sin embargo a nivel paquetes activos estos mecanismos son mucho más livianos. Esta aproximación permite equilibrar los requerimientos de seguridad y eficiencia.

2.2.2.7 CANEs (*EE*)

El proyecto CANEs [Bhattacharjee, 97], liderado por investigadores de la Universidad de Kentucky y el Instituto Tecnológico de Georgia, tiene como principal objetivo el aplicar reglas de composición de servicios como un modelo general de red programable. Hasta el momento, *cápsulas*, paquetes activos o extensiones activas, promovían la creación de nuevos servicios a través de la creación de nuevos bloques constituyentes o añadiendo componentes a servicios existentes. CANEs construye servicios de red compuestos a partir de componentes individuales mediante un método de composición, el cual se especificado mediante un lenguaje de programación con capacidades extendidas que opera sobre dichos componentes para construir servicios de redes programables. Este método de composición

debe poseer: control sobre la secuencia en que los componentes son ejecutados, control sobre los datos compartidos entre componentes, control sobre los tiempos de ejecución, métodos de invocación (eventos que causan la ejecución de un componente), y división de funcionalidad entre múltiples componentes (los cuales pueden existir en el nodo o ser llevados por los paquetes).

2.2.2.8 Smart Packets (*EE*)

El proyecto *Smart Packets* (SP), desarrollado en BBN, tiene por objetivo aumentar la eficiencia de redes grandes y complejas por medio del uso de tecnologías de redes activas. La tecnología SP es usada para mover los puntos de de toma de decisiones de gestión cerca de los nodos que están siendo manejados, abstrayendo, además, los conceptos de gestión en construcciones del lenguaje. Los centros de gestión pueden enviar programas a los nodos gestionados. Así el proceso de gestión puede ser dirigido hacia el interés específico del centro de gestión reduciendo la cantidad de tráfico de fondo y los datos requeridos por la observación.

Un paquete SP consiste en una cabecera y un *payload* encapsulado usando ANEP, y pueden contener programas a ejecutar, resultados de una ejecución, mensajes de información o informes de condiciones de error. Los programas pueden ser escritos en dos lenguajes: *Sprocket* (como el lenguaje C de alto nivel con construcciones relacionadas con la seguridad) y *Spanner* (lenguaje ensamblador de bajo nivel, que produce un código optimizado). Programas escritos en *sprocket* son compilados en código *spanner*, el cual es a su vez, ensamblado en código independiente de arquitectura y colocado finalmente en los SP. Los programas llevan a cabo funciones de red y extracción de información de los MIB.

2.2.2.9 FAIN: Future Active IP Networks (*NodeOS*, *EE*)

El proyecto FAIN [Galis, 00] tiene como principal objetivo el desarrollo de una arquitectura de red activa abierta, flexible, programable y fiable. Este proyecto de investigación y desarrollo fue parcialmente financiado por la Unión Europea y fue llevado a cabo por un consorcio internacional integrado por universidades europeas y norteamericanas además de empresas Paneuropeas.

La plataforma de nodo activo FAIN, consiste en un sistema operativo de nodo (NodeOS), un marco de control de accesos de recursos del nodo y componentes activos para gestión, seguridad y provisión de servicios. Estos elementos forman la base en la cual los entornos de ejecución son desplegados y operados independientemente.

FAIN ha resultado ser útil en servicios como aplicaciones multimedia, redes activas basadas en ORB, servicios VPN, servicios Web, entornos de ejecución de alta productividad, gestión de red basada en agentes móviles, demultiplexación de paquetes activos, distribución de la carga, gestión de entornos virtuales, seguridad en redes activas, *WebTV*, video bajo demanda, y escenarios móviles y *DiffServ*[3].

2.2.2.10 Nodo programable SARA (*EE*)

Un nodo de red SARA [Larrabeiti, 02] es un nodo de red programable cuya peculiaridad reside en que se basa en la llamada arquitectura *router*-asistente, formada por dos elementos: el *router* IP, que encamina paquetes, identifica aquellos susceptibles de un procesamiento especial y los desvía al asistente para que los procese una aplicación específica; por otro lado el asistente es un ordenador de propósito general conectado al *router* mediante una red local de alta velocidad, con la misión de proporcionar un entorno de ejecución especialmente diseñado para lanzar y ejecutar aplicaciones de red con rapidez y seguridad.

En esta arquitectura, desarrollada en la universidad Carlos III de Madrid, el asistente está formado por el sistema operativo anfitrión (distribución Linux) como *NodeOS*, la máquina virtual java, código nativo de acceso a los servicios del sistema operativo, el API de SARA y las aplicaciones de red.

2.2.2.11 Bowman (*NodeOS*)

Aunque no es propiamente dicho un nodo programable, el proyecto *Bowman* [Merugu, 00] ha dado como resultado un sistema operativo de nodo activo (NodeOS), originalmente usado como plataforma para CANESs (δ2.2.2.7) y más tarde promocionado como una plataforma general sobre la cual otros proyectos de nodos programables pueden ser construidos.

Los objetivos de diseño de este NodeOS son: proporciona el soporte multi-*thread* para flujos de larga duración, conservando un *fast-path* para paquetes que no requieran este tipo de procesamiento; globalmente hablando, *Bowman* da soporte para diferentes tipos de topologías, todo ello mostrando un rendimiento razonable, usando el estándar POSIX de tiempo real y planificación del tiempo del procesador.

[3] La lista completa puede encontrarse en *http://www.ist-fain.org/project_demo.html*.

La estructura de un nodo basado en *Bowman* está formado por: el sistema operativo anfitrión, las extensiones Bowman y los entornos de ejecución.

El sistema operativo anfitrión entrega los mecanismos de bajo nivel relativos a la gestión de memoria, creación y planificación de procesos, primitivas de sincronización y servicios de entrada/salida. *Bowman* se implementa encima de esta entidad entregando un interface uniforme sobre diferente hardware y sistemas operativos que soporten el estándar de llamadas al sistema POSIX.

Bowman está basado en extensiones, el cual es un mecanismo análogo a los módulos de carga dinámica[4] en los sistemas operativos tradicionales. Usando este mecanismo el interface puede ser extendido dinámicamente para proporcionar el soporte de abstracciones adicionales.

Este *NodeOS* posee un clasificador de tráfico eficiente que identifica paquetes individuales y los dirige hacia el entorno de ejecución adecuado. Este clasificador se basa en separar el reconocimiento de la estructura del paquete y el criterio de clasificación.

2.2.2.12 Janos (*NodeOS, EE*)

Este *NodeOS* desarrollado en la Universidad de UTAH [Tullmann, 01], hace hincapié en una fuerte gestión de los recursos y el control sobre aplicaciones activas poco fiables. El proyecto *Janos* unifica los conceptos de *NodeOS* (aquí llamado *Moab*) y entorno de ejecución (denominado *JanosVM*) en una única solución. *JanosVM* es una versión modificada de la máquina virtual Java donde las aplicaciones activas se diseñan bajo una arquitectura semejante a ANTS (aquí llamado ANTSR).

Janos se diseñó con el objetivo de prevenir la interferencia de aplicaciones activas separadas una de otra y para entregar a los administradores del nodo un fuerte control sobre las aplicaciones y su uso de recursos.

2.3 Arquitectura VAIN

En secciones anteriores hablamos de las redes VAN (*Value Added Networks*) como infraestructuras intermediarias orientadas a servicios, mediante una red de comunicaciones,

[4] Librerías de carga dinámica (*Dynamic Link Libraries* o DLLs).En la familia de SSOO Unix poseen la extensión (.so).

entre dos o más socios comerciales. Aunque nuestro objetivo, el procesamiento embebido de la semántica de los mensajes, parece alejarse de dicha definición, existe una semejanza entre ambas dado que vamos a describir una arquitectura intermediaria entre dos o más partes de una comunicación, con el objetivo de procesar la información después de haber sido emitida y antes de ser recibida por el o los destinatarios. A tal arquitectura la hemos denominado EVAN (*Embedded Value Added Networks*).

Las redes programables tienen como principal objetivo el dotar a la subred de comunicaciones de capacidad polimórfica sobre las acciones que sus componentes obran sobre la información que las atraviesa. Esto es, su esfuerzo se centra en arquitecturas de nodo programables en tiempo de ejecución.

Las redes activas son un tipo especial de redes programables, donde el nuevo código que ha de procesar el caudal de datos es inyectado formando parte de los mismos datos (paquetes activos) o, al menos, de forma tal que diferentes paquetes puedan ser procesados por diferente código (extensiones activas).

Las redes EVAN que proponemos son un enfoque diferente de estas arquitecturas. Su interés no se centra en como programar (y controlar) los componentes de la subred, ni de cómo distribuir el código que ha de transformar el caudal de tráfico. Si no que centra su interés en la transformación misma, en el uso que se hace de la información. Así una red EVAN puede ser implementada partiendo de redes programables o redes activas. Un nodo VAIN (*Value Added Independent Node*) es un componente no atómico[5] de una red EVAN.

En el momento de establecer un canal de comunicación entre dos (o más) puntos extremos en el contexto de una red programable hay que tener en cuenta que un mayor grado de programación de la red implica inevitablemente un mayor conocimiento de la misma. El grado de dependencia entre la implementación del algoritmo y la red sobre la que es ejecutado, puede ser muy elevado; siendo susceptible a cambios imprevistos en la infraestructura de la red. Además de esto, los proveedores pueden querer, o ver necesario, ocultar peculiaridades de su red de comunicaciones.

Para llevar a cabo la implementación de la arquitectura de red que describiremos posteriormente, nos hemos guiado por los criterios de flexibilidad, abstracción, transparencia e integración.

[5] Un nodo VAIN no es un componente atómico de una red EVAN por que esta está formada por servicios y flujos de información. Cada nodo puede implementar varios servicios y poseer multitud de flujos, internos o con el exterior.

Los nodos que forman parte de una red programable deben poseer las características de flexibilidad y abstracción en lo que respecta a su comportamiento como integrantes de una red de comunicaciones. Esta necesidad de adaptabilidad a su entorno – a los flujos de datos que lo atraviesan y al ambiente en que se encuentra inmerso – lo convierte en un elemento altamente programable, lo que le otorga la capacidad de flexibilidad dado que su comportamiento (globalmente hablando) es susceptible de ser modificado a voluntad. La necesidad de abstracción viene dada, en general, por la falta de conocimiento que en tiempo de diseño se tienen de los procesos o servicios que prestará a los usuarios de la red.

Desde el punto de vista de los dispositivos tradicionales de la subred, los nodos son vistos como otros de los elementos pertenecientes a la misma red tradicional, de esta forma se alcanza el objetivo de transparencia. Así mismo la integración es la necesidad de colaboración entre nodos programables y con la red tradicional para llevar a acabo un objetivo (la semántica global).

2.3.1 El modelo de tres capas VAIN

En el contexto que nos ocupa, la de reducir la influencia de los cuellos de botella sobre la eficiencia de una arquitectura de recuperación de información, las redes EVAN (redes de valor añadido embebido) permiten manipular el significado (la semántica) último del tráfico que atraviesa sus componentes fundamentales (nodos VAIN, nodos independientes de valor añadido).

Cada nodo programable VAIN de una red EVAN posee un diseño basado en tres capas: la capa de comunicaciones, la capa de ejecución y la capa virtual. Este diseño (Ilustración 4) es ligeramente diferente al visto anteriormente (Ilustración 3), en donde dividíamos los nodos programables en tres capas (NodeOS, EE y AA) desde el punto de vista del software. La subdivisión mostrada aquí se basa en una partición basada en las comunicaciones y, desde nuestro punto de vista, permite un diseño más flexible y racional.

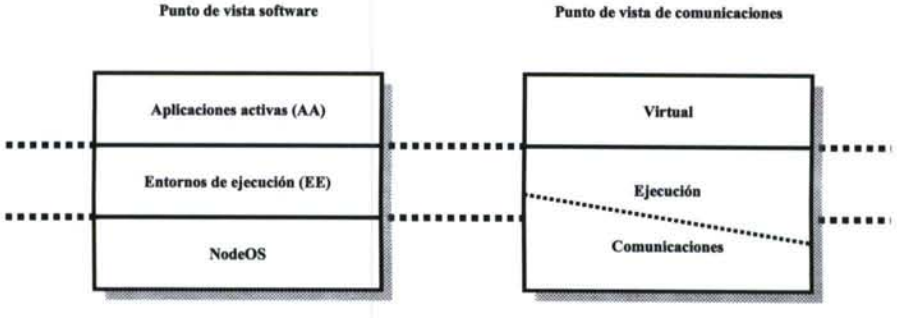

Ilustración 4: Diferencias entre los modelos de nodo programable

De hecho esta diferencia es la base del nuevo enfoque que las redes EVAN poseen sobre el mismo problema. El punto de vista software, usado en redes programables y redes activas, entrega una arquitectura de nodo basada en bloques donde se hace hincapié en cómo se ejecutan los nuevos servicios (AA) y las características funcionales del *NodeOS*. El punto de vista de las comunicaciones, aún teniendo los mismos objetivos, hace hincapié en cómo se mueven los datos en el interior del nodo y entre nodos (sean o no nodos VAIN). Desde luego poseen muchos puntos en común, dado que los nodos VAIN heredan gran parte del conocimiento alcanzado en el desarrollo de nodos programables en general y activos en particular.

La capa de comunicaciones garantiza la comunicación entre nodos y con otros elementos de la subred. Se corresponde con los diseños de redes tradicionales hasta el nivel de red OSI (ISO). Además, es en esta capa donde se lleva a cabo la distribución de tráfico hacia entidades de capas superiores. La capa de ejecución tiene como objetivo controlar los diferentes entornos de ejecución y garantizar las adecuadas políticas de seguridad, concurrencia y acceso a los recursos del nodo. La capa virtual, finalmente, está formada por el conjunto de servicios ejecutándose en un momento dado.

Globalmente hablando, la suma de las diferentes capas de una red programable forman planos (ver Ilustración 5): El plano de comunicaciones se corresponde con el diseño tradicional de una subred. El plano de ejecución permite ver a la red como un gran sistema distribuido. El plano virtual, formado por las capas virtuales de los nodos individuales, establece la abstracción de más alto nivel, proyectándose sobre el plano inferior y consiguiendo así el objetivo de independencia de la infraestructura subyacente (transparencia e integración).

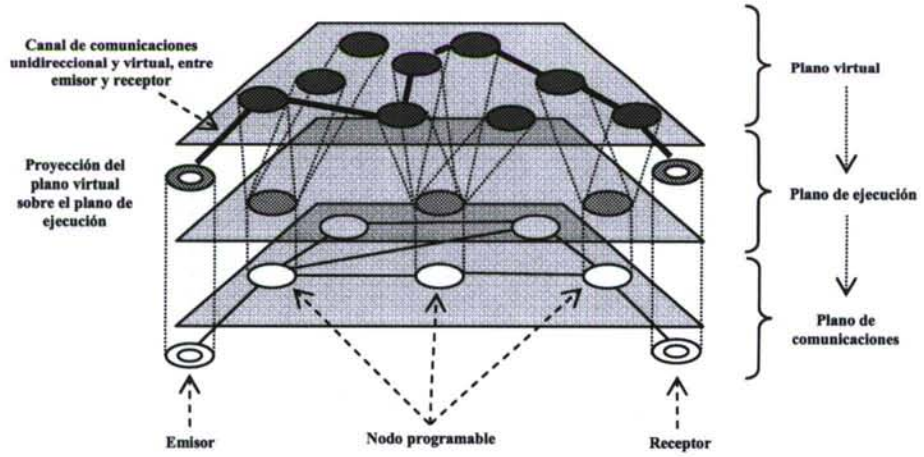

Ilustración 5: Planos en un modelo de tres capas

La división en planos permite a los diferentes actores que manejan las infraestructuras ver el sistema desde tres puntos de vista. El punto de vista del plano de comunicaciones permite diseñar la red tomando en consideración términos como congestión, equilibrio de carga, etc. El propietario de la infraestructura (o infraestructuras con varios propietarios no colaborativos) puede seguir diseñando como hasta ahora su red, con la excepción de situar en determinados puntos nodos programables que habiliten el siguiente punto de vista. El punto de vista del control de la ejecución permite a los diferentes arquitectos de la red programable establecer limitaciones de recursos hardware y software global o independientemente. Finalmente cualquier entidad o actor ajeno puede crear el diseño del plano virtual sin tener en cuenta el diseño o limitaciones impuestos por los planos inferiores.

La capa de comunicaciones se comporta como un *router*, principalmente leyendo tráfico de los interfaces de entrada y calculando el siguiente salto, gracias a lo cual el dispositivo conoce el interface de salida adecuado. En nuestra implementación, entre las subetapas de lectura y *forwarding* (ver Ilustración 6) se lleva a cabo el proceso de demultiplexación, el cual extrae del tráfico los paquetes que deberán ser procesados en entidades superiores (δ2.3.2). Nuestro modelo no impone ninguna estructura o formato concreto del paquete; al revés que otras soluciones permite trabajar con cualquier tipo de tráfico (salvo las limitaciones que veremos más adelante) y la clasificación de este por servicios la lleva a cabo el demultiplexor. Este proceso de catalogación es uno de los ejes fundamentales del modelo VAIN.

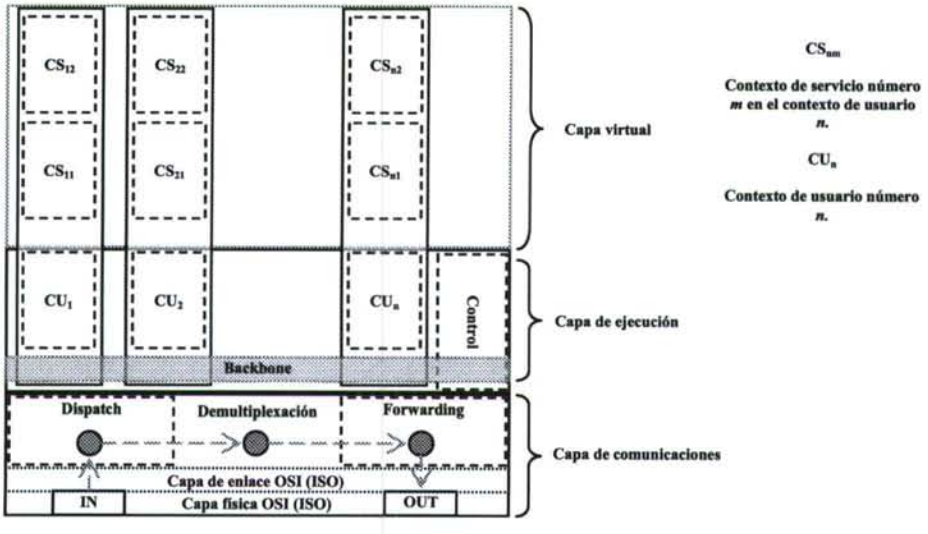

Ilustración 6: Arquitectura de tres capas de un nodo programable VAIN

La capa de ejecución está formada principalmente por el subsistema de control de la ejecución (control horizontal, o de nodo) y los subsistemas de contexto de usuario (control

vertical, o de usuario). Además esta capa implementa el subsistema de interconexión local (*backbone*) del nodo, un mecanismo de transferencia de datos entre el nodo y los entornos de ejecución. Este subsistema permite que las diferentes entidades puedan comunicarse entre sí mediante el paradigma de envío de mensajes. La capa de ejecución es un factor clave que habilita gran parte de la funcionalidad del nodo. No sólo mantiene el control sobre los elementos del plano virtual (servicios) si no que es el encargado de mantener la proyección entre los planos virtuales y de comunicaciones.

Finalmente la capa virtual está formada por un determinado conjunto de servicios (ejecutándose en contextos de servicio, CS), cada uno de ellos perteneciente a un determinado usuario (y cada uno de ellos propietario de un contexto de usuario, CU). Cada usuario y cada servicio poseen su contrapartida como contexto de ejecución. El contexto de usuario (situado en la capa de ejecución) se encarga del control de los contextos de servicios del usuario: los inicia, finaliza, y en general gestiona todos los aspectos tenidos en cuenta en tiempo de diseño del nodo. El contexto de servicio entrega, al código hospedado en el nodo, un interface uniforme y un conjunto de interfaces que habilitará (y controlará) su funcionalidad.

2.3.2 Capa de comunicaciones: clasificación de tráfico

Un aspecto muy importante a tener en cuenta entre las diferentes soluciones de redes activas es la forma en que el tráfico de entrada es procesado. Las estrategias basadas en entornos de ejecución Java, conteniendo código en el interior de los paquetes (objetos serializados), típicamente crean un *thread* para el objeto recién llegado y este pasa a tener el control de parte del flujo de entrada. Por otra parte aquellas otras arquitecturas no basadas en "paquetes activos" llevan a cabo el proceso de clasificar el tráfico entrante antes de distribuirlo hacia los servicios o aplicaciones activas. Los nodos VAIN pertenecen a este último tipo de arquitecturas. En nuestros nodos los componentes encargados de recibir el tráfico (*dispatch*), encaminarlo internamente (*demultiplexor*) y reenviarlo hacia el exterior (*forwarding*) son a su vez servicios, contenidos en el contexto de usuario administrador.

La Ilustración 7 muestra como ejemplo un esquema de las trayectorias seguidas en el interior del nodo. Esta configuración no es obligatoria. Dado que los componentes de entrada/salida son, a su vez, servicios y por tanto pueden ser modificados en tiempo de ejecución, estas trayectorias pueden cambiar a gusto de los administradores/diseñadores de la red activa e incluso convivir varias estrategias de encauzamiento dentro del mismo nodo.

El servicio *dispatch*[6] es el encargado de gestionar las lecturas físicas, entrada de tráfico, del nodo. Su cometido se limita a comprobar la corrección del paquete leído y encolarlo hacia el servicio *demultiplexor* (A). Este servicio ejecuta la rutina – que describiremos más adelante – que obtiene la lista de servicios que han reclamado el tráfico al cual pertenece el paquete sometido al proceso. El servicio *demultiplexor* encola (B) el paquete en la lista de entrada del servicio que ocupa el primer puesto en dicha lista. Una vez procesado el paquete, este es insertado secuencialmente en el resto de los servicios (C) registrados en la lista y finalmente es enviado hacia el servicio (*forwarding*) para ser retransmitido hacia el siguiente salto (D).

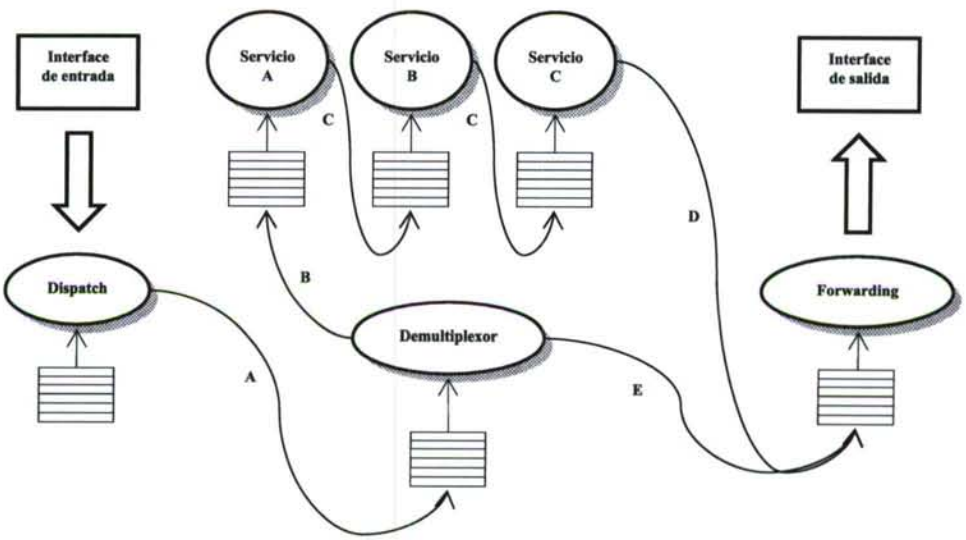

Ilustración 7: Trayectoria de procesamiento de tráfico en el nodo

Los servicios que han procesado el paquete tienen tres opciones: eliminarlo, con lo cual este no será recibido por más servicios ni retransmitido; enviarlo directamente al siguiente servicio de la lista; o reenviarlo al servicio *demultiplexor*. En la Ilustración 7 sólo se muestra la segunda estrategia. El servicio *demultiplexor*, al recibir un paquete por primera vez crea la lista de servicios, y lo reenvía al primero de la lista. Si vuelve a recibir el paquete – con una lista ya creada – reenvía el paquete hacia el siguiente servicio en la lista. De no haber más servicios el paquete es encolado hacia el servicio *forwarding* a la espera de que sea retransmitido.

[6] En nuestra implementación los servicios *dispatch* y *forwarding* son el mismo servicio. En el texto los describiremos como si se tratasen de dos para simplificar la explicación. Al final del texto (δ7.11.2) es posible encontrar una descripción más detallada de este servicio de entrada/salida para tráfico IP sobre redes *Ethernet*.

La solución que hemos creado no crea copias del paquete en caso de haber más de un servicio interesado por él, si no que habilita que este sea procesado secuencialmente. La alternativa, crear copias y que estas sean enviadas en paralelo hacia los servicios, causa una elevada degradación del rendimiento [Varghese, 05] y no permite una procesamiento en *pipeline*, por parte de los servicios.

La necesidad de clasificar el tráfico entrante en un dispositivo aparece desde el principio de la implementación de las redes de conmutación de paquetes. Sea como sea, la motivación básica es la de, a partir de un paquete, identificar una determinada categoría, de entre las que podemos dividir el tráfico de entrada, basándonos en unas reglas que hacen referencia a parte o todo el contenido del paquete. Como ejemplos de clasificación de tráfico, muy simples y muy bien conocidos por la comunidad científica y técnica, tenemos los algoritmos de *forwarding* IP[7], los selectores de protocolos (en las pilas IP) y los selectores de puertos (en las pilas TCP/UDP). Cada uno de estos clasificadores usa una determinada información de las cabeceras presentes en el paquete para determinar el destino de cada uno de ellos (respectivamente, el interface de salida, el gestor del protocolo adecuado y la entidad/cliente del nivel de aplicación destino del tráfico).

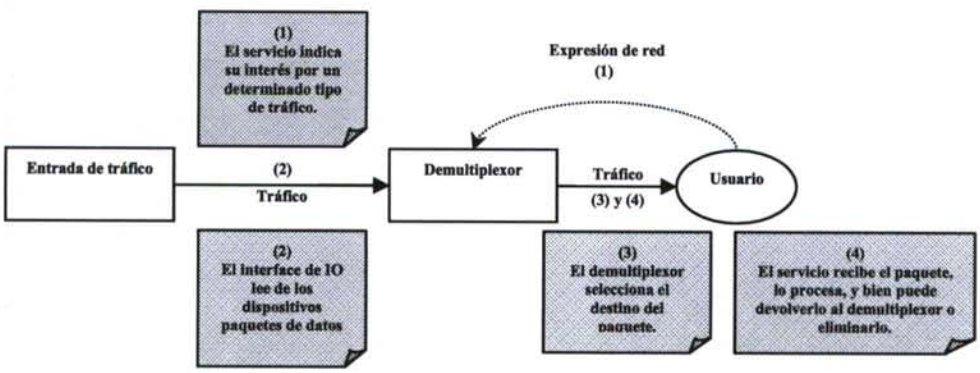

Ilustración 8: Visión global del filtrado del tráfico entrante

Como podemos ver en estos ejemplos, existe una íntima relación entre la información que el paquete contiene y el algoritmo de clasificación. Dado que las redes programables tienen como objetivo procesar todo tipo de tráfico – incluso tráfico con protocolos aun no diseñados – es deseable que su estrategia de clasificación sea lo más independiente posible.

[7] Donde a parir de un paquete se obtiene la entrada de la tabla de enrutamiento que determinará el siguiente salto.

La Ilustración 8 especifica el proceso global que representa la clasificación del tráfico entrante: Un servicio muestra su interés por un determinado subconjunto del conjunto total de entrada, de alguna forma establecida previamente informa al demultiplexor de su interés por medio de la especificación de un criterio, el cual a partir de ese momento y hasta que el servicio informe de lo contrario, analiza el tráfico entrante y desvía hacia el servicio aquellos paquetes que satisfagan el criterio establecido previamente.

2.3.2.1 Trabajos previos de clasificación de tráfico en nodos programables

Históricamente los trabajos anteriores relacionados con la clasificación de tráfico (también se usan los términos filtrado y demultiplexación temprana) parten de la aparición de la conmutación de paquetes [Kleinrock, 61]. De entre los primeros trabajos al respecto destacamos [Mogul ,87], [McCanne, 93] y [Yuhara, 94] (este último como mejora del anterior). Estos trabajos consisten en un lenguaje binario (orientado a pila o a registros) muy cercano a la arquitectura RISC que habilita escribir código que al ejecutarse en el interior del *kernel* del O.S. se evalúan a *true* o *false*, dependiendo si reconocen o no el paquete. Las virtudes de estos primeros trabajos son la rapidez y la generalidad del trabajo que llevan a cabo. Entre sus desventajas se encuentra el hecho de que los filtros son independientes entre ellos, realizan cálculos redundantes (con filtros que tienen subexpresiones comunes) y no reconocen partes opcionales y/o repetitivas del paquete (ya sea cabeceras o datos). *Pathfinder* [Bailey, 94] es una solución a los problemas encontrados en los anteriores, al igual que estos es genérico, pero además puede manejar convenientemente cabeceras de longitud variable. *Dynamic Packet Filter* (DPF) [Engler, 96b] es una de las referencias para hacer nuestro trabajo. Respecto a MPF [Yuhara, 94] y *Pathfinder*, DPF avanza en la implantación del filtro en código binario de la máquina huésped y sobre todo la optimización de expresiones comunes. El lenguaje usado, sigue siendo de lógica de predicados.

Respecto a la clasificación de tráfico en el entorno de las redes programables, destacamos los trabajos de LARA++ [Schmid, 00], FAIN [Galis, 00] y *Bowman* [Merugu, 00]. El primero usa un ordenamiento jerárquico de los filtros en forma de árbol, el cual es interpretado a cada llegada de un paquete. Los filtros no son genéricos, reconocen los valores de campos de protocolos diseñados en tiempo de compilación. *Bowman* [Merugu, 00] es la otra referencia de nuestro trabajo. Esta aproximación al problema que nos ocupa define por un lado la estructura en campos que debe de tener el paquete y por el otro permite definir filtros sobre dicho esquema. *Bowman* no puede reconocer campos por encima de cabeceras de longitud variable u opcional, y hace recaer sobre el filtro todo el poder de cómputo del proceso, dado que los filtros son interpretados.

Las siguientes subsecciones detallan el funcionamiento del clasificador de tráfico creado *ex proceso* para el nodo VAIN. En primer lugar detallaremos como el demultiplexor es instruido para reconocer cualquier tipo de tráfico (esquemas de red), seguidamente como los clientes/servicios pueden especificar el criterio, sobre el esquema de red, que el demultiplexor debe usar y como estos criterios forman canales de tráfico. Finalmente mostraremos las ventajas y desventajas de esta implantación y las medidas de eficiencia como un resultado parcial de esta tesis.

2.3.2.2 Esquemas de red.

Al igual que [Merugu, 00] nuestra arquitectura separa la declaración y el reconocimiento de la estructura del paquete de la declaración del criterio que deberá reconocer dichos paquetes. Básicamente un esquema de red es un documento en memoria que especifica los campos de uno o más protocolos y como estos deben de relacionarse (qué protocolo aparece dentro de cual otro y en que orden). Al final de este documento (δ7.8) podemos encontrar una referencia más extensa y detallada de la implantación de los esquemas de red.

El demultiplexor del nodo mantiene un esquema de red global, la representación conceptual en memoria de todas las posibles formas de interpretar un paquete de datos. Este es un documento en forma de árbol donde cada nodo puede representar un protocolo, un campo (atómico o compuesto), un bloque opcional, el final del reconocimiento (*payload*) o una expresión de red (el o los criterios). Cada nodo tiene su propio significado y es tratado de forma diferente. Inicialmente el demultiplexor posee un esquema vacío y, a medida que los clientes así lo indican, va completando el esquema de red añadiendo a la estructura jerárquica los esquemas que aquellos proveen. Por ejemplo, al iniciar el servicio que reclamará tráfico IP versión 4, este instala[8] en el demultiplexor el esquema de red correspondiente al protocolo siguiente:

```
install scheme=
{
 protocol=
   {
    name="ipv4",
    field={ name="version",  length="4",mustbe="ipv4.version==4"   },
    field={ name="hlen",     length="4"                            },
    field={ name="TOS",      length="8"                            },
    field={ name="tlen",     length="16"                           },
    field={ name="ident",    length="16"                           },
    field={ name="flags",    length="3"                            },
    field={ name="offset",   length="13"                           },
    field={ name="TTL",      length="8"                            },
```

[8] El código se corresponde con la orden de instalar un determinado esquema de red, en este caso IPv4, expresado en el lenguaje CL (ver δ7.6para una referencia exhaustiva del lenguaje CL).

```
    field={ name="protocol", length="8"                          },
    field={ name="checksum", length="16"                         },
    field={ name="origin",   length="32"                         },
    field={ name="target",   length="32"                         },
    field={ name="options",  length="8*((ipv4.hlen*4)-(5*4))"    },
    payload
   }
  }
```

Otros servicios instalarán sus propios esquemas, bien a la misma altura relativa dentro del árbol que el mostrado anteriormente (otros protocolos de red) bien como descendientes de alguno ya instalado. Por ejemplo, el servicio UDP instala el siguiente esquema:

```
install scheme=
         {
          protocol=
            {
             name="udp",
             mustbe="ipv4.protocol==17",
             field=
               {
                name="port",
                field={ name="origin", length="16" },
                field={ name="target", length="16" }
               },
             field={ name="length",    length="16" },
             field={ name="checksum",  length="16" },
             payload
            },
         }
        anchor="ipv4"
```

En este caso el esquema – realmente subesquema – de red especifica los campos del protocolo UDP y, además, que este debe instalarse como un subárbol descendiente[9] del protocolo IP (*anchor*).

A medida que los servicios se inician van instalando los protocolos que el demultiplexor debe reconocer (o quizá van preguntando si el demultiplexor ya tiene instalado dicho protocolo). Al finalizar su ejecución el protocolo es desinstalado si no hay ninguna referencia a él y por tanto puede ser eliminado sin repercutir en el funcionamiento de cualquier otro servicio o cliente.

El demultiplexor mantiene la estructura global actualizada y exporta un API que habilita la gestión del mismo[10]. Ante determinados acontecimientos este genera, a partir del árbol que representa el esquema global, el código máquina que realmente es el encargado de clasificar el tráfico entrante.

[9] Realmente será insertado inmediatamente antes y como hermano del nodo *payload* del protocolo IP.

[10] Un API binario o bien en lenguaje CL o XML.

2.3.2.3 Generación de código

Así pues realmente existen dos partes bien definidas en el demultiplexor, el API-control y el demultiplexor dinámico; el primero es el responsable de atender las peticiones del usuario y generar en tiempo de ejecución el segundo (ver Ilustración 9). El demultiplexor dinámico genera una tabla por cada paquete procesado conteniendo las referencias de los clientes que han solicitado tráfico con las características que el paquete en cuestión cumple.

La generación de código se produce en dos etapas: en una primera la parte estática genera a partir del esquema almacenado, interpretándolo, llamadas al ensamblador en tiempo de ejecución (δ4.2) creado para este proyecto. En una segunda etapa el ensamblador genera en una porción de memoria reservada de forma dinámica[11], el código que finalmente reconoce y clasifica el tráfico.

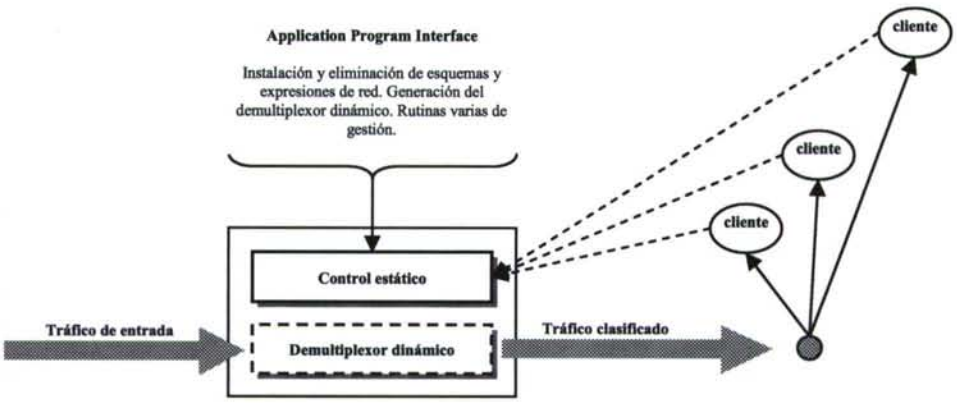

Ilustración 9: Tipo de demultiplexores

Al generar el código, el árbol que representa el esquema es recorrido de tal forma que un nodo es visitado (esto es, generado) antes que otro si y sólo si el primero ocupa una posición dentro del paquete anterior al segundo bajo el criterio del desplazamiento absoluto dentro del paquete. Los únicos nodos que se apartan ligeramente de esta descripción, son los nodos que representan a los protocolos, que dentro del mismo nivel y el mismo antecesor, no importa[12] el orden en que estos sean generados (visitados).

[11] *Malloc* y *realloc*. No usa la memoria reservada a la pila o a datos estáticos. Al contrario que otras implantaciones, como por ejemplo DPF; nuestra estrategia evita colisiones en las cachés del procesador de datos e instrucciones, lo que degradaría la eficiencia del código.

[12] Aunque realmente tiene una incidencia sobre el rendimiento. Situando antes los protocolos más comunes se puede acelerar *ligeramente* la eficiencia del algoritmo.

Podemos describir este u otro recorrido r cualquiera como una lista ordenada de nodos $r = \{ n_1, \ldots, n_n \}$, donde para todo par de elementos de la lista, $n_i < n_j$ si y sólo si n_i ocupa siempre una posición dentro del paquete previa a n_j.

Sea $\rho(n_i)$ el protocolo al que pertenece el nodo n_i, y sea $\omega(n_i)$ el último nodo de los descendientes directos de n_i (conjunto vacío en caso de ser atómico). Sea también $P(n_i)$ la lista ordenada de nodos del tipo protocolo antecesores de n_i.

Sea un nodo n_i del árbol que representa un esquema, definimos su recorrido parcial $r(n_i)$ como la lista (incluyendo a n_i) ordenada de nodos que habrá que visitar previamente para alcanzar dicho nodo con la siguiente condición:

$$\forall\, n_r, n_s \in r(n_i), \text{ si } n_r < n_s \Rightarrow P(n_r) \subseteq P(n_s)$$

Esto es, si n_r es visitado antes que n_s (por tanto ocupa siempre una posición anterior a n_s dentro de cualquier paquete posible) entonces los protocolos donde está definido n_r transitivamente es un subconjunto de los protocolos donde está definido n_s transitivamente.

El último elemento de $r(n_i)$ es precisamente n_i, y la condición garantiza, por ejemplo, que los campos del protocolo UDP nunca formarán parte del mismo recorrido que los campos del protocolo TCP[13].

Con las precisiones dadas, definimos el recorrido total r_t del nodo n_i como $r_t(n_i) = r(\omega(\rho(n_i)))$, esto es, como el recorrido que acaba en el nodo de tipo *payload* del protocolo al que pertenece el nodo dado[14].

2.3.2.4 Expresiones de red, canales de tráfico: Coherencia

Las expresiones de red (δ4.3 y δ7.7) están formadas por expresiones aritmético-lógicas-relacionales, de hecho son un subconjunto de las reglas gramaticales de las expresiones del lenguaje C, resultado de excluir aquellas que implicaran un cambio en los operandos de la expresión (asignación, auto incremento, etc.). Los operadores que podemos usar pueden ser relaciones (mayor, menor, igual a, etc.), booleanos (*and*, *or*, *not*, *xor*), a nivel de bit (*and*, *or*, *not*, *xor*) de desplazamiento (*right-shift*, *left-shift*) y aritméticos (suma, resta, módulo, etc.).

[13] Siempre y cuando hayamos definido los protocolos TCP y UDP como descendientes de otro protocolo, esto es, siendo hermanos.

[14] El último nodo hijo de un nodo de tipo *protocol* es siempre de tipo *payload*.

Los operandos pueden ser constantes sin signo (decimal, octal, hexadecimal, binarios y direcciones IP de 32 bits), variables y funciones preestablecidas y trayectorias (*path*) que hacen referencia a un campo del esquema (por ejemplo *ipv4.udp.port.target*).

Los clientes especifican el criterio por el cual muestran su interés por un determinado subconjunto del tráfico entrante, mediante la instalación en el esquema de una o más expresiones de red. Dicha expresión es creada en el árbol que representa el esquema como un nodo hijo de uno o más nodos de tipo *payload*. En cual o cuales nodos ha de ser instalada la expresión es un problema a tener en cuenta.

Sea $l(e_x)$ la lista de los operandos de la expresión e_x, tales que hagan referencia a un nodo del esquema, esto es, operandos tipo *path*.

Una expresión e_x es coherente con un esquema S dado si y sólo si:

$$\forall f \in l(e_x), f \in S$$

Esto es, si para todo operando de la expresión tipo *path* este está definido en el esquema.

Digamos también que expresión e_x es coherente con respecto al nodo n_i perteneciente al esquema S si y sólo si:

$$\forall f \in l(e_x), f \in r(n_i)$$

Esto es, si para toda referencia en la expresión a un nodo del esquema, esta referencia pertenece al recorrido del nodo n_i.

Podemos demostrar que si una expresión es coherente con el nodo n_i perteneciente al esquema S, entonces esta misma expresión es coherente con el esquema S. Por reducción al absurdo, supongamos que la expresión e_x es coherente con n_i, dentro de S y así mismo e_x no es coherente con S. Entonces por definición de coherencia con un esquema, existirá una referencia en e_x, tal que no exista en S, lo cual es absurdo, dado que partimos del hecho de que todas las referencias de la expresión pertenecen al recorrido del nodo sobre S y por definición todos los elementos de dicho recorrido pertenecen a S.

Así pues la coherencia de una expresión sobre un nodo perteneciente a un esquema implica que dicha expresión es coherente con el esquema.

La definición de coherencia de una expresión es realmente útil, dado que cada expresión en forma de árbol será insertada como un subárbol hijo de aquellos nodos p_i tipo *payload* del

esquema tal que se cumpla que la expresión sea coherente con p_i. Veamos un ejemplo negativo:

```
local(ipv4.target) AND (ipv4.udp.port.target==80 OR
ipv4.tcp.port.target==80)
```

Esta expresión se interesa por los paquetes dirigidos al nodo (dirección destino local), en concreto al puerto 80 bien TCP o UDP. Este filtro es coherente con el esquema en que está definido, pero no podrá ser introducido en él, dado que especifica en la misma expresión dos campos pertenecientes a dos protocolos que nunca serán reconocidos al mismo tiempo, y por tanto no será posible encontrar ningún nodo del tipo *payload* que posea en su recorrido, al mismo tiempo, los nodos que hacen referencia a *ipv4.udp* y *ipv4.tcp*. En general diremos que este tipo de expresiones son incoherentes con la semántica del esquema; siendo la definición anterior de coherencia o incoherencia con respecto a la estructura o sintaxis de un esquema,

Para instalar la expresión anterior, debemos hacerlo como dos expresiones por separado:

```
local(ipv4.target) AND ipv4.udp.port.target==80
local(ipv4.target) AND ipv4.tcp.port.target==80
```

Ahora cada una de estas dos expresiones es coherente con la semántica del esquema y por tanto determinarán efectivamente un subconjunto del tráfico de entrada.

Finalmente definimos el canal de una expresión coherente con la semántica de un esquema, como el medio por el que fluye un subconjunto del tráfico de entrada determinado por una expresión así mismo coherente con dicho esquema, en dirección al servicio o cliente que ha creado el canal.

Un canal es abierto por un cliente al proporcionar e instalar una expresión de red coherente con el esquema activo en ese momento, es usado para leer el tráfico de red desviado hacia ese cliente y al ser cerrado provoca que dicha expresión sea eliminada del esquema en todos los nodos *payload* donde fuera instalada.

2.3.2.5 Ventajas y desventajas del clasificador de tráfico independiente.

El modelo ITC posee como característica principal la de separar el reconocimiento de la estructura del paquete del proceso de clasificarlo con el objetivo de reenviarlo hacia aquellos servicios que lo habrían reclamado previamente. La única conexión entre ambos términos es la coherencia de las expresiones de red respecto al esquema en el que se sitúan. Un demultiplexor ITC, puesto que usa un esquema de red dinámico y desconocido en tiempo de diseño, es absolutamente independiente del diseño de los protocolos que lo

atraviesan, dado que es posible incorporar en tiempo de ejecución una nueva especificación y que incluso esta conviva con versiones anteriores e este u otros protocolos.

Precisamente la mayor limitación del demultiplexor ITC es el tipo de protocolos que reconoce. Estos deben estar basados en campos de longitud fija o variable, pero en cualquier caso, calculable a partir de otros datos incluidos en la misma cabecera. Por ejemplo, los protocolos del tipo HTTP quedan fuera de esta clasificación, puesto que el campo URL de la primera línea[15] posee una longitud no calculable en virtud a los valores de campos anteriores, si no que está supeditado a la aparición de un determinado carácter (espacio en blanco) que marca el final de la dirección.

Otra limitación está relacionada con la característica del código del demultiplexor ITC generado. El algoritmo creado dinámicamente no tiene memoria, no puede relacionar lo acontecido en el instante *t*, con lo acontecido en el instante *t*-1. Por ejemplo, en caso de estar interesados por el tráfico dirigido a un puerto UDP determinado, y tener que dicho tráfico alcanza el nodo fragmentado, los fragmentos que no incluyan la cabecera UDP no podrán ser relacionados, por lo que no serán incluidos en el canal.

La primera limitación es inherente al propio diseño del demultiplexor; incorporar, por ejemplo, la posibilidad de *pattern matching* mediante expresiones regulares es una solución ambiciosa aunque ocasionaría un elevado tiempo de procesamiento por paquete. La solución adoptada se limita a depositar la responsabilidad de reconocer diseños de protocolos orientados a texto, en los servicios, de los cuales se espera que reclamen un súper conjunto del tráfico en el que están interesados para luego seleccionar aquellos paquetes que, en concreto, sean de su interés.

Respecto a la ausencia de memoria en el algoritmo generado, en la práctica sólo nos hemos encontrado la necesidad de reconocer tráfico fragmentado, por lo que el nodo se limita a reconstruir los paquetes IP fragmentados antes de someterlos al demultiplexor.

En la Ilustración 10 podemos observar las medidas comparadas del tiempo empleado para procesar cada paquete de entrada comparado con los resultados mostrados en [Merugu, 00] teniendo en cuenta el número de expresiones incorporadas en el demultiplexor. En concreto debemos de fijarnos en las tendencias de ambas curvas (ITC *rtasm* y *Bowman* en microsegundos) dado que ambas estrategias han sido probadas en arquitecturas diferentes.

[15] La primera línea de texto de un comando HTTP, está formado por tres campos: el método (básicamente GET o PUT), la URL (texto de tamaño indeterminado que finaliza al encontrar un espacio en blanco) y la versión del protocolo (HTTP1/1, por ejemplo). La línea finaliza con un retorno de carro.

Así pues, ambas líneas de tendencia pueden ser tomadas como medida de la complejidad temporal. Nuestra solución crece monótonamente con una pendiente claramente inferior a la de la estrategia presentada por la arquitectura *Bowman*.

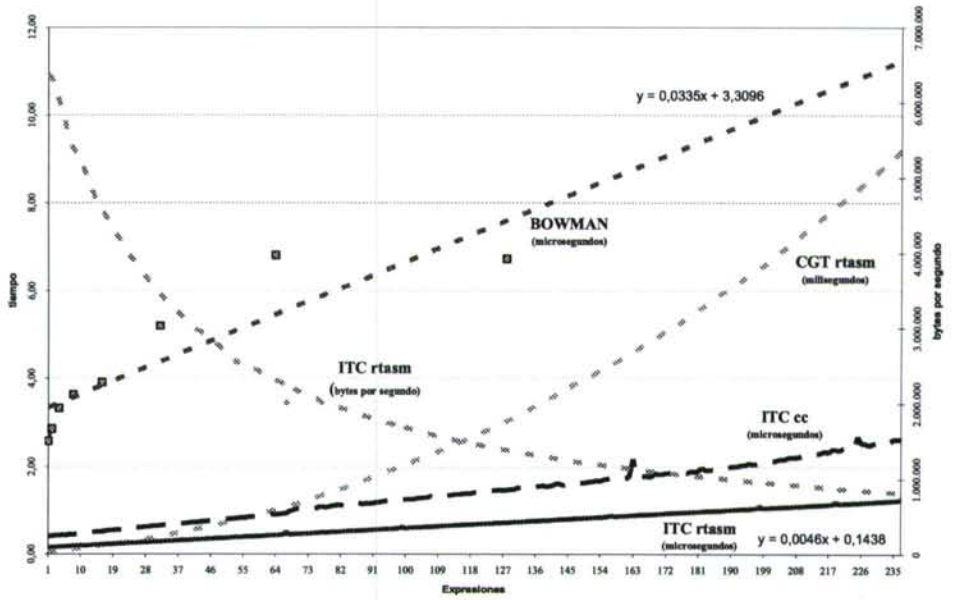

Ilustración 10: Eficiencia del demultiplexor ITC

La ilustración muestra otras curvas que hemos creído interesantes para comparar o entender mejor los resultados. ITC *rtasm* en bytes por segundo muestra el ancho de banda soportado por el demultiplexor generado (debe ser tomado como una cota superior de la eficiencia total de un sistema que emplee esta estrategia), podemos observar que esta disminuye logarítmicamente a medida que aumentamos el número de expresiones de red; esto está ocasionado principalmente por el tiempo empleado en generar el código dinámico (CGT *rtasm* en milisegundos). Finalmente como comparación hemos incluido la curva equivalente a ITC *rtasm*, pero empleando un compilador de C[16] en lugar de nuestro ensamblador RTASM, la curva ITC *cc* muestra una tendencia lineal pero de pendiente superior a la versión anterior, dado que al generar el código cercano a la solución óptima (el compilador no es capaz de encontrar la optimización óptima de las expresiones) conseguimos un código más eficiente.

[16] Tiny C Compiler (*tcc*); URL: http.//fabrice.bellard.free.fr/tcc

2.3.3 Capa de ejecución: contextos de ejecución

El la Ilustración 6 podemos observar como la capa de ejecución contiene principalmente tres elementos: el control de la capa, los contextos de usuario y un medio de comunicación (*backbone*) entre estos elementos.

Una de las actividades principales de un sistema operativo es la gestión de procesos. Cuando hablamos de nodos programables (o *NodeOS*), el término proceso se sustituye por contexto o entorno de ejecución, pudiendo ambos ser considerados sinónimos si no se especifica lo contrario. Así pues los contextos de ejecución entregan un entorno a la ejecución al código, eficaz, eficiente, flexible y seguro. La tecnología Java ha sido la elegida para implementar un gran números de estos entornos por ofrecer, además de la portabilidad de código, un modelo de ejecución donde los programas con comportamientos erróneos o mal intencionados se puedan ejecutar sin causar daños a terceros o al propio entorno. Sin embargo, al usar el modelo de ejecución de Java se opta por un entorno poco eficiente y con un control muy limitado del uso de los recursos por parte de las aplicaciones. Respecto a este último punto, en los últimos años, han aparecido diversas soluciones bajo dos estrategias: por un lado incluir el control en las propias aplicaciones [Czajkowski, 98] [Binder, 01] y por el otro incorporar los mecanismos de control en la máquina virtual [Back, 00] [Tullmann, 01]. Sea cual sea la estrategia que finalmente se imponga, el código Java ejecutado en tales contextos de ejecución lo harán aún menos eficientemente, aunque mucho más eficaz.

Al diseñar el nodo VAIN hemos optado por una solución radicalmente opuesta. Hemos usado como sistema operativo anfitrión una distribución basada en el *kernel* de Linux[17] y lo hemos considerado como el precursor del *NodeOS*. Como entorno de ejecución hemos usado, por tanto, el modelo de procesos del S.O. y como compilador/lenguaje el GNU/C++. El contexto de ejecución, en nuestro caso, no está basado en una máquina virtual, no existe código interpretado ni este se compila al tiempo que se ejecuta (*just-in-time*). El código se escribe en C++ y se compila en tiempo de ejecución del nodo.

Este diseño posee ventajas y desventajas: las ventajas residen básicamente en la eficiencia del entorno y de las amplias posibilidades de acceso al sistema, aspecto, por otro lado, necesario para una plataforma de prueba y aprendizaje. Respecto a las desventajas están la falta, relativa, de portabilidad y la gran elegancia de un entorno java, además del carácter dinámico de los contextos de ejecución de esta plataforma.

[17] Suse Linux v9.0, durante la mayor parte del tiempo de desarrollo.

Respecto a los contextos de ejecución en VAIN, estos pueden ser de más de un tipo: contexto de usuario (CU) y contexto de servicio (CS). Los CU se corresponden con el concepto de proceso de un SO moderno, residen en la capa de ejecución, son estáticos y existe uno por usuario del nodo.

En el control de capa reside básicamente la lógica que gobierna el control y acceso a los recursos del nodo, y es común a todos los contextos de ejecución. A su inicio esta entidad es la encargada de inicial el CU del administrador y la lista de servicios (CS) que habrán de iniciarse por defecto, cada uno de ellos en su respectivo contexto de usuario. Una vez iniciado, el control de ejecución es el encargado de mediar entre el servicio y el sistema operativo local, controlando el acceso a los recursos y entregando las API que estos usarán.

En nuestra implementación el API de ejecución está formado por las primitivas englobadas en los siguientes paquetes: sistema, *shared memory*, *mailbox*, dependencias y *sockets*.

El paquete de sistema (*sys*) entrega primitivas de diversa índole, entre las que destacan la obtención del nombre del servicio (la trayectoria), el nombre del usuario del servicio (propietario), obtención del *pid* del proceso, finalización del servicio, tiempo consumido desde el inicio del servicio, etc.

El paquete denominado *shared memory* (*sys.shm*), permite a los servicios acceder al uso de la memoria compartida del sistema la cual es usada como repositorio global y como base para el envío de mensajes. Las primitivas son similares a las de la gestión de memoria de la librería *libgc*.

El paquete *mailbox* (*sys.mbx*) implementa el sistema de envío de mensajes en el interior del nodo, básicamente entre servicios. Las primitivas permiten enviar (síncrona y asíncronamente), recibir y confirmar, así como esperar por la recepción de un mensaje.

Un subpaquete del anterior citado (*sys.mbx.deps*) permite secuenciar el inicio de los servicios, registrando dependencias entre ellos, de forma que un servicio puede esperar a que otros existan en el nodo y similarmente finalizar su ejecución cuando algún otro servicio desaparezca del nodo.

Finalmente el paquete *sockets* (*sys.socket*) entrega a los servicios la funcionalidad similar que podemos encontrar en la gestión de los *sockets* BSD (*accept*, *bind*, *connect*, *listen*, *read*, *write*, *close*). En este contexto los servicios tanto pueden ser usuarios de este API como ser destinatarios del mismo. Por ejemplo el servicio UDP implementa este API con el objeto de que cualquier otro servicio lo pueda usar.

Esta capa es al mismo tiempo la responsable del paradigma de ejecución de los servicios, entregando la funcionalidad y acceso a los paquetes mediante el mecanismo de herencia. Esta capa implementa la clase *context* la cual es la clase padre de cualquier servicio que desee ser instalado en el nodo.

Esta capa es la responsable, también, de la conversión de lenguaje fuente a código máquina. Hemos usamos el compilador nativo GCC que viene instalado con la mayoría de las distribuciones Linux. Los servicios son entregados en modo texto como lenguaje fuente (C++) y compilados en tiempo de ejecución bajo demanda.

2.3.4 Capa virtual: servicios

Finalmente la capa virtual está formada por el conjunto de los contextos de ejecución que hemos denominado contextos de servicio, en donde, en cada uno de ellos, se ejecuta un servicio independientemente de los demás. Desde el punto de vista del SO anfitrión, cada servicio se ejecuta en un *thread* del proceso de usuario[18] al que pertenece dicho servicio. Esto tiene como consecuencia que heredamos las políticas de seguridad de acceso y ejecución del sistema operativo anfitrión; además el código de un proceso no interferirá en otro en caso de mal funcionamiento dado que el SO anfitrión mantiene espacio de direcciones/pila y juegos de registros separados para cada uno de los procesos (contextos de usuario).

El modelo de ejecución de un servicio puede ser tanto proactivo como reactivo, esto es, puede por si sólo generar tráfico independientemente del tráfico recibido o bien reaccionar únicamente a la llegada de este. Un servicio puede, incluso, no recibir ni enviar tráfico, en cuyo caso su labor es de monitorización.

En la δ7.11 se encuentra una lista detallada de los servicios creados para llevar a cabo el objetivo de esta tesis.

[18] El contexto de usuario se ejecuta en un proceso y los contextos de servicio de cada usuario en un *thread* dentro del proceso de dicho usuario.

3 Mecanismos de filtrado en ámbitos de IR

3.1 Introducción

Con lo visto en el capítulo 2, y teniendo en cuenta lo introducido al respecto en el capítulo 1, se trata por tanto de convertir una red de comunicaciones pasiva en otra activa que aleje el horizonte de congestión y disminuya la cantidad de respuestas, dado que finalmente sólo nos interesan las 1000 mejores (siguiendo a TREC [Craswell, 02] [Clarke, 04]) de entre las respuestas emitidas originalmente por los QS. Nuestra solución pasa por posicionar uno o más nodos programables (Ilustración 2) en el circuito de respuesta entre los actores de la arquitectura haciendo las veces de filtro y de conmutador de paquetes al mismo tiempo.

Pongamos como ejemplo de implementación del canal de respuesta en una arquitectura DIR aquella en que cada QS, como réplica a cada consulta, envía un paquete UDP conteniendo 1000 respuestas, cada una de ellas como un par de números enteros sin signo de 32 bits (*identificador*, *peso*) donde la clave es *identificador* y el orden está impuesto por el campo *peso*, de mayor a menor dentro de cada paquete. El objetivo final es seleccionar de entre todos los pares generados el subconjunto único formado por los 1000 documentos de mayor peso, ordenados de mayor a menor.

Este capítulo se encarga del estudio de dichos filtros y se estructura de la siguiente manera: En primer lugar definiremos un modelo teórico de filtro IR, deduciremos los parámetros de calidad que los caracteriza y que luego nos permitirá compararlos; seguidamente estableceremos una clasificación en familias basado en parámetros cuantitativos y cualitativos. Más adelante describiremos los algoritmos que serán parte de nuestro estudio y deducidos a partir de la experiencia previa y de la descomposición en familias de la subsección anterior. El siguiente paso será medir experimentalmente aquellos parámetros

que no hayamos podido inferir de manera analítica y confirmar aquellos que habremos deducido en las subsecciones anteriores con el fin de tener a nuestra disposición todos los criterios necesarios para comparar los algoritmos mostrados.

3.2 Modelo del filtro IR

El tipo de filtro que someteremos a estudio y desarrollo se basa en el modelo mostrado en la Ilustración 11, en donde podemos observar una entidad que recibe un total de n_{in} documentos agrupados en p paquetes, ejecuta un algoritmo en t_f unidades de tiempo en total y envía n_{out} documentos agrupados en q paquetes para lo cual consume t_t unidades de tiempo. El tiempo que consume para llevar a cabo su propósito lo mediremos sin tener en cuenta la recepción de tráfico, que consideraremos incluido en el tiempo t_t de la etapa anterior.

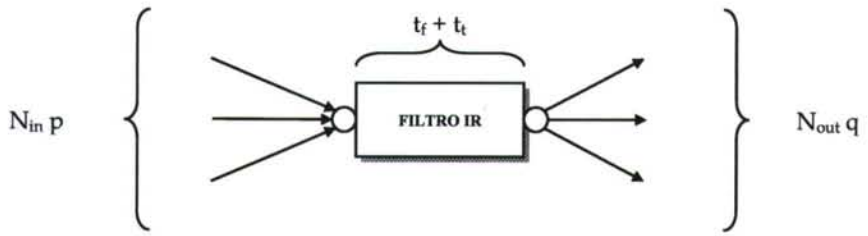

Ilustración 11: Modelo de filtro IR

El objetivo final, y la razón de ser, de estos filtros es garantizar que a su salida serán retransmitidos un subconjunto de los documentos de entrada – con sus pesos correspondientes – asegurando que entre estos se encuentran los T documentos de mayor peso de todo el conjunto de documentos de entrada para una consulta dada. El valor de T es una constante global del sistema que denominaremos *objetivo*. A lo largo de este documento T representa tanto dicho subconjunto como su cardinalidad, indistintamente y según el contexto.

El número de documentos por paquete a la entrada y a la salida del filtro no tienen por que distribuirse uniformemente, por tanto completaremos los parámetros p y q con $p(i)$ y $q(i)$ los cuales serán, respectivamente, el número de documentos en el paquete i-ésimo de entrada y de salida. Igualmente los tiempos de procesamiento y retransmisión por paquete dependerán del tamaño de los mismos, por lo cual complementamos las definiciones de t_f y t_t con $t_f(i)$ y $t_t(i)$ indicando, respectivamente, los tiempos necesarios para procesar el i-ésimo paquete de tamaño $p(i)$ y enviar hacia la siguiente etapa el paquete i-ésimo de tamaño $q(i)$.

Más formalmente hablando, el tiempo total ($t_f + t_t$) que emplea un filtro en llevar a cabo su tarea para un conjunto de paquetes de entrada p y con q paquetes de salida, es:

$$t_f + t_t = \sum_{i=1}^{p}(\alpha + \beta f(p(i))) + \sum_{i=1}^{q}(\delta + \psi)g(q(i))$$

Donde:

α	factor independiente de entrada/salida.
β	factor dependiente necesario para procesar la entrada.
δ	factor dependiente necesario para preparar la salida.
ψ	factor dependiente necesario para retransmitir la salida.
f(x), g(x)	funciones dependientes del algoritmo usado para llevar a cabo el proceso.

Con la siguiente descomposición, que será la que tendremos en cuenta en las mediciones:

$$t_f = \sum_{i=1}^{p}(\alpha + \beta f(p(i))) + \sum_{i=1}^{q}\delta g(q(i)) = \alpha p + \beta\sum_{i=1}^{p} f(p(i)) + \delta\sum_{i=1}^{q} g(q(i))$$

$$t_t = \sum_{i=1}^{q}\psi g(q(i)) = \psi\sum_{i=1}^{q} g(q(i))$$

Así mismo la eficiencia de un algoritmo de filtrado será medida calculando su complejidad, pero no la del algoritmo por si solo (esto es, para un paquete individual cualquiera) si no la del algoritmo resultante de procesar todos los paquetes correspondientes a una consulta dada.

El tamaño de un paquete – tanto de entrada como de salida – lo mediremos, si no indicamos lo contrario, según el número de documentos que contiene. La siguiente fórmula convierte su tamaño en documentos en el tamaño del paquete IP versión 4 en bytes[1]:

$$B_{ip}(i) = 20 + 4 + 8(i+1) = 8(i+4)$$

[1] Con 20 bytes para la cabecera IP, 4 para la cabecera UDP, 8 para la cabecera de la capa de aplicación (propia de nuestra implementación) y finalmente 8 bytes por cada par (*documento*, *peso*).

Según lo mostrado podemos resumir los parámetros del filtro en:

T	Número de documentos objetivo, con $n_{out} \leq T \leq n_{in}$.
N_{in}, n_{in}	Conjunto y cardinalidad de los documentos de entrada.
p	Paquetes de entrada conteniendo un subconjunto disjunto de N_{in}.
p(i)	Documentos en el paquete de entrada i-ésimo. Con $1 \leq i \leq p$.
N_{out}, n_{out}	Conjunto y cardinalidad de los documentos de salida, con $n_{out} \leq n_{in}$
q	Paquetes de salida conteniendo un subconjunto de N_{out}.
q(i)	Documentos en el paquete de salida i-ésimo. Con $1 \leq i \leq q$.
$t_f(i)$	Tiempo de procesamiento del i-ésimo paquete.
$t_t(i)$	Tiempo empleado en transmitir el i-ésimo paquete.
$B_{ip}(i)$	Tamaño en bytes del paquete IPv4 generado por *i* documentos.

A partir de estos últimos, deducimos otros parámetros que serán útiles para conocer el comportamiento del filtro:

$$r = \frac{t_f}{t_f + t_t}$$ Balance del filtrado, con $0 \leq r \leq 1$, en tanto por uno.

El balance, *r*, de un filtro nos entrega un valor en tanto por uno que nos indica cuan importante es el tiempo consumido filtrando, respecto al tiempo consumido transmitiendo los resultados. Así mismo podemos establecer la relación del tiempo de retransmisión respecto al tiempo de filtrado, la inversa de *r*, como:

$$\overline{r} = \frac{t_t}{t_f + t_t} = (1 - r)$$

3.2.1 Parámetros de calidad

La complejidad de un algoritmo mide la eficiencia que este presenta independientemente de su implementación y de la arquitectura sobre la que está construido. En el contexto que nos ocupa no solamente nos preocuparemos de la eficiencia, si no también de la eficacia de un filtro, o en un sentido más general, de su calidad, la cual mediremos y usaremos para

comparar los algoritmos propuestos entre sí, sin dejar de lado el cálculo de las complejidades, espacial y temporal, de los algoritmos.

Por tanto definimos los siguientes factores y parámetros de calidad:

$$F_f = \frac{n_{in} - n_{out}}{n_{in} - T}$$ Factor de filtrado, con $T \leq n_{out} \leq n_{in}$, $0 \leq F_f \leq 1$ (1 mejor).

$$T_f = \frac{p - q}{p - 1}$$ Factor de tráfico, con $1 \leq q \leq p$, $0 \leq T_f \leq 1$ (1 mejor).

La Real Academia Española de la lengua define la eficacia como "*la capacidad de lograr el efecto que se desea o espera*" [RAE, 01]. Siguiendo esta definición definimos una medida de la eficacia según la cual:

$$E_f = \frac{F_f + T_f}{2}$$ Eficacia del filtrado, con $0 \leq E_f \leq 1$ (1 mejor).

En formulación de la eficacia vista, esta se alcanza plenamente cuando se han alcanzado ambos objetivos por separado, esto es, el mejor factor de filtrado y el mejor factor de tráfico generado. Esta medida nos entrega el primer criterio cualitativo de comparación entre filtros IR.

Así mismo la RAE define la eficiencia como "*capacidad de disponer de alguien o de algo para conseguir un efecto determinado*" [RAE, 01]. En nuestro contexto la eficiencia nos servirá como medida de comparación entre dos filtros en virtud de sus tiempos, por tanto un criterio cuantitativo:

$$E_{ff} = \frac{1}{t_f t_t}$$

Según este criterio, a medida que aumentan los tiempo de filtrados y/o los de retransmisión, la eficiencia del filtro disminuye. Nosotros, en nuestras medidas, usaremos una modificación de esta fórmula dado que los valores resultantes para E_{ff} son muy pequeños y por tanto poco legibles. Sin disminuir su semántica, introducimos un escalado proporcional y logarítmico:

$$E_{ff} = \omega \frac{1}{\log_{10}(t_f t_t)}$$

Por un lado el resultado del producto de los tiempos es sometido a un logaritmo con el fin de conservar más información sobre valores pequeños del producto, por el otro un factor que denominaremos umbral o escala de eficiencia, ω, de valor arbitrario que nos permitirá disminuir o aumentar el valor de la medida a nuestra voluntad. Para poder comparar dos o más filtros debemos usar el mismo valor de ω.

Finalmente definimos la calidad relativa de un filtro como:

$$Q = E_f E_{ff}$$

El cual es el parámetro que usaremos para comparar un conjunto de filtros, una vez que hayamos obtenido sus parámetros de calidad.

3.2.2 Descomposición de filtros IR bajo criterios de calidad

Dado que hemos definido los parámetros del filtro IR y algunas de las medidas de calidad del mismo, construiremos ahora un conjunto de criterios que nos guiarán en el diseño de nuevos algoritmos y en el estudio de los existentes. Estos criterios establecerán familias (y variaciones de familias) de algoritmos dadas sus características

Decimos que un filtro lleva a cabo un filtrado perfecto si E_f=1. Esto es, si el número de documentos de salida es igual al conjunto de documentos objetivo T (en cuyo caso F_f es 1) y el número de paquetes de salida es 1 (un solo paquete conteniendo las T respuestas de mayor peso, el mejor factor de tráfico).

Como veremos más adelante, es posible que un filtro dado genere más de un paquete de salida. Dado que estos filtros se situarán entre los QS y el *broker* final (ver Ilustración 18) conteniendo el *interface* de usuario, es necesario que este último conozca el número exacto de paquetes que integran la respuesta parcial ya filtrada. Así, si es posible determinar *a priori* el número de paquetes de salida, q, independientemente del número de paquetes de entrada o del número de documentos tanto de entrada como de salida, decimos que este filtro es determinista. En general un filtro no determinista tendrá que entregar a la siguiente etapa un mecanismo de señalización tal que esta pueda deducir cuando empiezan o, como mínimo, acaban los documentos pertenecientes a una consulta dada. Todos los algoritmos implantados en este trabajo, sean deterministas o no, entienden y manejan la señalización principio/fin y entregan dicha señalización a su salida.

Los filtros nulos poseen F_f y T_f igual a 0, por tanto una eficacia cero. Los dispositivos pasivos de comunicaciones, como por ejemplo *routers*, pueden ser considerados filtros nulos. Incluso un segmento *Ethernet* puede ser considerado como un filtro pasivo con un

tiempo de procesamiento y retransmisión, $t_f + t_t$, igual al retardo promedio introducido dada una determinada carga de red.

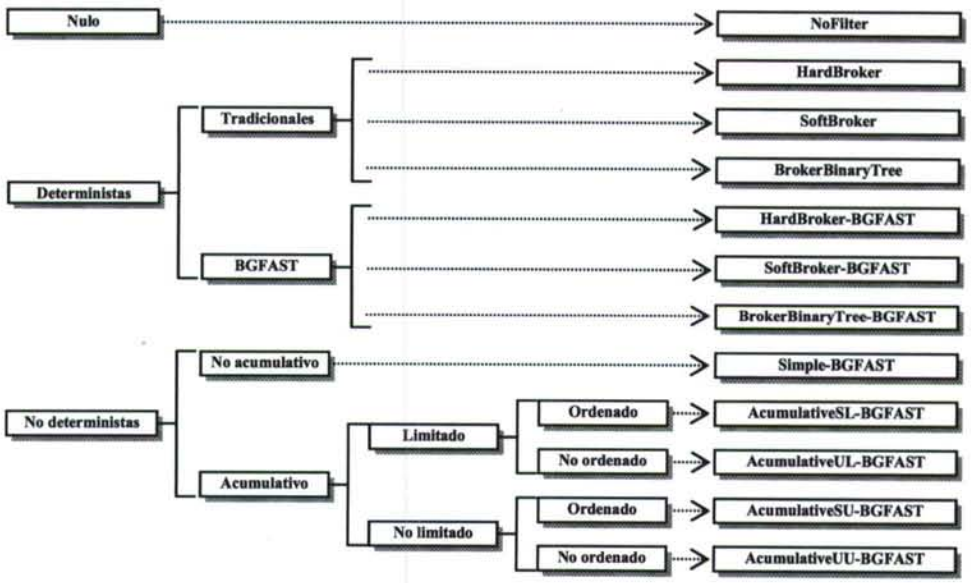

Ilustración 12: Taxonomía de los filtros implantados

Un algoritmo de filtrado puede llevar a cabo su proceso sin espera o con espera (lo etiquetaremos como *NoWait/Wait*). Los primeros emitirán tráfico tan pronto puedan sin esperar a que un nuevo paquete de la misma consulta les sea entregado. Los procesos con espera optimizan el canal de comunicaciones de salida, esperando nuevas entradas. Esta estrategia busca disminuir el factor de tráfico (T_f) a costa de aumentar el tiempo necesario para llevar a cabo el filtrado (t_f). Que un algoritmo sea *NoWait* implicará obligatoriamente que no será perfecto y no determinista, dado que no es hasta el final el momento en el que un algoritmo puede determinar el subconjunto exacto de los T mejores documentos según sus pesos.

Finalmente contemplamos los algoritmos que producen paquetes con el contenido ordenado o no ordenado (*Sort/NoSort*) principalmente bajo el criterio de mayor a menor peso en caso afirmativo. Que los resultados parciales filtrados estén o no ordenados implica por un lado que el filtro requerirá, en general, menos tiempo para llevar a cabo su tarea, pero por el contrario obligará a la siguiente etapa del proceso de IR a ordenar de alguna forma los datos o al menos a tener en cuenta todos los documentos del paquete. Básicamente esto significa disminuir el tiempo de filtrado, t_f, en una etapa a costa de aumentar posiblemente el de la siguiente. Todos los algoritmos implantados en este trabajo aceptan procesar paquetes tanto si están ordenados o no, sin embargo algunos emitirán el juego de salido no ordenado.

3.3 Filtros basados en algoritmos clásicos o tradicionales

Las medidas de calidad y otras características de los filtros IR nos han permitido clasificar este tipo de entidades en tres tipos y varias subfamilias con sus variantes. Esto nos abre la puerta a, al menos, tres tipos básicos de diseños. Diseño de algoritmos sin semántica embebida (filtros nulos), aquellos con factor de filtrado $0 < F_f < 1$, y por último aquellos con el mismo factor igual a 1. Esta subsección analiza en profundidad cuatro tipos de implantaciones objetivo de nuestro desarrollo y evaluación, bajo de denominador común de usar algoritmos conocidos hasta ahora y que podemos considerar con clásicos o tradicionales.

3.3.1 Filtro IR: NoFilter

El algoritmo más simple es aquel no hace nada y simplemente se limita a retransmitir los datos de entrada sin modificar. Aunque formalmente su semántica es nula – no modifica la información de entrada ni posee memoria – usaremos este filtro como referencia básica de las medidas de tiempos.

Para este algoritmo $n_{out} = n_{in}$ y $q = p$, entonces

$$F_f = \frac{n_{in} - n_{in}}{n_{in} - T} = 0, \qquad T_f = \frac{p - p}{p - 1} = 0, \qquad E_f = \frac{0 + 0}{2} = 0$$

Este filtro es determinista y no perfecto, su eficacia es cero y el balance (r) de este tipo de filtros será cercano a cero o en el mejor de los casos un valor cercano. Dado que la eficacia es cero, su calidad será cero independientemente de su eficiencia.

Respecto a su complejidad temporal, esta es:

$$O(\alpha p + \psi \sum_{i=1}^{p} p(i))$$

Dado que este algoritmo sólo se ocupa de reenviar sin transformar los paquetes recibidos.

3.3.2 Filtro IR: HardBroker

Aunque los agentes intermediarios, o *brokers*, los hemos contemplado hasta ahora como la última etapa de una arquitectura IR distribuida, estos pueden ser vistos también como filtros, dado que cumplen con todas sus características funcionales y semánticas.

El algoritmo consiste en, a medida que llegan paquetes, almacenar los documentos que contienen de forma secuencial sin importar su orden en una memoria lo suficientemente grande como albergar a todo el conjunto N_{in}. Una vez se ha alcanzado el final de la entrada, el proceso se limita ordenar toda esta memoria y enviar los T mejores elementos, esto es los T primeros de la lista ordenada, si se hace de mayor a menor.

En *pseudo* código, el filtro se define de forma algorítmica para cada paquete de entrada como:

```
function HardBroker(packet)
begin
   for each di en packet do
   begin
     añade di al final de total
   end

   if packet es el último
   begin
      ordenación parcial de los T primeros elementos de total
      total.count es T
      total.flags es SORT, FIRST, LAST
      transmite total hacia peer
   end
end
```

Cada elemento del paquete es añadido al final de una lista denominada *total*. A la recepción del último paquete, que forma una respuesta parcial completa, tendremos sobre dicha lista todo lo recibido hasta ese momento. En tal caso se ordena toda la lista *total*, teniendo en cuenta que sólo nos interesan los T mayores elementos. Antes de enviar hacia la siguiente etapa la lista *total* en forma de paquete, reestablecemos el tamaño de la lista y la información sobre la misma; en este caso que el paquete retransmitido es el primero y el último de una respuesta parcial y además se encuentra ordenado.

Para este tipo de filtro, los criterios de calidad cualitativos son:

$$F_f = \frac{n_{in} - T}{n_{in} - T} = 1, \qquad T_f = \frac{p-1}{p-1} = 1, \qquad E_f = \frac{1+1}{2} = 1$$

Esto es, la mejor eficacia. Respecto a la eficiencia esta dependerá claramente de los tiempos de filtrado y en menor medida de los de retransmisión. Nuestra implementación usa el algoritmo *Quicksort* [Hoa, 62] modificado con el objeto de ordenar completamente solo los T mejores elementos de la lista *total*. Al finalizar el proceso de ordenación, esta lista tendrá los documentos de mayor peso ordenados de mayor a menor en los T primeros elementos. El orden del resto de los elementos, n_{in}-T, en la lista no estará definido.

La complejidad espacial de este algoritmo es claramente $\mathbf{O}(n_{in})$, dado que necesitamos almacenar cada uno de los elementos a medida que van llegando y esperar al último antes de ordenarlos y retransmitirlos a la siguiente etapa.

Respecto a la complejidad temporal esta es:

$$O(\alpha p+\beta n_{in}+\delta n_{in}\log_2(n_{in})+\psi T)$$

Suponiendo el uso de *Quicksort* como algoritmo de ordenación[2], el tiempo para añadir elementos a la lista es proporcional al tamaño de los paquetes de entrada (β) y el tiempo de retransmisión es proporcional al tamaño del paquete de salida (ψ). A priori podemos observar como el mayor coste temporal del algoritmo lo encontramos en la preparación de la salida (δ). Hemos de recordar que la complejidad calculada es la del algoritmo sometido a todo el conjunto de paquetes de entrada y no sólo a uno en particular. Este hecho se repetirá en el resto de los algoritmos mostrados en este capítulo.

El coste inherente al añadir un elemento al final de la lista (β) es despreciable, junto con el de enviar un paquete de tamaño T (ψ), al menos en comparación con la complejidad del algoritmo de ordenación (δ) para valores de n_{in} grandes.

3.3.3 Filtro IR: SoftBroker

Podemos reducir la complejidad espacial de los *brokers* tradicionales si mantenemos una lista de tamaño fijo T ordenada constantemente y no esperar a la llegada del último paquete del resultado parcial para ordenar todo de una sola vez:

```
function SoftBroker(packet)
begin
   for each di en packet do
   begin
      inserta di en la lista ordenada total
      if la inserción ha fallado, break
   end

   if packet es el último
   begin
      total.flags es SORT, FIRST, LAST
      transmite total hacia peer
   end
end
```

Para este nuevo algoritmo, los criterios de calidad cualitativos coinciden con el anterior:

[2] *Heapsort* y *Mergesort* son algoritmo también óptimos pero en promedio con un factor constante menor que *Quicksort*.

$$F_f = \frac{n_{in} - T}{n_{in} - T} = 1, \qquad T_f = \frac{p-1}{p-1} = 1, \qquad E_f = \frac{1+1}{2} = 1$$

La razón de ser de la ruptura del bucle *for-each* cuando falla la inserción de un elemento del paquete es la de aprovechar cierta experiencia que la inserción ordenada en una lista de tamaño limitado nos entrega. Dado que dicha lista se mantiene ordenada y es limitada, y dado que los documentos dentro de un paquete se encuentran ordenados de mayor a menor, podemos deducir que si en un momento dado la inserción de un documento en dicha lista falla[3], el resto de los que forman el paquete no necesitan ser insertados, dado que podemos asegurar que también fallarán.

Así pues, la inserción en una lista ordenada finita de tamaño T, nos entrega un filtro perfecto. Sabemos además que el coste computacional asociado a insertar un elemento en una lista de estas características es $\mathbf{O}(T)$, dado que necesitaremos de media $\mathbf{O}(T/2)$ unidades de tiempo para buscar en la lista la posición[4] donde el elemento ha de ser incorporado, y también de media $\mathbf{O}(T/2)$ para abrir el hueco[5] necesario para llevar a cabo la inserción.

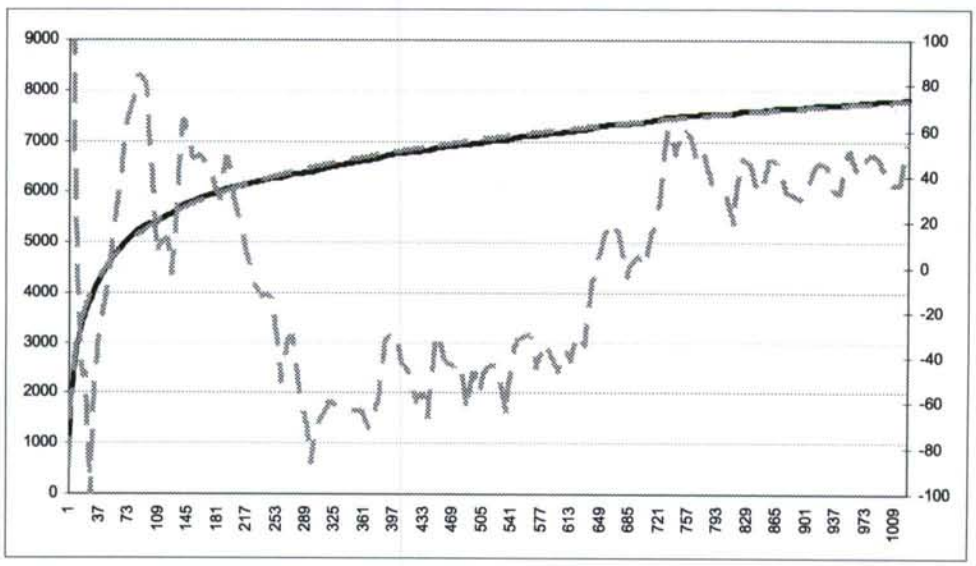

Ilustración 13: Documentos realmente insertados en la lista finita de tamaño T

[3] En este contexto, que la inserción en una lista finita ordenada falle, significa que el elemento que estamos insertando posee un peso menor que el último de la lista. Por lo tanto no debe ser añadido a la misma.

[4] La búsqueda es lineal, el objetivo es encontrar la cota superior del conjunto de elementos de la lista inferiores al criterio de búsqueda. La posibilidad de que exista en la lista un documento con el mismo peso es inferior al 0,03%.

[5] Desplazando una posición el resto de la lista y descartando – posiblemente – el elemento más pequeño.

La complejidad temporal del algoritmo dependerá del número de documentos que insertemos en dicha lista. Así pues, como a medida que este procesa paquetes es más probable que menos documentos sean insertados debemos estimar el número de documentos que realmente serán añadidos a la lista.

En la Ilustración 13 podemos observar el número acumulado de documentos realmente insertados en la lista a medida que sometemos los paquetes a este algoritmo. La línea en color negro muestra los datos obtenidos para la media de 100 muestras de 1024x1000 documentos cada uno de ellos. La línea negra sólida muestra la línea gobernada por la función[6]:

$$f(x) = 1046{,}8Ln(x) + 514{,}36$$

La línea gris de patrón intermitente – referida al eje secundario – muestra la diferencia entre las anteriores. El test chi-cuadrado de Pearson arroja un valor de r = 0,9989 para las primeras dos curvas. El error cometido tras procesar 1024x1000 documentos es de 54.

Al usar como estimación la función:

$$f(x) = 1000\sum_{i=1}^{x} \frac{1}{i^{0{,}988}}$$

Obtenemos una mejor correlación, la cual arroja un valor de r = 0,9991, así qué será esta la función que tomaremos como aproximación del valor real de documentos que procese el algoritmo. Esta función está basada en la versión no normalizada de la función de distribución Zipf (la cual es a su vez la versión discreta de la distribución de Pareto), con el valor s = 0,988.

La complejidad temporal pasa ahora a ser:

$$O(\alpha p + \beta T \sum_{i=1}^{p} p(i) \frac{1}{i^{0.988}} + \psi T)$$

Suponiendo que todos los paquetes de entrada son de tamaño T:

$$O(\alpha p + \beta T^2 \sum_{i=1}^{p} \frac{1}{i^{0.988}} + \psi T) = O(\alpha p + \beta T^2 H_{p,0.988} + \psi T)$$

[6] Línea de tendencia logarítmica generada por la herramienta usada.

Siendo el valor de la serie armónica $H_{p,\,0.988}$ aproximadamente 7.80478 para $p = 1024$, por tanto:

$$O(\alpha 1024+\beta 7.80478T^2+\psi T) \approx O(\alpha 1024+\beta 7805T+\psi T)$$

Y el número de documentos efectivamente insertados en la lista: $TH_{p,0.988}$

Respecto a la complejidad espacial, efectivamente, la hemos reducido a $\mathbf{O}(T)$, en comparación al anterior el ahorro puede ser considerable; en nuestra implantación 8MB frente a alrededor de 8KB por cada consulta simultánea.

3.3.4 Filtro IR: BrokerBinaryTree

Finalmente en esta subsección presentamos un filtro implantado usando árboles binarios. Semánticamente es igual que el anterior, pero en vez de usar una lista que se conserva ordenada (con un coste) se inserta en un árbol binario (con un coste diferente, igual o menor al anterior) que, consecuentemente, quizá permita un recorrido de mayor a menor, pero con la necesidad de preparar la salida.

El algoritmo es idéntico al anterior, salvo que la inserción se lleva a cabo sobre una estructura en forma de árbol y finalmente, antes de enviar el paquete, se recorre el árbol y se vuelca sobre un paquete vacío los T primeros, mejores, documentos.

```
function BrokerBinaryTree(packet)
begin
   for each di en packet do
   begin
      inserta di en el árbol tree
      if la inserción ha fallado, break
   end

   if packet es el último
   begin
      vuelca tree sobre lista
      lista.flags es SORT, FIRST, LAST
      transmite lista hacia peer
   end
end
```

La complejidad espacial de este algoritmo es ligeramente superior a la de su antecesor, dado que es un poco más costoso, en términos de memoria, almacenar N elementos en una lista (*array*) que en forma de árbol, mediante nodos y puntero a nodos.

Nuestra implantación modifica ligeramente el procedimiento de inserción, de forma que esta garantiza que el árbol contendrá T elementos como máximo. En caso de que una nueva inserción situé T+1 elementos en el árbol, el elemento más pequeño es eliminado inmediatamente.

La complejidad temporal es básicamente igual que para el caso anterior, pero con un coste inferior asociado a insertar y eliminar un documento en el árbol, y la aparición del factor dependiente (δ) cuyo significado es volcar los elementos del árbol en la lista.

$$O(\alpha p+\beta 2\log_2(T)TH_{p,0.988}+(\delta+\psi)T)$$

Este algoritmo posee los siguientes parámetros de calidad:

$$F_f=\frac{n_{in}-T}{n_{in}-T}=1,\qquad T_f=\frac{p-1}{p-1}=1,\qquad E_f=\frac{1+1}{2}=1$$

Igual que los dos anteriores, dado que es determinista y perfecto.

La implementación usando árboles binarios sin control de balanceo tiene un grave inconveniente. Pensemos que a medida que introducimos elementos en el árbol llega un momento[7] en que debemos ir paralelamente eliminando los documentos de menor peso del mismo. A medida que procesamos paquetes de esta forma obtenemos un árbol degenerado, con tendencia al desequilibrio de las hojas hacia el lado izquierdo (mantenemos el árbol ordenado de mayor a menor). Cuantos más elementos introduzcamos más grande es el desequilibrio y el árbol tiende a parecerse a una lista. En esta situación la complejidad temporal al usar un árbol binario sin control de balanceo es idéntica al del algoritmo *SoftBroker* para el factor β, esto es el logaritmo en base 2 de T se convierte en T, por tanto nos encontramos en este caso que su complejidad temporal sería:

$$O(\alpha p+\beta T^2H_{p,0.988}+(\delta+\psi)T)$$

Para solucionar esta degradación del rendimiento al usar árboles binarios sin control de balanceo hemos implementado, también, el mismo algoritmo usando árboles binarios AVL, en donde después de cada inserción o borrado se rebalancean los subárboles derecho e izquierdo de cada nodo implicado. Esta estrategia permite que la búsqueda en el árbol (en cada inserción, borrado o preparación) sea óptima (logarítmica). Pero para ello deberemos de pagar un coste asociado a las rutinas de rotación de los subárboles. Las medidas reales indicarán finalmente si el coste vale la pena.

[7] A partir del documento número T.

3.4 Algoritmo GFAT (Grooved Filter with Adaptative Threshold)

Hemos visto hasta ahora implantaciones de filtros perfectos, aquellos que entregan a su salida un único paquete conteniendo exactamente el subconjunto T de N_{in} formado por los documentos de mayor peso para todos los de entrada. Si observamos sus complejidades temporales podemos ver que gran parte de su valor depende del número de documentos n_{in} a procesar - por medio de $p(i)$. Si reducimos este número, podremos reducir el tiempo t_f.

Para tal fin hemos desarrollado un algoritmo con el objetivo de ser usado para construir filtros más complejos. Este algoritmo es en sí un filtro no perfecto y no determinista. Como tal, este algoritmo garantiza que el conjunto de resultados objetivo T forma parte de su N_{out} y además como veremos su tiempo de ejecución en sumamente bajo.

Hemos denominado a este algoritmo filtrado ranurado con umbral adaptativo (GFAT, *Grooved Filtered with Adaptative Threshold*) y la idea básica detrás de este proceso consiste en contar el número de pesos que van pasando por encima de un umbral. Este valor se inicia con el peso menor y se va incrementando a medida que más documentos van pasando. El espacio de pesos (valores que puede tomar el peso de un documento) se encuentra ranurado, esto es, dividido en secciones iguales, con el objeto de disminuir la complejidad espacial del algoritmo, como veremos en los siguientes párrafos.

Existe una relación con el algoritmo visto en *SoftBroker*: este estaba basado en recordar (*almacenar* en una lista de tamaño fijo) los mejores documentos que hayan pasado hasta el momento. Nuestro filtro está basado en *contar* el número de los mejores documentos hasta el momento. Ambas estrategias son parecidas en sus conceptos básicos, pero difieren significativamente en implementación y resultados. El primero forma un filtro perfecto, el segundo no.

Como hemos indicado previamente las respuestas parciales de los QS, R_p, es un conjunto ordenado de pares (*id*, *peso*) donde *id* es el identificador del documento y *peso* es el valor relativo que el documento posee dentro de la consulta de la que la respuesta parcial forma parte. La relación de equivalencia de R_p es tal que los pares se consideran ordenados por el elemento *peso* del par. El elemento del par *id* pertenece al conjunto de los números naturales.

Sea W el conjunto ordenado definido como:

$$W = \left\{ x \in N^* / w_{low} \leq x \leq w_{high} \right\}$$

y

$$|W| = w_{high} - w_{low} + 1$$

El elemento del par *peso* pertenece a W. Las cotas w_{low} y w_{high} son respectivamente el mínimo y el máximo del conjunto W. En nuestro desarrollo w_{low} es 0 y w_{high} es 2^{32}-1. Por tanto |W| es 2^{32} (un entero largo sin signo de 32 bits).

Sea W_p una colección no vacía de conjuntos tal que:

$$Wp = \{r_i \subset W / 0 \le i \le c-1, \cup r_i = W, \cap r_i = 0, \forall x \in r_i, y \in r_{i+1} \Rightarrow x < y\}$$

Vemos claramente que W_p es una partición de W en elementos r_i que denominaremos *ranuras*. Así mismo la cardinalidad de W_p es c, esto es, el número de ranuras en que dividimos el conjunto W. Los subconjuntos que forman la colección de conjuntos W_p están sujetas a una relación de orden inducida por sus elementos, de forma que podemos afirmar $r_0 < r_2 < \ldots < r_{c-2} < r_{c-1}$. El concepto de cardinalidad de r_i, esto es, el número de elementos que contiene, lo asociaremos con la expresión *anchura de la ranura* y lo denotaremos por w en el caso de que sean todas iguales[8].

Según esto, el menor ancho de ranura es 1, el mayor es $|W|$. La relación entre w, el ancho de la ranura y c es la siguiente:

$$c = \frac{|W|}{w}$$

Dando por supuesto que el ancho de las ranuras es idéntico para todas ellas dentro de W_p, como será el caso que nos ocupa en este documento.

Cada ranura posee un mínimo y un máximo que denotaremos por:

$$\lfloor r_i \rfloor = x \in r_i / \neg\exists y \in r_i, y \neq x / y < x$$

$$\lceil r_i \rceil = x \in r_i / \neg\exists y \in r_i, y \neq x / y > x$$

Así, es trivial que:

$$\lceil r_i \rceil + 1 = \lfloor r_{i+1} \rfloor$$

[8] En este trabajo supondremos que en efecto todas las ranuras son de idéntico tamaño.

$$\lceil r_{i-1} \rceil = \lfloor r_i \rfloor - 1$$

También podemos demostrar fácilmente que:

$$\forall x \in W \Rightarrow (\exists r_i \in W_p / x \in r_i)$$

Los pesos en W poseen una función de densidad $f_W(x)$ que nos indica la probabilidad del peso x perteneciente al conjunto W. En el contexto que nos ocupa los pesos se distribuyen con una frecuencia uniforme así pues:

$$f_W(x) = \frac{1}{|W|} = \frac{1}{W_{high} - W_{low} + 1}$$

Siendo r una ranura de W_p sobre W tenemos su función de densidad:

$$f_{W_p}(r) = \sum_{i=\lfloor r \rfloor}^{\lceil r \rceil} f_W(i) = \sum_{i=\lfloor r \rfloor}^{\lceil r \rceil} \frac{1}{|W|} = w \frac{1}{|W|}$$

El filtro que estamos elaborando posee el parámetro T, constante global mayor que cero que indica el número mínimo de documentos que deben pasar el filtro de entre los documentos de mayor peso. El número de documentos efectivamente aceptados por el filtro dependerá de este valor y, por supuesto, del número de documentos finalmente sometidos al filtrado.

Para una consulta dada, el filtro recibirá un número determinado de respuestas parciales de varios QS. El filtro mantiene un *array* o lista de enteros sin signo $C[i]$, $1 \leq i \leq c$, por consulta, con una entrada para cada elemento del conjunto W_p, de forma que $C[i]$ contiene el numero de documentos que han pasado el filtro con peso $x \in r_i$.

También, por consulta, el filtro mantiene un valor umbral f_{th} ($0 \leq f_{th} \leq c\text{-}1$) que indica el valor actual del filtro como un índice de la lista C. Los documentos con un peso inferior a este valor no pasarán el filtro. En general:

$$f_{th} \Rightarrow \lfloor r_{f_{th}} \rfloor$$

El cual es el valor *real* del filtro (valor de peso mínimo en la ranura f_{th}-ésima). Hablaremos de f_{th} como el umbral índice de C y F_{th} como el umbral *real* del filtro (el mínimo de la ranura a la que hace referencia el índice). La conversión de uno a otro es trivial[9].

Además, sea:

$$S_{th} = \sum_{i=f_{th}}^{c} C[i]$$

Esto es, la suma de los documentos ya filtrados con pesos igual o superiores a f_{th}. Como veremos en el algoritmo, su valor puede ser calculado y mantenido sin necesidad de incorporar un bucle a tal efecto.

El valor inicial de f_{th} es 0, e inicialmente C contiene una lista de ceros.

Siendo GFAT(T, w) un filtro con parámetros T, objetivo a alcanzar, y w el ancho de la ranura usada, el algoritmo que determina si un documento ha de ser descartado por el filtro dado su peso, el valor índice del filtro f_{th}, el valor acumulado S_{th} y la lista C de contadores, es:

```
boolean GFAT::filter(peso)
begin
        if peso/w mayor o igual que fth then
                c[peso/w] ← c[peso/w] + 1
                Sth ← Sth + 1
                while Sth - C[fth] mayor o igual que T do
                        Sth ← Sth - C[fth]
                        fth ← fth + 1
                end while
                retorna falso
        else
                retorna cierto
        end if
end
```

Podemos demostrar trivialmente que cuando f_{th} alcanza el valor c-1, su máximo valor permitido, la expresión S_{th} - $C[f_{th}]$ es cero; y dado que T es siempre mayor que cero, f_{th} nunca alcanza c. El bucle que condiciona esta expresión asegura un rápido crecimiento de f_{th} en el caso de existir índices de C con un contenido igual a cero, esto es, cuando no hayan pasado documentos con peso entre los límites de la ranura que le corresponde a dicho índice.

[9] Multiplicar y dividir respectivamente por el ancho de la ranura, suponiendo estas iguales. Si elegimos un ancho de ranura potencia de 2 las operaciones se convierten en desplazamientos a la derecha (multiplicar) y a la izquierda (dividir) del exponente. Tal es el caso de nuestro desarrollo, donde el ancho de ranura elegido es 2^{23} (8M).

La complejidad espacial de este algoritmo es **O**(c) puesto que necesita almacenar un contador por cada elemento del conjunto W_p. Por el otro lado la complejidad temporal es también **O**(c).

3.4.1 Selección del ancho de ranura óptimo

La complejidad del algoritmo GFAT, tanto espacial como temporal, dependen formalmente del número de ranuras, o lo que es lo mismo, es inversamente proporcional al ancho de la ranura seleccionada. La eficacia del proceso, por medio del factor de filtrado, también depende de este valor, sobre todo para conjuntos de entrada N_{in} grandes.

Dado un filtro GFAT(T, w), supongamos que el umbral al cabo de n documentos se ha situado a su máximo valor (c - 1); a partir de este instante y hasta el final del conjunto de entrada el filtro dejará pasar cualquier documento con peso mayor o igual a (c-1)*w. La media del número de documentos que pasarán el filtro a partir del instante n y hasta el final (n_{in}) es:

$$(n_{in} - n)\frac{w}{|W|}$$

Esta expresión nos muestra que a medida que hacemos más grande el ancho de la ranura (w) será mayor el número de documentos que pasen el filtro una vez que este ha alcanzado el máximo permitido del valor del índice del umbral. Si w es 1, entonces este valor medio se hace mínimo.

W	c
1K	4M
64K	64K
128K	32K
256K	16K
512K	8K
1M	4K
2M	2K
4M	1024
8M	512
16M	256
32M	128
64M	64
128M	32
256M	16
512M	8
1024M	4

Tabla 1: Relación entre los parámetros de los experimentos

Los valores empleados en los experimentos nos indican que n=550T para un conjunto n_{in} de 1024x1000 documentos. Si trabajamos con un ancho de ranura (w) igual a 2^{23} (8M) y |W| igual a 2^{32}, tenemos que el número de documentos que pasarán el filtro durante la última

etapa es de aproximadamente 926 documentos. Con los mismos datos, si el ancho de ranura es la unidad, serán 0.001 el número de documentos aceptados.

Para demostrar gráficamente que la eficacia del filtro depende del ancho de la ranura, hemos sometido un mismo juego de prueba a diferentes filtros, donde cada uno de ellos se diferenciaba de los otros exclusivamente en el valor de *w*.

Los diferentes filtros han recibido nombres que indican el ancho de la ranura que usan, por ejemplo el denominado 32M es aquel filtro tal que GFAT(*T*=1000, *w*=32M). Así, desde 1K a 1024M sus relaciones con el número de ranuras se muestran en la Tabla 1.

Tanto en la Ilustración 14 como en la Ilustración 15 la leyenda de los gráficos muestran los nombres y las curvas de los filtros según un código de color en tonos de grises. Cuanto mejor trabaje un filtro en el contexto de la ilustración (eficacia, eficiencia) mas oscura será la curva que la representa.

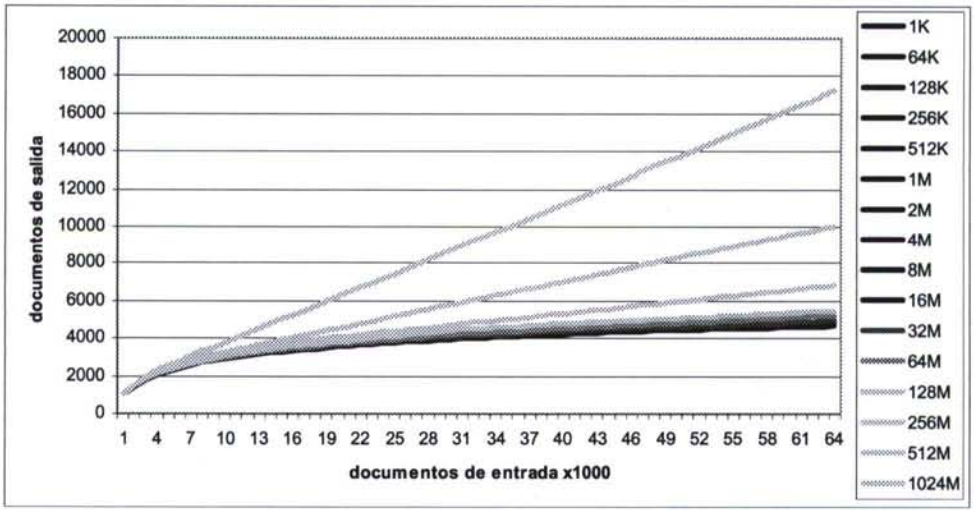

Ilustración 14: Relación entre documentos de entrada y salida según *w* (eficacia)

En la Ilustración 14 podemos observar como el peor resultado lo tenemos en el valor de *w* igual a 1024M, y generalizando, a menor valor del ancho de la ranura, mejor filtrado. Tras 64 paquetes con un millar de documentos cada uno de ellos, los filtros que obtuvieron mejores resultados fueron los encuadrados entre los valores 1K y 16M, si que hubiera grandes diferencias entre ellos (142 documentos de diferencia en total).

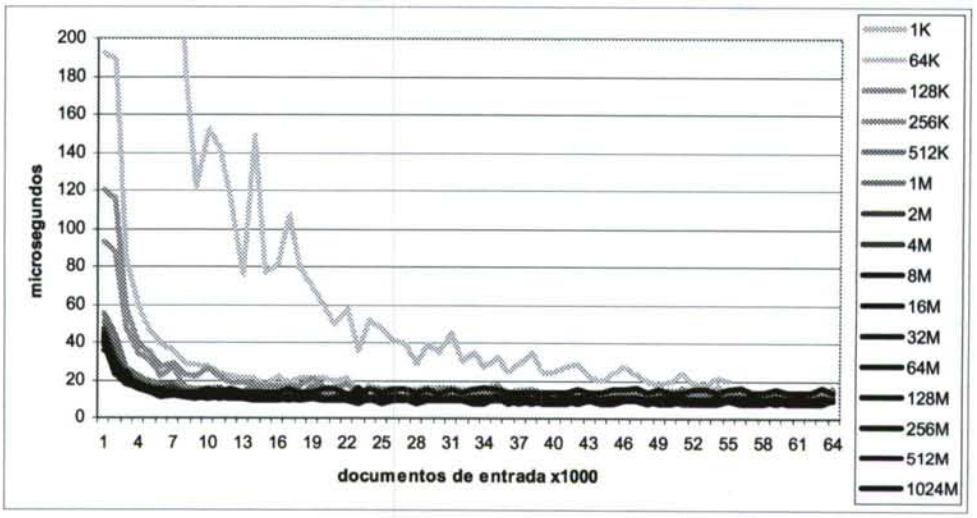

Ilustración 15: Tiempo empleado filtrando para diferentes valores de ancho de ranura (eficiencia).

Podemos observar que a medida que aumentamos el número de ranuras – esto es, disminuimos el tamaño de la ranura –, el filtrado se hace cada vez mejor. Respecto a los tiempos empleados en llevar a cabo e filtrado, la Ilustración 15 muestra como para valores de w bajos el proceso consumió mas ciclos de CPU. En general veremos más adelante que cuanto más documentos descarte un filtro, menos tiempo empleará en procesar un paquete ordenado de mayor a menor.

En la Tabla 2 vemos las medidas tomadas para estos filtros. Podemos ver como el mejor factor de filtrado (F_f) se da para valores de w pequeños, mientas que para el tiempo empleado los mejores valores se encuentran entre 2M y 256M (E_{ff}). Finalmente el parámetro de calidad (Q) establece los mejores valores entre 8M y 64M.

Estas medidas nos han servido en nuestros experimentos para tomar la decisión de qué valor de ancho de ranura tomar. Nuestra elección ha sido 8M, dado que reúne un buen equilibrio entre eficacia y eficiencia y de entre los destacados por su factor de filtrado (F_f) es el que menos memoria consume (el conjunto C tan sólo posee 512 elementos).

3.4.2 Mediciones y estudio analítico.

La calidad del filtrado depende del orden en que los pesos son presentados al algoritmo. El mejor caso es aquel donde los documentos se encuentran ordenados de mayor a menor y el peor es, claramente, de menor a mayor. En el primer caso, si el ancho de la ranura es 1, el filtro será perfecto, el conjunto de documentos que este algoritmo permite pasar en el mejor de los casos coincide con T, el objetivo. En nuestro contexto los documentos son procesados en paquetes y dentro de cada uno de ellos los pesos se encuentran ordenados de

mayor a menor. Este hecho ayuda bastante a mejorar el rendimiento del algoritmo dado que, aunque los T primeros documentos que forman el primer paquete pasan siempre el filtro, el algoritmo aprende muy rápidamente y para los siguientes documentos la capacidad de filtrar se incrementa hasta estabilizarse (ver Ilustración 16)[10]. En dicha ilustración podemos observar como el valor del umbral f_{th} parte del valor cero y se incrementa rápidamente al principio para luego estabilizarse[11] y crecer monótonamente con una pendiente cercana o igual a cero. Por otra parte los valores que s_{th} toma forman una gráfica tipo "*dientes de sierra*"; esto es lógico, dado que cada vez que f_{th} modifica su valor – incrementándose – s_{th} es decrementado para luego volver a crecer linealmente.

	Nout	tf	Ff	Ef	Eff	Q
1K	4672	123737	0,942	0,471	0,196	0,0925
64K	4672	1578	0,942	0,471	0,313	0,1472
128K	4672	1285	0,942	0,471	0,322	0,1515
256K	4672	995	0,942	0,471	0,334	0,1571
512K	4676	808	0,942	0,471	0,344	0,1619
1M	4678	789	0,942	0,471	0,345	0,1625
2M	4681	732	0,942	0,471	0,349	0,1644
4M	4697	718	0,941	0,471	0,350	0,1648
8M	4739	702	0,941	0,470	0,351	0,1652
16M	4814	689	0,939	0,470	0,352	0,1655
32M	4947	660	0,937	0,469	0,355	0,1662
64M	5245	659	0,933	0,466	0,355	0,1654
128M	5504	676	0,929	0,464	0,353	0,1641
256M	6837	737	0,907	0,454	0,349	0,1582
512M	10036	745	0,857	0,428	0,348	0,1491
1024M	17230	967	0,742	0,371	0,335	0,1243

Tabla 2: medidas y parámetros de calidad de diversos anchos de ranura

En la ilustración anteriormente citada, el valor de f_{th} es el valor del índice del *array* C, mientras que el valor del umbral real (F_{th}) es el resultado de multiplicar f_{th} por el ancho de la ranura w. Si transformamos la gráfica de índices a valores reales del filtro, la pendiente se conserva. Sea esta función:

$F_{th}(x) = f(x)w$, con $f(x)$ la función f_{th} representada en la Ilustración 16.

Podemos observar que la probabilidad de que un documento sea aceptado o no depende del valor del umbral en el instante en que este es procesado.

[10] Esta ilustración sólo muestra el intervalo entre los valores 1 y 100, que representan el número de paquetes procesados o los miles de documentos que el algoritmo procesa dado que trabajamos con 1000 documentos por paquete. El experimento real consta de 1024 paquetes en total.

[11] En la ilustración el valor de f_{th} es el índice del array C, no el valor del umbral real.

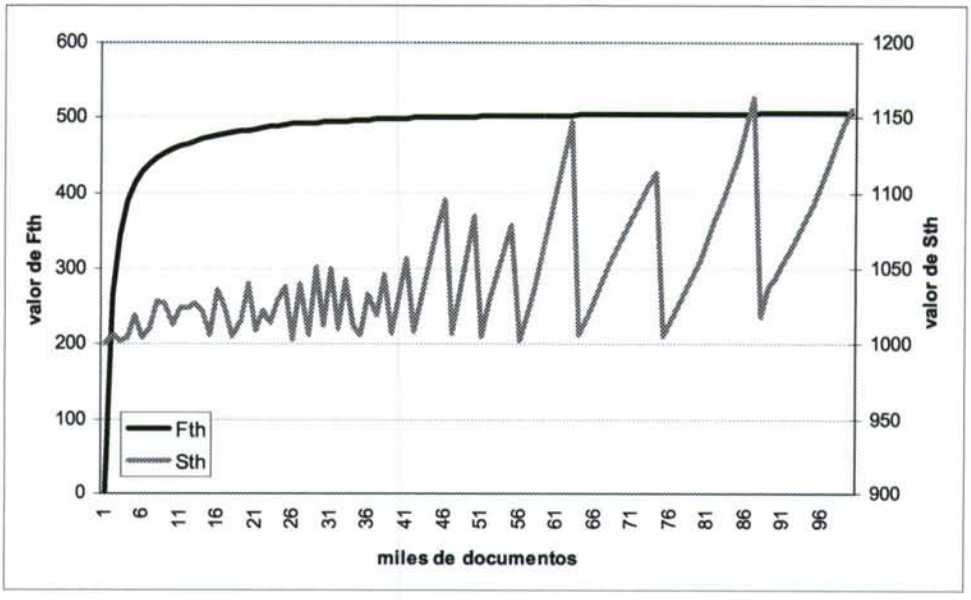

Ilustración 16: Evolución de f_{th} y s_{th} para los 100 primeros paquetes

Por tanto la probabilidad de que el documento en el paquete i-ésimo sea rechazado es:

$$\overline{p}(i) = \frac{F_{th}(i)}{|W|}$$

Evidentemente la posibilidad de que este mismo documento sea aceptado es:

$$p(i) = \frac{|W| - F_{th}(i)}{|W|} = 1 - \frac{F_{th}(i)}{|W|} = 1 - \overline{p}(i)$$

En general la probabilidad de que cualquier documento haya sido rechazado habiendo procesado hasta el momento *n* paquetes es:

$$\overline{P}(i \le n) = \sum_{k=1}^{n} \frac{F_{th}(k)}{n|W|}$$

Y la de ser aceptado:

$$P(i \le n) = 1 - \sum_{k=1}^{n} \frac{F_{th}(k)}{n|W|}$$

Por tanto, de media, el número de documentos aceptados después de haber procesado *n* paquetes, con *T* documentos cada uno de ellos, es:

$$TnP(i \le n) = Tn(1 - \sum_{k=1}^{n} \frac{F_{th}(k)}{n|W|}) = Tn - T\sum_{k=1}^{n} \frac{F_{th}(k)}{|W|}$$

Siendo el número medio de documentos rechazados:

$$Tn\bar{P}(i \le n) = Tn\sum_{k=1}^{n} \frac{F_{th}(k)}{n|W|} = T\sum_{k=1}^{n} \frac{F_{th}(k)}{|W|}$$

Calcular $F_{th}(x)$ es difícil, dado que depende del orden en que los elementos se presentan al filtro. Sin embargo una estimación analítica, suponiendo que a la entrada los documentos se presentan desordenados y sus frecuencias se encuentran uniformemente distribuidas es:

$$F_{th}(x) = w\left\lfloor c(1 - \frac{1}{x}) \right\rfloor$$

Donde w es el ancho de ranura y c es número de estas en las que dividimos el conjunto W. Esta función nos entrega el valor real del umbral después de haber procesado el paquete x, con $1 \le x$. El test chi-cuadrado de Pearson arroja un valor[12] de $r = 0{,}99991$ para $c = 512$.

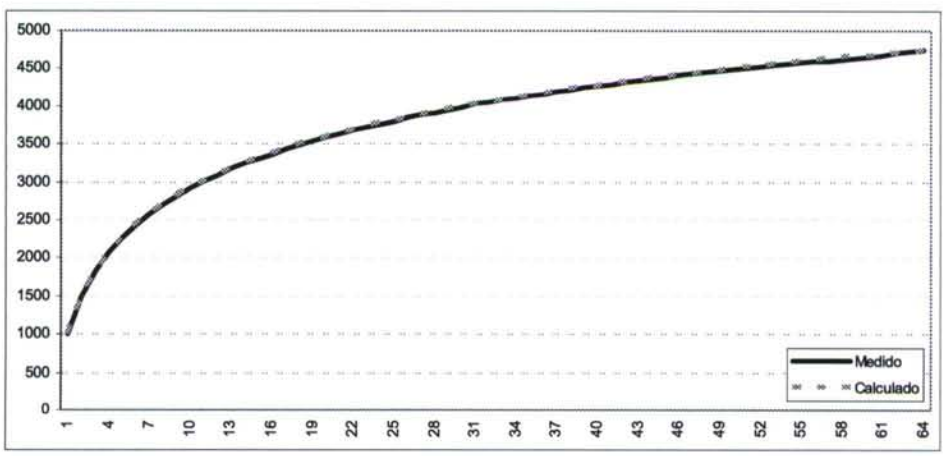

Ilustración 17: Relación entre documentos de entrada (en miles) y de salida acumulados

Sustituyendo, la estimación del número de documentos aceptados es:

[12] Para valores menores de c, el mismo test arroja valores de $r = 1$.

$$TnP(i \le n) = Tn - T\sum_{k=1}^{n} \frac{w\left\lfloor c(1-\frac{1}{k}) \right\rfloor}{|W|} \ge Tn - T\sum_{k=1}^{n} \frac{wc(1-\frac{1}{k})}{|W|} = Tn - T\sum_{k=1}^{n}(1-\frac{1}{k}) =$$

$$Tn - T(n - \sum_{k=1}^{n}\frac{1}{k}) = Tn - Tn + T\sum_{k=1}^{n}\frac{1}{k} = T\sum_{k=1}^{n}\frac{1}{k} = TH_n$$

Finalmente el número aproximado de documentos aceptados después de haber procesado n paquetes de T documentos cada uno de ellos es proporcional a la serie armónica de primer orden de n. El test chi-cuadrado de Pearson entrega un valor $r = 1$ para las curvas mostradas en la Ilustración 17. El cálculo arroja un resultado de 4744 documentos aceptados para n = 64, frente al valor experimental realmente obtenido de 4770 documentos[13].

Podemos calcular aproximadamente el valor de $TnP(\, i \le n\,)$ de la siguiente manera:

$$TnP(i \le n) \approx T(Ln(n) + \gamma)$$

Con γ la constante de Euler-Mascheroni (aproximadamente 0,577215665).

Hemos de remarcar las diferencias entre el número de documentos que GFAT filtra

$$\mathrm{TH}_n$$

Frente al número de documentos que en el filtro *SoftBroker* inserta finalmente el la lista de tamaño T, calculado previamente como

$$\mathrm{TH}_{n,0.988}$$

Esto implica que la inserción en una lista finita de T posiciones filtra menos elementos que GFAT (experimentalmente la diferencia para n = 1024 es de aproximadamente 296 documentos). Este hecho se hará significativo cuando combinemos ambas estrategias en un solo filtro.

[13] Recordemos que hemos calculado una cota inferior.

3.5 Algoritmo GFAST (Grooved Filter with Adaptative and Scaled Threshold)

Como hemos visto en la sección anterior, el ancho de la ranura afecta a la eficacia del filtrado. El valor ideal para w es la unidad sin embargo en dicha situación el tamaño de la lista C se hace lo suficientemente grande como para colapsar la memoria de un sistema[14]. La elección de un ancho de ranura con valor 8M es un compromiso entre la eficacia y la complejidad espacial del algoritmo. Esta subsección está dedicada a mejorar la capacidad de filtrado sin modificar el ancho de la ranura y conservando la complejidad tanto espacial como temporal de algoritmo.

Dado que es el umbral el último responsable del filtrado es lógico pensar en estrategias que manipulen el índice f_{th} con el objetivo de influir en la eficacia del filtrado. Veamos alguna de estas estrategias.

Observando el código del algoritmo podemos deducir que existe un valor del índice f_{th} para el cual el filtrado es óptimo (dentro de los límites que impone el ancho de ranura establecido). A medida que vamos sometiendo pesos al algoritmo, f_{th} va cambiando, incrementándose, buscando siempre un valor tal que asegure que hayan pasado menos de T documentos por encima de f_{th} +1 y más o igual de T documentos por encima de f_{th} (recordemos, esta peculiaridad está controlada por s_{th} y C). Decimos que el valor de f_{th} es óptimo (o cercano a él) por que en dicho punto este tiende a estabilizarse y el hecho de que pasen más documentos apenas cambia su valor.

Bajo este prisma podemos decir que el valor menos eficaz de f_{th} es precisamente su valor inicial, 0. Pensemos que a la llegada de los T primeros documentos estos *siempre* pasarán el filtro, precisamente por que el valor inicial de f_{th} es cero y por que hasta el documento T+1 no habrá por encima de f_{th} más de T documentos. La única situación completamente favorable es que los T primeros documentos sean los mejores de los que pasarán hasta el final del conjunto de entrada. Lo normal es que, al distribuirse aleatoriamente, una gran parte de los primeros T documentos no formen parte del objetivo.

Por tanto la primera propuesta de mejora es cambiar el valor inicial del índice f_{th} a, pongamos por ejemplo, su valor medio $c/2$. A priori podemos razonar que esta estrategia funciona bien, dado que tanto como el 44% de los documentos de entrada serán eliminados gracias sólo a esta mejora. En concreto esta estrategia dejará pasar aproximadamente ½(*N*-

[14] Para un valor de w igual a la unidad, el tamaño de C es $2^{32}*8$ bytes, esto es 32GB; y esto sólo para una consulta concurrente.

T)+T documentos, siendo N el número de documentos que pasarían en circunstancias normales (con f_{th} inicializado a cero). En general podemos escoger cualquier valor inicial n_0 tal que podamos garantizar que cualquier documento perteneciente al objetivo, su peso será mayor o igual a n_0*w, lo que supondría que dejaría pasar la siguiente cantidad de documentos:

$$(1-\frac{n_0}{c-1})(N-T)+T$$

Siendo N el número de documentos que realmente pasarían con f_{th} inicializado a cero, y n_0 el valor inicial de f_{th}. Si n_0 es cero, la expresión se hace igual a N, si n_0 es igual a c-1 (su máximo valor) entonces la expresión se evalúa a T.

Esta aproximación tiene dos inconvenientes graves. Por un lado el hecho de conocer a priori el mínimo del conjunto objetivo, algo que no consideramos posible en nuestro contexto. Por el otro, en el caso de apostar por una cota inferior de T, el hecho de que ya no tendríamos garantizado que en N_{out} estuviera el conjunto de los T mejores documentos de N_{in}, si no una fracción de estos (más pequeña cuanto más alta fuera el valor de la apuesta). Realmente podría darse el caso de que no hubiera documentos de salida.

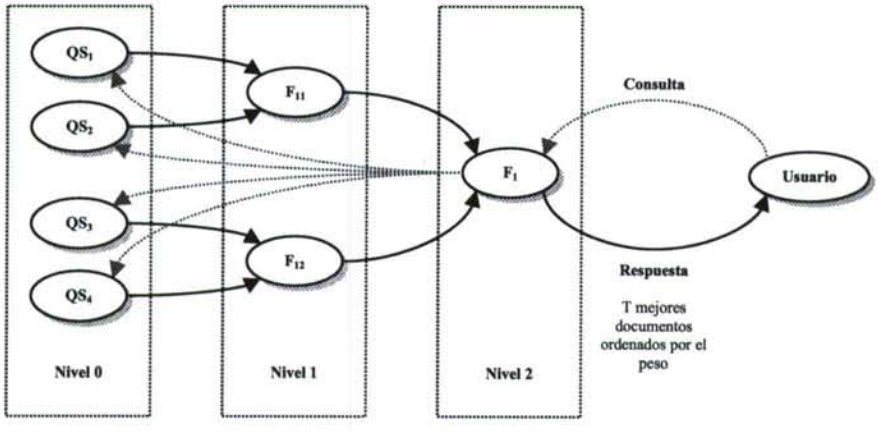

Ilustración 18. Arquitectura IR filtrada multinivel

La estrategia anterior no es aceptable, dado que funcionaría o no dependiendo de la entrada. Sin embargo nos da pié a introducir otra estrategia, con el mismo objetivo, pero más robusta. Se trata igualmente de reducir el conjunto de salida a base de sacrificar algunos de los elementos menores de T. Si pensamos en un filtro en solitario podría parecer poco útil, pero si pensamos que la salida de varios filtros en paralelo pueden alimentar a otro filtro en un segundo nivel de filtrado (ver Ilustración 18), entonces es un caso muy poco probable el que se elimine algún documento importante si reducimos el objetivo T de los filtros del

primer nivel. En [Cacheda, 05] los autores reflejan este hecho; permitiendo a los QS (1024 en total) recolectar únicamente p documentos por QS ($p < T$), sus resultados demuestran que si p=15, por ejemplo, la probabilidad de que finalmente tengamos la pérdida de algún documentos dentro del objetivo (pérdida de precisión) es de menos de $3*10^{-10}$. Cuantos menos QS tengamos en cuenta mayor será la probabilidad.

Por tanto es útil incorporar al filtro un parámetro tal que reduzca la cardinalidad del conjunto objetivo de salida. En concreto en nuestra implantación lo incluiremos en el código como el parámetro s (*scale*) de la siguiente manera.

```
boolean GFAST::filter(peso)
begin
        if peso mayor o igual que fth then
                c[peso/w] ← c[peso/w] + s
                Sth ← Sth + s
                while Sth - C[fth] mayor o igual que T do
                        Sth ← Sth - C[fth]
                        fth ← fth + 1
                end while
                retorna falso
        else
                retorna cierto
        end if
end
```

El único cambio es la constante que se añade a $C[i]$ y a S_{th}, la cual pasa de ser el parámetro del filtro s. Dado que ahora ambas variables se incrementan en s unidades, para que f_{th} cambie será necesario que hayan pasado menos de T/s documentos por encima de f_{th} +1 y más o igual de T/s documentos por encima de f_{th}. Por tanto el filtro GFAST(T, w, s) con objetivo T, ancho de ranura w y escala s, garantiza que dejará pasar una cantidad de documentos tal que entre estos estarán los T/s mejores de la entrada según sus pesos.

Así se tiene la siguiente equivalencia: GFAST(T, w, 1) = GFAT(T, w).

Con este cambio, para $s > 1$, el umbral crece s veces más rápido que con $s = 1$. Esto afecta a la curva vista en la Ilustración 16, *acelerando* el primer tramo (el tramo con la pendiente superior a cero). El cálculo del número de documentos que el filtro acepta efectivamente ahora es:

$$TnP_s(i \le n) = Tn - T\sum_{k=1}^{n} \frac{w\left\lfloor c(1-\frac{1}{sk})\right\rfloor}{|W|} \ge Tn - T\sum_{k=1}^{n} \frac{wc(1-\frac{1}{sk})}{|W|} = Tn - T\sum_{k=1}^{n}(1-\frac{1}{sk}) =$$

$$Tn - T(n - \sum_{k=1}^{n}\frac{1}{sk}) = Tn - Tn + \frac{T}{s}\sum_{k=1}^{n}\frac{1}{k} = T\sum_{k=1}^{n}\frac{1}{sk} = \frac{T}{s}H_n \approx \frac{T}{s}(Ln(n)+\gamma)$$

Recordemos, con s ≥ 1.

El uso del parámetro *s* puede ser constante para el filtro (no cambia durante su uso) o variable a lo largo del tiempo de vida de este (a medida que se avanza en el filtrado, se decrementa el valor de *s* hasta la unidad). En los experimentos que se describen en este documento trabajaremos con valores de *s* constante e igual a 1.

3.5.1 Escalado fino aprovechando la experiencia externa de otro algoritmo

El algoritmo GFAST que acabamos de describir no está completo dado que es posible incluir un escalado de granularidad fina, hasta la unidad, sin afectar al tiempo de ejecución ni al espacio de almacenamiento necesario. Sin embargo para usar esta peculiaridad del algoritmo de filtrado será necesaria cierta retroalimentación en forma de experiencia externa (ver Ilustración 19).

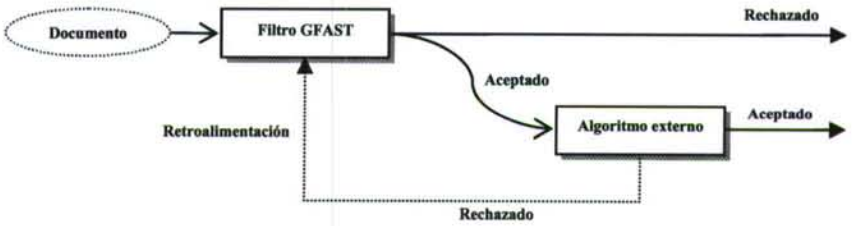

Ilustración 19: Relación del filtro con otro algoritmo - retroalimentación

En situaciones normales de uso, un filtro GFAST será combinado con otro algoritmo externo. Como ejemplo pongamos el que se corresponde con la inserción ordenada en una lista de tamaño limitado *T*. Por cada documento a procesar se invoca el filtro para minimizar el conjunto de documentos que entrarán en el algoritmo de inserción. En caso de ser aceptado por el filtro, el documento es insertado manteniendo el orden dentro de la lista, como esta es limitada es posible que como resultado obtengamos que dicho documento, aunque aceptado por el filtro, no es aceptado en la lista por sobrepasar el número máximo de elementos. Dado que el tamaño de la lista coincide con el tamaño del objetivo, podemos garantizar que dicho documento no formará parte del conjunto *T*, objetivo del filtrado; por tanto aquí tenemos una información, fruto de una experiencia externa al filtro que deseamos incorporar a este a modo de retroalimentación.

```
void GFAST::heuristic(peso)
begin
        if (peso + 1) / w igual a f_th then
                if (peso + 1) % w mayor que m_th
                        m_th ← (peso + 1) % w
                end if
        else
        if (peso + 1) / w mayor que f_th then
```

```
                f_th ← (peso + 1) / w
                m_th ← (peso + 1) % w
        end if
end
```

La heurística o experiencia de un algoritmo externo, provee la información acerca del documento con un peso dado que no debe ser incorporado al objetivo final, por tanto no debería pasar el filtro. Para ello se calcula de nuevo el umbral, teniendo ahora en cuenta que este está formado por un índice y un resto ocasionado al crear dicho índice a partir del peso y el ancho de la ranura.

El nuevo código modificado es tal que el umbral real (F_{th}) es $f_{th} * w + m_{th}$:

```
boolean GFAST::filter(peso)
begin
        if peso mayor o igual que (f_th * w + m_th) then
                c[peso/w] ← c[peso/w] + s
                S_th ← S_th + s
                while S_th - C[f_th] mayor o igual T do
                        S_th ← S_th - C[f_th]
                        f_th ← f_th + 1
                        m_th ← 0
                end while
                retorna falso
        else
                retorna cierto
        end if
end
```

De nuevo las modificaciones no afectan a la eficiencia del algoritmo, excepto por le hecho de que estamos invocando un método repetidas veces que en el mejor de los casos que hemos probado no pasó del 0,008% del total de documentos procesados[15].

Y al incrementar f_{th}, m_{th} se vuelve cero. Con esta modificación conseguimos que, si en algún momento durante el transcurso del filtrado un documento con *peso*, inicialmente aceptado, es finalmente rechazado por un algoritmo externo, el filtro pueda hacerse eco de este hecho, situando el umbral por encima del *peso* indicado, y haciéndolo con una granularidad mínima, dado que a partir de entonces estaremos filtrando teniendo en cuenta no sólo el índice f_{th}, si no el resto que dicho índice provoca.

3.6 Algoritmo BGFAST (Buffered Grooved Filter with Adaptative and Scaled Threshold)

GFAST funciona razonablemente bien, pero carece de una buena respuesta en los primeros momentos del filtrado. Durante este intervalo el algoritmo *aprende* y va situando el umbral

[15] Como ejemplo que ilustre este caso, un juego de prueba formado por 1024 paquetes con 1000 documentos cada uno invocó 82 veces el método *heuristic* y finalmente rechazó gracias a este motivo 436 documentos.

en valores cada vez mayores a medida que nuevos pesos van pasando. A partir de un punto el valor del umbral se estabiliza (ver Ilustración 16), o al menos crece más despacio, y su función pasa a ser la de un mero filtro. Podemos pues identificar dos etapas a lo largo del transcurso del filtrado de N_{in} documentos (ver Ilustración 20): Una etapa, que llamaremos de aprendizaje, corta y otra etapa, llamémosle de filtrado, más larga, que dura hasta el final del proceso. Si observamos el comportamiento de GFAST con $s = 1$, durante la etapa de aprendizaje pasan aproximadamente el 72% de los documentos de N_{out}, mientras que durante la etapa de filtrado sólo el 28% de los documentos[16] finalmente aceptados son procesados. Esto es, gran parte de los documentos de N_{out} que no formarán parte del subconjunto objetivo T, son aceptados durante la etapa de aprendizaje, y por tanto es aquí donde debemos enfocar nuestros esfuerzos de mejora.

Los cambios que introducimos nos proporcionan una versión más general del filtro a la que denominamos BGFAST (*Buffered Grooved Filter with Adaptative and Scaled Threshold*) y consiste en separar las etapas de aprendizaje y filtrado, de forma que el proceso aprenda y, cuando esté en una fase que podamos considerar estable, pase a la siguiente etapa en donde todos los pesos hasta entonces y a partir de entonces son eliminados o incorporados al conjunto final N_{out}. La versión más radical de BGFAST consiste en imaginar un filtro que a la llegada de los documentos, los procesa y los almacena (al menos los aceptados), y finalmente antes de reenviarlos, los vuelve a procesar sin modificar S_{th} dado que para entonces el valor de f_{th} (y m_{th}) ya será estable.

Un filtro BGFAST(T, w, s, f) con los parámetros objetivo (T), ancho de ranura (w), escala (s) y factor (f), posee un interface diferente al de su antecesor GFAST y su complejidad espacial es también diferente. Dado que estamos separando las etapas de aprendizaje y filtrado, necesitamos de una memoria que retrase lo más posible, para un documento dado, el momento en que este será aceptado o rechazado. Esta memoria implementada como una cola cíclica (FIFO), estará gobernada por el parámetro f, con la equivalencia BGFAST(T, w, s, 0) =GFAST(T, w, s). Con valores de f mayores que 0 aumentamos la capacidad de separar ambas etapas y llevar a cabo así un mejor filtrado. El valor de f es en realidad el tamaño de la cola cíclica en unidades de T, esto es, un factor f implica que la cola puede contener como máximo fT documentos.

[16] Esta característica nos recuerda el principio de Pareto.

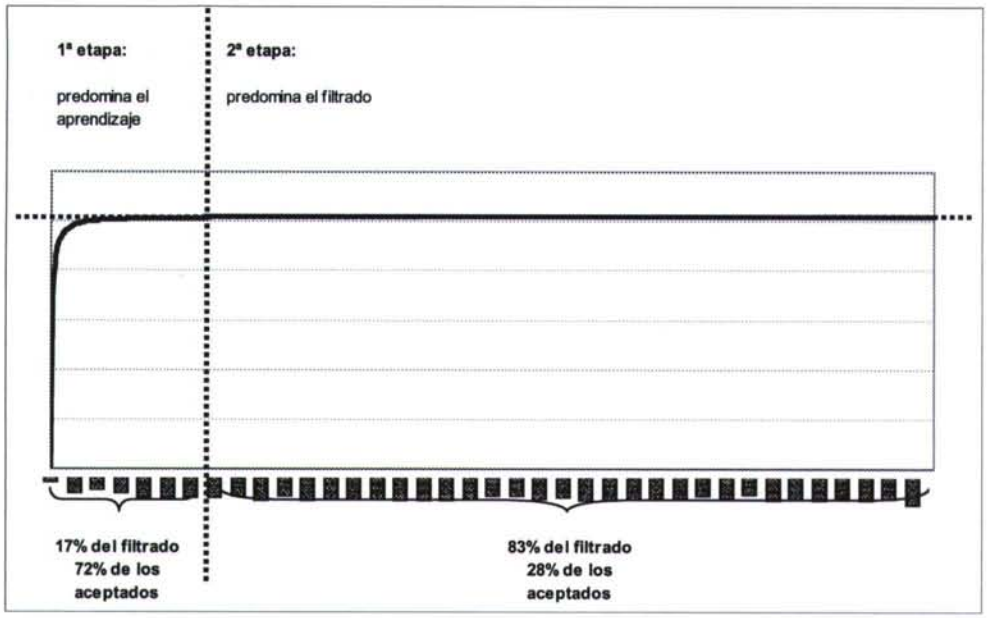

Ilustración 20: Etapas de aprendizaje y filtrado en un filtro GFAST

Este tipo de filtro posee tres interfaces o puntos de entrada: en común con GFAST el método *heuristic*, además de los métodos *push* y *pop*, este último con dos variantes.

```
integer BGFAST::push(documento)
begin
        if documento.peso mayor o igual que (f_th * w + m_th) then
                c[documento.peso/w] ← c[documento.peso/w] + s
                S_th ← S_th + s
                while S_th - C[f_th] mayor o igual que T do
                        S_th ← S_th - C[f_th]
                        f_th ← f_th + 1
                        m_th ← 0
                end while
                introduce documento en la cola circular.
                Si ha sido posible retorna 0
                Si no retorna +1
        else
                return -1
        end if
end
```

El algoritmo *push*, respecto a *filter* de GFAST, apenas presenta diferencias, excepto en los detalles de que si el documento es aceptado entonces es introducido en la cola circular y el significado del valor que retorna el método. Si la cola está llena, simplemente no se añade[17]. Sea como sea, el algoritmo *push*, expuesto aquí, representa la puerta de entrada de un documento en el filtro. Respecto al valor que retorna, esta función devuelve menor que cero

[17] Una buena práctica de programación debería evitar esta situación y garantizar que cuando se llame a *push*, haya al menos espacio para un nuevo elemento en la cola.

si el documento no pasa el filtro, mayor que cero si pasa el filtro y finalmente cero si aún no se sabe nada acerca de si pasa o no el filtro, esto es, el filtro retrasa la decisión de incluir o no el documento en el conjunto de salida.

```
integer BGFAST::pop(mandatory, var documento)
begin
        if mandatory == falso then
                si la cola está llena extrae e introdúce en documento
                en caso contrario retorna cero
        else
                si la cola no está vacía extrae e introdúce en documento
                en caso contrario retorna cero
        end if
        if documento.peso ≥ (f_th * w + m_th) then
                retorna +1
        else
                return -1
        end if
end
```

El método *pop* posee dos variantes, según el valor que tome el parámetro *booleano mandatory*. Si este es falso, el método sólo trabaja extrayendo un documento de la cola cíclica si esta está llena. De no estarlo, el método devuelve cero (en este caso, significa "*no hay nada que procesar*"). En el caso contrario, que *mandatory* sea cierto, el método extrae un documento de la cola circular si esta no está vacía, devolviendo cero en caso contrario (significando "*no hay nada más que procesar*"). En cualquier caso si *mandatory* es falso y la cola está llena o si *mandatory* es cierto y la cola no está vacía, el documento extraído es filtrado de nuevo, devolviéndose mayor que cero o menor que cero según sea aceptado o no.

Supongamos que la cola circular es lo suficientemente grande como para contener el número total de documentos a procesar, n_{in}. Por cada documento de entrada, este es procesado con *push*, el cual devuelve -1 si el documento es rechazado y 0 si es almacenado (siempre lo es en este caso). Al finalizar el procesamiento de todos los documentos, tenemos en la cola cíclica aquellos aceptados en la primera etapa, la de aprendizaje. Observemos que en este punto si no introducimos más documentos el filtro no aprende nada nuevo (f_{th} y m_{th} no cambiarán en el futuro) y por tanto estamos seguros de que la etapa de aprendizaje ha acabado. Aquí empieza la etapa de filtrado puro, llamando al método *pop* con *mandatory* igual a *true*. Invocando este método vamos extrayendo de la cola los documentos y sometiéndolos al valor umbral del filtro, el cual va rechazando (-1) o aceptando (+1) documentos hasta que no queden más que procesar (0).

Lógicamente, no siempre podremos disponer de una cola circular tan grande como el conjunto de documentos de entrada, bien por ser este muy grande bien por ser desconocido *a priori*. Por ello es interesante estudiar el tamaño adecuado de la memoria FIFO, dado que además, redundará en la eficacia del filtrado. La Ilustración 21 muestra la mejoría que presenta un filtro al que se somete el mismo juego de ensayo con diferentes valores para el parámetro *f*. La mejora es evidente. Sin embargo podemos observar como a partir de un

valor ($f = 8$), el resultado se vuelve constante, no hay mejoría. La razón para este límite hay que buscarla en el significado de f, recordemos que al tomar este parámetro el valor 8, significa que la cola puede almacenar $8T$ documentos, en nuestro caso 8000, precisamente casi el número de documentos (8259) que podemos esperar que filtre en el caso de $f = 0$. Aunque formalmente el valor óptimo de f debería ser 9 (primer múltiplo de T superior a 8259), debido a la distribución de los datos en el interior de los paquetes y entre ellos este valor se alcanza un poco antes.

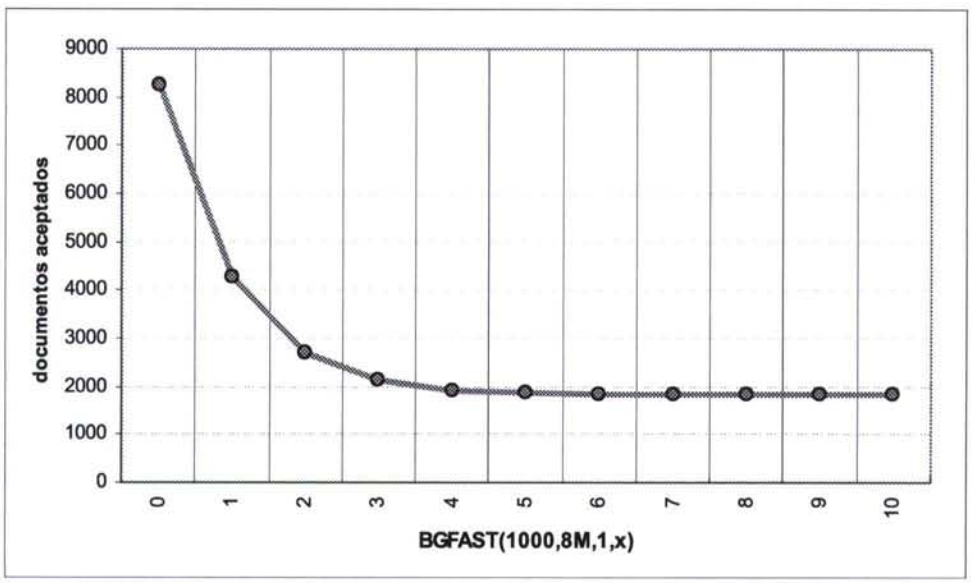

Ilustración 21: Evolución del filtrado para diferentes valores de *f*.

Dado que hemos calculado anteriormente una estimación del número de documentos que este algoritmo puede filtrar positivamente, podemos deducir que el número óptimo para f es:

$$\left\lceil \frac{\frac{T}{s}H_p}{T} \right\rceil = \left\lceil \frac{H_p}{s} \right\rceil = f$$

Y el cálculo del número de documentos efectivamente aceptados por el filtro con parámetros s y f es:

$$TnP_{s,f}(i \le n) = Tn - T\sum_{k=1+f}^{n+f} \frac{w\left\lfloor c(1-\frac{1}{sk}) \right\rfloor}{|W|} \ge Tn - T\sum_{k=1+f}^{n+f} \frac{wc(1-\frac{1}{sk})}{|W|} = Tn - T\sum_{k=1+f}^{n+f}(1-\frac{1}{sk})$$

El cual:

$$Tn - T(n - \sum_{k=1+f}^{n+f} \frac{1}{sk}) = Tn - Tn + \frac{T}{s} \sum_{k=1+f}^{n+f} \frac{1}{k} = \frac{T}{s} \sum_{k=1+f}^{n+f} \frac{1}{k} = \frac{T}{s}(H_{n+f} - H_f)$$

Para $f \geq 1$.

Esta fórmula puede ser aproximada para $f > 1$ como:

$$TnP_{s,f}(i \leq n) \approx \frac{T}{s}(Ln(n+f) + \gamma - (Ln(f) + \gamma)) = \frac{T}{s}(Ln(n+f) - Ln(f))$$

Y para el caso $f = 0$ como:

$$TnP_{s,0}(i \leq n) \approx \frac{T}{s}(Ln(n) + \gamma)$$

3.7 Filtros Basados en BGFAST

En la taxonomía mostrada en secciones anteriores algunos filtros estaban basados en algoritmos tradicionales combinados con un filtro BGFAST y otros eran diferentes modos de usar los filtros BGFAST individualmente.

Esta sección está dedicada a describir y analizar estos últimos algoritmos.

Por simplicidad, en esta documentación, mostraremos el código de los algoritmos usando el interface que GFAT entrega, mientras que las complejidades serán calculadas teniendo en cuenta los parámetros *s* y *f*.

3.7.1 Simple BGFAST

Este algoritmo muestra la aplicación más simple del filtro no determinista BGFAST(*T*, *w*, *s*, *f*). Textualmente hablando, el algoritmo procesa cada documento y cuando encuentra una entrada que ha de ser rechazada, recorta el paquete de este elemento hasta el final[18] y retransmite el paquete resultante. Es fácil entender como esta estrategia es el resultado de combinar el algoritmo *NoFilter* con el filtro BGFAST:

[18] Recordemos que la razón de hacerlo así es por que los documentos, dentro de cada paquete, estás ordenados de mayor a menor. Por lo que si al encontrar un documento que no pasa el filtro y sabiendo que el resto de los documentos del paquete desde esta hasta el final poseen un peso inferior al actual, podemos deducir que estos no pasarán igualmente el filtro.

```
function Simple-GFAT(packet)
begin
   for each di en packet do
   begin
      if di es rechazado por el filtro entonces break
   end

   Transmite packet recortado hacia peer
end
```

Este algoritmo presupone que los documentos en el paquete de entrada se encuentran ordenados según su peso. De esta forma al momento de encontrar un peso que no pasa el filtro (el algoritmo devuelve cierto) puede presuponer que el resto de documentos desde ese punto tampoco pasarán el filtro.

Su complejidad espacial es:

$$O(c + fT)$$

Dado que BGFAST requiere el uso de una memoria de fT elementos y un array de c entradas.

Respecto a la complejidad temporal de este algoritmo:

$$O(\alpha p + (\beta + \psi)\frac{T}{s}(H_{p+f} - H_f))$$

Donde podemos observar la fórmula para BGFAST que calcula el número de documentos que el filtro acepta y que finalmente determina el número de documentos que procesa (β) y retransmite (ψ). A medida que los paquetes se presentan al algoritmo, el filtro elimina más documentos debido al aprendizaje del umbral, y por tanto tarda menos en procesarlos y enviarlos.

Esta aproximación al filtrado IR ocasiona, como veremos unos tiempos muy pequeños de procesamiento (t_f), pero por otro lado provoca una gran cantidad de tráfico de salida (q). Dependiendo de la infraestructura de comunicaciones usada, este algoritmo será más o menos eficiente. También hemos de tener en cuenta dos consideraciones acerca de la calidad del tráfico resultante: por un lado los documentos de salida (N_{out}) están ordenados,

por el otro, al no poder estimar el número de documentos que este filtro genera, la implantación tiene que habilitar el protocolo de señalización *principio/fin*[19].

3.7.2 AcumulativeSL BGFAST

La subfamilia de filtros *Acumulative*, son una variación del anterior, en donde se esperará, bajo algún criterio, a reenviar el paquete de salida; al revés que el filtro *Simple* BGFAST, el cual retransmitía inmediatamente el paquete procesado.

El primero que veremos es aquel donde los documentos son insertados de forma ordenada (*Sorted*) en una memoria temporal de tamaño limitado y transmite su contenido, o parte de él, cuando alcanza un determinado valor (*Limited*) o se alcanza el final.

```
function AcumulativeSL-GFAT(packet)
begin
   for each di en packet do
   begin
      if di es aceptado then
         inserta con orden di en lista
         if di es rechazado realimenta el filtro, break
      else
         break
      end if
   end

   if en lista hay más de T documentos then
      transmite los T primeras entradas de lista hacia peer
      elimina las T primeras entrada de lista
   end if

   if es el último paquete then
      transmite lista hacia peer
   end if
end
```

Ahora la complejidad temporal de este algoritmo es

$$O(\alpha p+(T\beta+\psi)\frac{T}{s}(H_{p+f}-H_f))$$

Diferente que el anterior, dado que el procesamiento del paquete de entrada exige insertar en una lista ordenada y a la salida se retransmiten menos paquetes que en el caso anterior. Podemos esperar respecto al anterior que el tiempo de filtrado sea mayor, pero el tiempo de transmisión menor, dado que al agrupar más documentos en paquetes tendremos que

[19] A medida que procesa paquetes los recorta de forma que es posible que el último paquete recibido tenga cero documentos. Esta es realmente la única complicación que podemos encontrar y provoca el envío de un paquete con cero elemento para poder marcar el final del caudal.

invocar menos veces la entrada/salida de red, lo cual provocará menos retardos. El número de paquetes transmitidos (q), suponiendo que limitamos su tamaño a T documentos, es:

$$q = \left\lceil \frac{(H_{p+f} - H_f)}{s} \right\rceil$$

Que para $p = 1024$, $f = 0$ y $s = 1$ entonces es $q = 8$, aproximadamente. Esta fórmula será aplicable a todos los filtros de la familia *Acumulative Limited.*

El tamaño de la lista donde se almacenan ordenadamente los documentos aceptados es el doble del tamaño de paquete máximo esperado, que estamos suponiendo igual a T. De esta forma está garantizado que siempre habrá espacio para añadir un paquete entero a la lista. Su complejidad espacial es:

$$O(c + fT + 2T) = O(c + (f + 2)T)$$

3.7.3 AcumulativeUL BGFAST

El siguiente tipo de algoritmo que veremos es aquel donde los documentos son añadidos sin orden (*Unsorted*) en una memoria temporal de tamaño limitado y transmite su contenido, o parte de él, cuando alcanza un determinado valor (*Limited*).

```
function AcumulativeUL-GFAT(packet)
begin
   for each di en packet do
   begin
      if di es aceptado then
         añade di al final de lista
      else
         break
      end if
   end

   if en lista hay más de T documentos then
      transmite los T primeras entradas de lista hacia peer
      elimina las T primeras entrada de lista
   end if

   if es el último paquete then
      transmite lista hacia peer
   end if
end
```

La complejidad temporal de este algoritmo es

$$O(\alpha p + (\beta + \psi)\frac{T}{s}(H_{p+f} - H_f))$$

Muy inferior al anterior en cuanto al tiempo invertido en el procesamiento del paquete de entrada dado que ahora lo añadimos al final de una lista y finalmente se retransmiten en menos paquetes que en el caso anterior. Podemos esperar respecto al anterior que el tiempo de filtrado sea menor, y el tiempo de transmisión aproximadamente igual.

Su complejidad espacial es:

$$O(c+(f+2)T)$$

Igual que el anterior.

La lacra que introduce esta estrategia consiste en que los documentos que forman la salida no se encuentran ordenados. Esto puede ser un problema, dependiendo de la entidad destino del tráfico, dado que ahorramos tiempo en esta capa (ver Ilustración 18) pero inducimos un mayor tiempo de procesamiento en la etapa siguiente.

3.7.4 AcumulativeSU BGFAST

El siguiente tipo de algoritmo que veremos es aquel donde los documentos son añadidos en orden (*Sorted*) en una memoria temporal de tamaño ilimitado (*Unlimited*) y transmite su contenido de una sola vez al final del procesamiento de la consulta.

En general la subfamilia de filtros *Limited*, de la familia *Acumulative*, provocan una excepción ocasionada por las limitaciones intrínsecas de los protocolos utilizados. El tamaño de la memoria no está limitado, así que si tenemos el suficiente número de paquetes de entrada podemos encontrarnos con que el tamaño de la memoria a enviar crece por encima de los límites de transmisión del protocolo usado. En caso de que este sea UDP tenemos que el límite del datagrama a enviar es de 64KB.

El algoritmo queda así representado:

```
function AcumulativeSU-GFAT(packet)
begin
   for each di en packet do
   begin
      if di es aceptado then
         inserta con orden di en lista
         if di es rechazado realimenta el filtro, break
      else
         break
      end if
   end

   if es el último paquete then
      transmite lista hacia peer
   end if
end
```

La complejidad temporal de este algoritmo es

$$O(\alpha p+(T\beta+\psi)\frac{T}{s}(H_{p+f}-H_f))$$

Podemos esperar respecto al anterior que el tiempo de filtrado sea aproximadamente igual, y el tiempo de transmisión menor[20].

Respecto a su complejidad espacial, esta es:

$$O(c+fT+\frac{T}{s}(H_{p+f}-H_f))$$

Dado que tiene que almacenar todos los documentos aceptados por el filtro.

3.7.5 AcumulativeUU BGFAST

El siguiente tipo de algoritmo que veremos es aquel donde los documentos son añadidos sin orden (*Unsorted*) en una memoria temporal de tamaño ilimitado y transmite su contenido cuando alcanza el final de la consulta (*Unlimited*).

```
function AcumulativeUU-GFAT(packet)
begin
   for each di en packet do
   begin
      if di es aceptado then
         añade di al final de lista
      else
         break
      end if
   end

   if es el último paquete then
      transmite lista hacia peer
   end if
end
```

La complejidad temporal de este algoritmo es

$$O(\alpha p+(\beta+\psi)\frac{T}{s}(H_{p+f}-H_f))$$

La lacra que introduce esta estrategia consiste en que los documentos que forman la salida no se encuentran ordenados. Esto puede ser un problema, dependiendo de la entidad destino

[20] Transmitimos el mismo número de documentos que el anterior, pero en un solo paquete, lo que implica menos llamadas de entrada/salida al S.O., menos cambios de contexto y en general menos tiempo de procesamiento.

del tráfico, dado que ahorramos tiempo en esta capa (ver Ilustración 18) pero inducimos un mayor tiempo de procesamiento en la etapa siguiente.

Igual al anterior, su complejidad espacial es:

$$O(c + fT + \frac{T}{s}(H_{p+f} - H_f))$$

3.7.6 HardBroker BGFAST

El filtro visto en el apartado 3.3.2*HardBroker*, usaba un método muy simple (almacenar, ordenar, enviar) pero *a priori* poco eficiente. El que ahora nos ocupa es el resultado de aplicar un filtro BGFAST a aquel con el objetivo de disminuir el número de elementos almacenados y por tanto ordenados antes de ser enviados.

```
function HardBroker-GFAT(packet)
begin
   for each di en packet do
   begin
     si di pasa el filtro, añade di al final de total
   end

   if packet es el último
   begin
      ordenación parcial de los T primeros elementos de total
      total.count es T
      total.flags es SORT, FIRST, LAST
      transmite total hacia peer
   end
end
```

A diferencia del que desciende, este filtro antes de añadir cada documento a la lista, comprueba si pasa el filtro, esto es, si formará parte del objetivo. Con esta estrategia disminuimos considerablemente la complejidad temporal siendo esta:

$$O(\alpha p+\beta\frac{T}{s}(H_{p+f} - H_f)+\delta\frac{T}{s}(H_{p+f} - H_f)\log_2(\frac{T}{s}(H_{p+f} - H_f))+\psi T)$$

Obviamente menor que su versión sin filtrar siempre y cuando se cumpla [21], aproximadamente, para $s = 1$ y $f = 0$:

$$n_{in} > T(1+\frac{\gamma}{Ln(p)})$$

[21] Dicha inecuación se cumple para aproximadamente n_{in} igual a 1083, con T igual a 1000 y $p = 1024$.

Su complejidad espacial queda determinada por:

$$O(c + fT + \frac{T}{s}(H_{p+f} - H_f))$$

Mucho menor que su antecesor *HardBroker*, el cual requería un espacio **O**(pT).

3.7.7 SoftBroker BGFAST

En la sección 3.3.3definimos el filtro *SoftBroker*, el cual refinaba el anterior, *HardBroker*, con el objetivo de disminuir la complejidad espacial del algoritmo. Como en el apartado anterior, es posible aplicar de nuevo el filtro BGFAST con el propósito de disminuir la complejidad temporal:

```
function SoftBroker-GFAT(packet)
begin
   for each di en packet do
   begin
      si di pasa el filtro, inserta di en la lista ordenada total
      if la inserción ha fallado, realimenta el filtro y break
   end

   if packet es el último
   begin
      total.flags es SORT, FIRST, LAST
      transmite total hacia peer
   end
end
```

La complejidad espacial es:

$$O(c+fT+T)=O(c+(f+1)T)$$

Respecto a la eficacia, suponiendo el uso de GFAT, apenas significa una mejora para este algoritmo respecto a su antecesor, *SoftBroker*, dado que el número de documentos efectivamente insertados en una lista finita de tamaño T es ligeramente superior a la del número de documentos que un filtro GFAT permite pasar. Esta diferencia es:

$$T(H_{p,0.988} - H_p)$$

Que para $p = 1024$, apenas es 300 documentos. Esto implica que habrá pocos documentos[22] que habiendo sido aceptados por GFAT sean rechazados por la inserción en la lista finita y por tanto poca heurística de realimentación entre ambos algoritmos.

Generalizando para BGFAST:

$$T(H_{p,0.988} - (H_{p+f} - H_f))$$

A medida que aumentamos el valor de *f*, BGFAST filtra más documentos y, por tanto, aumenta esta diferencia, disminuyendo la posibilidad de que al insertar elementos en la lista finita estos sean rechazados. Según esto cabe esperar una disminución de la heurística entregada por el filtro que forma la lista finita.

Respecto a su complejidad temporal, el número de documentos efectivamente insertados en la lista (y por tanto no rechazados) está gobernado por el filtro BGFAST, quedando así:

$$O(\alpha p + \beta \frac{T^2}{s}(H_{p+f} - H_f) + \psi T)$$

Debemos observar que para $f = 0$ y $s = 1$ (GFAT), esta fórmula es muy parecida a la calculada anteriormente para *SoftBroker*:

$$O(\alpha p + \beta T^2 H_p + \psi T)$$

Lo que refleja una ligera mejoría dado el menor número de documentos insertados; pero hemos de destacar que habrá una penalización por el hecho de incorporar el código GFAT al filtro, por lo que podría arrojar peores resultados, en términos temporales, que su antecesor.

3.7.8 BrokerBinaryTree BGFAST

Finalmente el algoritmo resultante de añadir la etapa de filtrado, previo a la inserción en un árbol, se muestra en el siguiente *pseudo* código:

```
function BrokerBinaryTree-GFAT(packet)
```

[22] Según los cálculos expuestos aquí, de hecho no habrá documentos que sean aceptados por GFAT y rechazados por la lista finita. En los experimentos realizados podremos observar algún caso a través de la heurística del filtro BGFAST para *f* igual a cero.

```
begin
   for each di en packet do
   begin
      is di pasa el filtro, inserta di en el árbol total
      if la inserción ha fallado, realimenta el filtro y break
   end

   if packet es el último
   begin
      vuelca total sobre temp
      temp.flags es SORT, FIRST, LAST
      transmite total hacia peer
   end
end
```

Conceptualmente hablando la mejora que aporta esta estrategia lo hace por medio de la inserción en un árbol con control de balanceo. Por tanto, su complejidad temporal es:

$$\mathrm{O}(\alpha p + \beta 2\log_2(T)\frac{T}{s}(H_{p+f} - H_f) + (\delta + \psi)T).$$

A sí mismo, la complejidad espacial se puede considerar idéntica al anterior: $\mathrm{O}(c + (f+1)T)$, pero teniendo en cuenta que es necesario emplear más memoria para construir un árbol de T elementos que una lista de, igualmente, tamaño T.

4 Implementación del nodo programable

4.1 Introducción

Anteriormente, en el capítulo 2, hemos descrito conceptualmente las bases necesarias para entender como introducir el procesamiento de la semántica de la comunicación en el interior de su trayectoria. En dicho capítulo hemos descrito la arquitectura de nodo programable VAIN y en el actual vamos a describir en detalle las técnicas y tecnologías implicadas en la construcción de dicho nodo.

En primer lugar describiremos la herramienta RTASM creada al efecto para luego poder describir la técnica de clasificación de tráfico independiente. Seguidamente el texto dará paso a la implementación del gestor de memoria compartida el cual es la base del modelo de comunicaciones interno del nodo (*backbone*). Finalmente describiremos las estructuras de datos implicadas en dichas comunicaciones.

4.2 RTASM: Ensamblador en tiempo de ejecución

Los proyectos de desarrollo de software, en su fase de codificación, siguen un paradigma en donde podemos encontrar diferenciados al menos dos fases (Ilustración 22): construcción o compilación (análisis léxico y sintáctico, generación de código y montaje) y ejecución.

Cuando se desarrolló el primer ordenador digital las instrucciones que ejecutaba consistían en códigos numéricos que señalaban a los circuitos que la máquina los estados correspondientes en cada operación. Esta notación basada en códigos numéricos se denominó lenguaje máquina, que hoy en día es interpretado mediante el microcódigo residente en los procesadores. A medida que avanzó la investigación en lenguajes formales

[Chomsky, 59] creció el número de lenguajes disponibles para representar un algoritmo y el paradigma necesario para convertir un lenguaje fuente en lenguaje interpretable por la máquina cambió evolucionando hacia el que hoy en día podemos considerar tradicional[1].

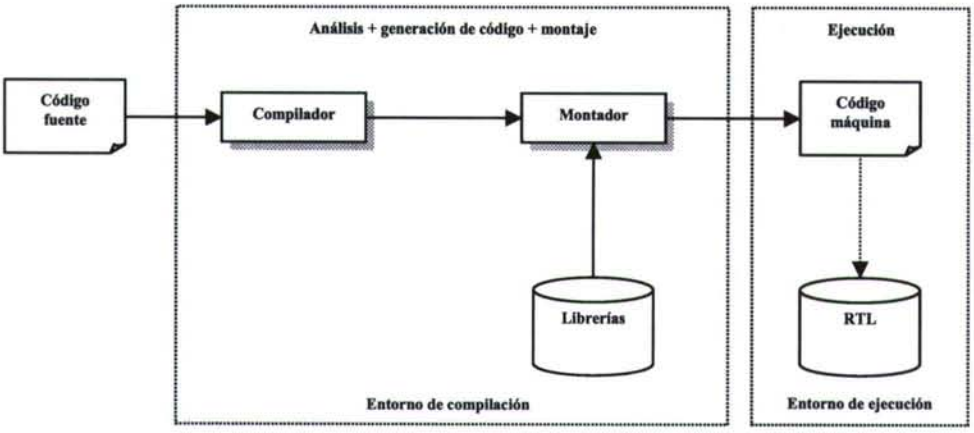

Ilustración 22: Entorno de compilación y ejecución tradicional

Según este modelo (Ilustración 22) un código fuente distribuido en uno o más ficheros de entrada es analizado (análisis léxico y sintáctico) y convertido a una notación intermedia, para más adelante ser enlazado con otros módulos y generar un código máquina susceptible de ser ejecutado en un entorno dispuesto al efecto. En el momento de la ejecución, y a instancia del código máquina generado, es posible la invocación de rutinas (RTL, *Runtime Libraries*) exclusivas del contexto de ejecución. Este modelo se caracteriza por que los contextos de generación de código y ejecución son dos contextos separados en tiempo, entorno e, incluso, en arquitectura.

La generación de código en tiempo de ejecución (RTCG) es una técnica consistente en unificar los contextos de compilación y ejecución en términos de tiempo y obviamente teniendo la misma arquitectura como fin.

RTCG puede definirse como la técnica consistente en añadir código dinámicamente al flujo de instrucciones de un programa en ejecución [Keppel, 91]. El mismo concepto está relacionado o es idéntico a los conceptos de especialización de código y generación dinámica de código.

[1] Aquí no consideramos los entornos interpretados o de ejecución JIT (*Just In Time*).

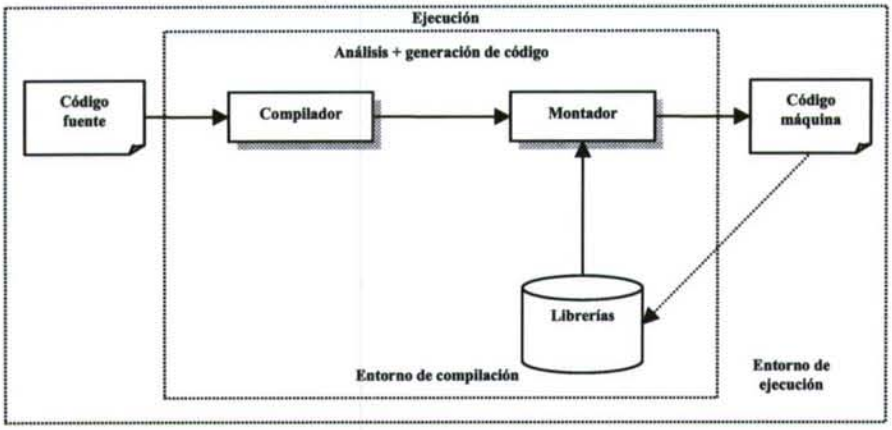

Ilustración 23: entorno de compilación y ejecución integrado

En el contexto de nuestro trabajo de investigación hemos desarrollado una herramienta (RTASM) con el objeto de disponer de la posibilidad de generar código dinámicamente. En el ámbito investigador existen herramientas con propósitos similares, como VCODE [Engler, 96a], o proyectos más extensos encuadrados en ámbitos de sistemas de compilación. Sin embargo ninguna de estas herramientas soportaba por entero los requerimientos establecidos, bien por no ser una herramienta estable, bien por no poder soportar un entorno multitarea, principalmente ente otras causas.

RTASM es una herramienta que entrega al programador un API sencillo y le permite incorporar código dinámicamente en tiempo de ejecución codificado como instrucciones en ensamblador de la arquitectura IA32 de Intel. Por ejemplo, el siguiente código:

```
RtAsm::Code*code=RtAsm::code_create(NULL, NULL);

RtAsm::Parser*parse=RtAsm::parser_create(code);

RtAsm::parser_parse(parser, "module test\n");
RtAsm::parser_parse(parser, ".allow all\n");
RtAsm::parser_parse(parser, " function factorial(dword n)\n");
RtAsm::parser_parse(parser, " begin");
RtAsm::parser_parse(parser, "  .prolog");
RtAsm::parser_parse(parser, "   mov $eax,n;");
RtAsm::parser_parse(parser, "   cmp $eax,01;");
RtAsm::parser_parse(parser, "   je  is_zero;");
RtAsm::parser_parse(parser, "   dec  $eax;");
RtAsm::parser_parse(parser, "   push $eax;");
RtAsm::parser_parse(parser, "   call near factorial;");
RtAsm::parser_parse(parser, "   add  $esp,sizeof($eax);");
RtAsm::parser_parse(parser, "   xor  $edx,$edx;");
RtAsm::parser_parse(parser, "   imul $eax,n;");
RtAsm::parser_parse(parser, "   jmp the_end;");
RtAsm::parser_parse(parser, "  :is_zero");
RtAsm::parser_parse(parser, "   mov $eax,1;");
RtAsm::parser_parse(parser, "  :the_end");
RtAsm::parser_parse(parser, "  .epilog");
RtAsm::parser_parse(parser, "   ret;");
RtAsm::parser_parse(parser, " end");

RtAsm::parser_finish(parser);
```

```
RtAsm::parser_check(parser);

if(RtAsm::code_errors(code)==0)
  {
   RtAsm::Render*render=RtAsm::render_create(code);
   RtAsm::render_do(render);
   RtAsm::render_free(render);

   RtAsm::code_realloc(code, NULL);
   RtAsm::code_clean(code);
  }
```

Crea en tiempo de ejecución el código máquina necesario para ejecutar la función *factorial* con un parámetro (*n*) entero largo sin signo. La invocación de dicha función puede llevarse a cabo usando el siguiente código:

```
void*ptr=RtAsm::code_symb(code,"test::factorial");
if(ptr!=NULL)
  {
   unsigned long v=((unsigned long(*)(unsigned long))ptr)(10);
   printf("fibonacci(10)=%lu\n", v);
  }
```

El cual imprime en la salida estándar el valor del factorial de 10.

Como podemos observar (Ilustración 24) el objeto *code* de tipo *RtAsm*::*Code*, se crea vacío y secuencialmente pasa por los objetos *parser* y *render*, los cuales son responsables respectivamente de llevar a cabo los análisis (léxico y sintáctico) y la generación de código. Al finalizar ambas etapas se obtiene un objeto *code* conteniendo el código máquina correspondiente al código fuente incorporado en la etapa *parser*.

RTASM reconoce 520 *opcodes* IA32 [Intel, URL] en 1491 instrucciones diferentes; los registros de propósito general, de punto flotante, MMX y XMM; y todos los formatos de direccionamiento soportados por la arquitectura IA32. RTASM añade el soporte nativo de módulos, estructuras, funciones con parámetros, así como prólogos y epílogos de funciones estándar. En general la relación entre las instrucciones ensamblador y las instrucciones código máquina es 1:1, esto es, por cada instrucción en ensamblador se genera una instrucción código maquina[2]. Las únicas excepciones son los prólogos y epílogos de las funciones, las cuales pueden ser consideradas como macros. Esta relación es importante, dado que implica necesariamente que RTASM carece de una etapa de optimización de código.

[2] VCODE, por ejemplo, no soporta esta relación.

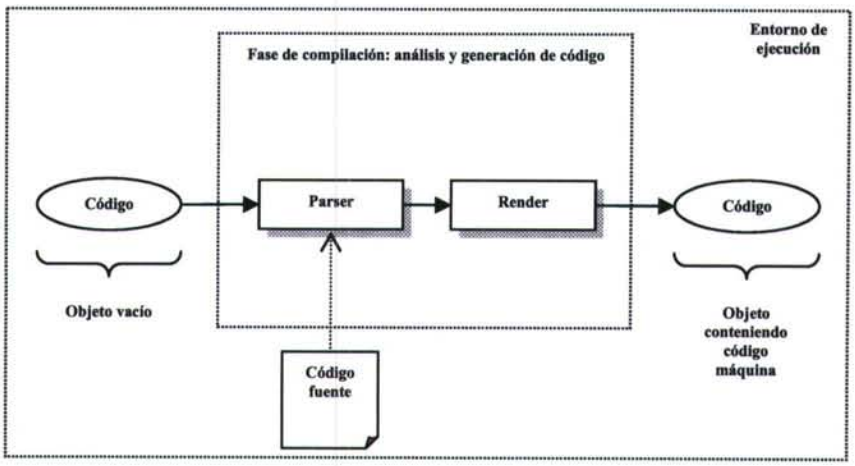

Ilustración 24: Modelo RTASM de generación de código

Para poder llevar a cabo esta herramienta hemos usado dos utilidades disponibles libremente. Por un lado la base de datos de instrucciones[3] del proyecto *Netwide Assembler* [NASM, URL], la cual ha servido como semilla para crear el codificador de las instrucciones RTASM y la razón de que este reconozca multitud de *opcodes* IA32. Por el otro lado hemos usado también el generador de analizadores sintácticos LALR(1) *Lemon* [Lemon, URL]: esta herramienta es muy similar a YACC [YACC, URL] o *Bison* [Bison, URL] pero con la característica de ser síncrono con respecto a la inserción de código fuente e independiente del analizador léxico, el cual se construyó en lenguaje C directamente.

4.3 Implementación del módulo de distribución de tráfico: ITC

En el capítulo 2 hemos introducido a nivel conceptual las bases de la clasificación de tráfico y hemos mostrado el funcionamiento general del nuestra solución ITC (δ2.3.2). En esta subsección vamos a mostrar cómo ha sido implementado, usando la herramienta RTASM vista en el apartado anterior (δ4.2).

Cada vez más a menudo las nuevas arquitecturas de comunicaciones exigen técnicas más complejas y eficientes para clasificar el tráfico en el interior de los dispositivos. Esta, a

[3] La base de datos de instrucciones del proyecto NASM consiste en una tabla en formato texto, conteniendo todos los *opcodes* IA32 de diversos procesadores (no necesariamente Intel), combinado con sus diferentes parámetros y reglas de codificación. A partir de esta información generamos de forma automática el código C necesario para codificar cada una de las 1491 instrucciones con sus respectivos y diferentes parámetros. Esto no sólo simplifico en términos temporales nuestro trabajo, si no que evitamos introducir errores dado que el proyecto NASM está considerado como estable y es ampliamente usado.

priori, humilde tarea se encarga de categorizar el tráfico de datos en virtud de sus características intrínsecas como paso previo a su procesamiento. Por su propia naturaleza la demultiplexación de tráfico es un cuello de botella en potencia, así que, de partida, la eficiencia es una condición necesaria. Sin embargo son deseables otros objetivos como la independencia de protocolos y la flexibilidad.

El objetivo secundario de la herramienta ITC es la construcción del árbol del esquema de red a partir de un API que permite al cliente de la utilidad añadir o eliminar nodos de forma ordenada. El diseñador puede también emplear un API de alto nivel que habilita introducir los subesquemas de red mediante la notación XML o CL (δ7.6).

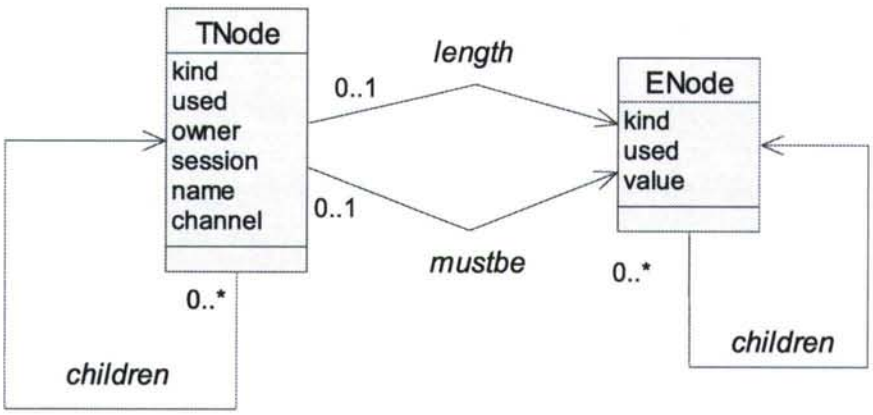

Ilustración 25: Modelo UML de las clases que forman el árbol del esquema

El árbol formado contiene en realidad dos tipos de estructuras (Ilustración 25), también de tipo árbol: *TNode* y *ENode*. Las primeras forman la súper estructura formada por protocolos, campos, opciones, *payloads* y expresiones, de forma jerárquica. La segunda contiene el árbol sintáctico de una expresión de red cualquiera (δ2.3.2.4 y δ7.7). Cada nodo *TNode* contiene los campos relacionados con el tipo (*protocol*, *field*, *payload*, etc.), el número de referencias (*used*), el usuario propietario (*owner*), la sesión actual (*session*), el nombre del nodo (*name*), el canal asociado (*channel*) y referencias raíz a nodos *ENode* para formar las expresiones *length* y *mustbe*. Por otro lado cada nodo *ENode* posee los campos relacionados con el tipo (el tipo de nodo, operador, operando, etc.), el número de referencias (*used*) y el valor del nodo (el valor del operando, por ejemplo). Cada nodo de cualquiera de los dos tipos, posee las referencias de sus nodos hijos (*children*) para poder formar la estructura de árbol.

El API de ITC permite la creación de nodos *TNode* y de forma indirecta la de nodos *ENode*. Los primeros son creados mediante las funciones[4]:

```
TNODE*itcCreateProtocol(ITC*itc, char*name,char*mustbe, TNODE*root);
TNODE*itcCreateField(ITC*itc, char*name,char*mustbe,char*length,TNODE*root);
TNODE*itcCreateOption(ITC*itc, char*name,char*mustbe, TNODE*root);
TNODE*itcCreatePayload(ITC*itc, char*mustbe, TNODE*root);
```

Estas funciones permiten crear los nodos del árbol principal, indicando los valores de los campos. Al hacerlo ITC controla que no se viola ninguna regla establecida, como incorporar un protocolo como nodo hijo de un campo. Así mismo compila las expresiones *length* y *mustbe* y las incorpora como referencias al nodo *TNode* creado.

Una vez que el esquema está formado (y en general cuando este sufre algún cambio significativo), usando la herramienta RTASM, es posible generar el código máquina encargado propiamente de clasificar el tráfico de entrada, el cual es el principal objetivo de la utilidad ITC, usando la siguiente función:

```
RtAsm::Code*itcGenerate(ITCAN*itcan);
```

La cual retorna un objeto *Code* (δ4.2) el cual permite acceder a dos rutinas generadas con las siguientes firmas:

```
unsigned long count();
long recognize(void*packet,unsigned length, list[], state[])
```

La primera es usada para conocer el número de canales abiertos, la segunda es la rutina generada que propiamente clasifica el paquete pasado como parámetro. Esta segunda función devuelve el número de canales que han reclamando el tráfico y de los cuales ha situado su identificador en *list*. El parámetro *state* permite activar/desactivar un canal sin regenerar todo el demultiplexor.

Al invocar la rutina *recognize*, los *arrays list* y *state* deben estar dimensionados para poder contener tantos elementos como indique la rutina *count*. La primera, al encontrar que el canal *i* ha reclamado tráfico que coincide con el paquete en curso comprueba si *state*[*i*] contiene un valor diferente de cero, en este caso añade el identificador del canal en *list*. En caso contrario ignora el canal y continúa el reconocimiento. Al finalizar la función *recognize* devuelve un número positivo o cero, indicando el número de canales situados en *list*; en caso de devolver un número negativo estaríamos ante un error, bien del esquema

[4] El API ITC contiene muchas más funciones, para crear sesiones, canales, para eliminar del árbol nodos *TNode* específicos, para recorrer el árbol, imprimirlo en la salida estándar, etc. Aquí mostraremos un subconjunto muy reducido de estas, realmente las más importantes.

bien de que este paquete no encaja con la estructura esperada (por ejemplo, el reconocimiento no ha acabado pero hemos rebasado el tamaño del paquete).

El código generado está optimizado según varias estrategias, por ejemplo las subexpresiones comunes de varias expresiones (árboles *ENode*) son calculadas sólo una vez, teniendo en cuenta cual de las subexpresiones es calculada primero. El resto de las subexpresiones no se evalúan y copian el valor producido para la primera subexpresión.

ITC posee una limitación debido a la arquitectura. El tamaño máximo de un campo, susceptible de ser usado como operando en una expresión, es de 32 bits, dado que ha de caber en un registro de propósito general IA32 (principalmente en EAX). Por ello, en determinados contextos, ITC ha recibido el nombre de ITC32, para diferenciarlo de futuras implementaciones que salven esta limitación[5].

4.4 Implementación del gestor de memoria compartida

En la Ilustración 6 (δ2.3.1) podemos observar como existe en la capa de ejecución un elemento que hemos denominado *backbone*, y lo hemos definido como el núcleo de comunicaciones interna del nodo. Esta entidad se implementa por encima de una memoria compartida que es usada, a lo largo del nodo, como memoria interna accesible por todos los elementos del mismo. Esta sección describe la implementación de esta memoria compartida, sus propiedades y limitaciones, y el modelo de concurrencia. La siguiente sección describirá cómo implantar el modelo de comunicaciones interna por encima de esta utilidad.

La Ilustración 26 muestra el modelo UML de la clase *shmem*, la cual representa el interface de la gestión de la memoria compartida y que forma parte del control de la capa de ejecución (δ2.3.3). Usaremos este interface como índice de la implementación.

La memoria compartida es una utilidad proporcionado por la gran mayoría de sistemas operativos actuales, y en concreto en la familia *Unix*, amparada bajo el epígrafe IPC (*Interprocess Communications*) y respaldado por el estándar POSIX (IEEE 1023). La memoria compartida se refiere típicamente a un gran bloque de memoria de acceso aleatorio (RAM) que puede ser accedido por diferentes procesos en un mismo entorno.

Desde nuestro punto de vista, existen tres grandes problemas a la hora de implantar un gestor de memoria compartida: el direccionamiento interproceso, la gestión de la

[5] Usando registros XMM o MMX podríamos manejar campos de 64 y 128 bits respectivamente. Lo que abriría la posibilidad de manipular fácilmente algebraicamente direcciones IPv6, por ejemplo.

concurrencia y la estrategia de reserva de espacio. Veamos como hemos solucionado estos tres problemas en nuestra implementación.

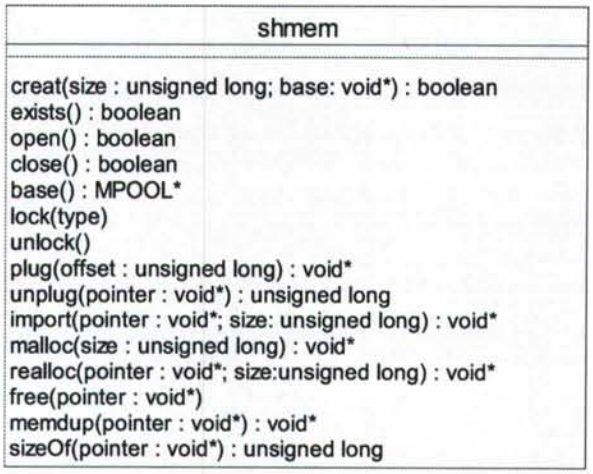

Ilustración 26: Modelo UML de la clase *shmem*

4.4.1 Direccionamiento interproceso

La técnica de memoria compartida es posible gracias a la tecnología de memoria virtual, en donde un sistema operativo gestiona la memoria disponible real y entrega a los procesos a los que sirve un punto de vista virtual de la misma.

La memoria virtual es una técnica de gestión de memoria usado por los sistemas operativos multi tarea, donde memoria no contigua es presentada a los procesos como memoria continua. El punto de vista de los procesos sobre la memoria que gestionan es denominado espacio de direccionamiento virtual. Esta habilidad de los sistemas operativos permite entre otras cosas almacenar páginas[6] de memoria real no usadas frecuentemente en memorias secundarias, entregando así un entorno jerárquico de memoria. Las páginas más frecuentemente usadas residen en RAM mientras que las usadas en pocas ocasiones se vuelcan (*swaping*) a dispositivos de almacenamiento secundario. Esta técnica permite que la memoria virtual disponible sea mucho más grande que la memoria real a disposición del sistema operativo.

Cuando el sistema operativo detecta un acceso, por parte de un proceso, a una dirección virtual que no se encuentra en estos momentos en la memoria RAM, se reclama al medio de

[6] Típicamente la memoria real es particionada en páginas de tamaño idéntico. La página es la unidad de intercambio (*swaping*) entre los niveles jerárquicos de la memoria virtual.

almacenamiento secundario y se instala sobre una página disponible (quizá intercambiándola con una existente, *swaping*) pudiendo continuar la ejecución del proceso sobre la dirección virtual invocada.

La gestión de la memoria virtual tiene como consecuencia que varios procesos convivan concurrentemente con idénticos espacios de direcciones virtuales, proyectados sobre un mismo espacio de almacenamiento real. Esto permite que entre dos o más procesos se puedan compartir segmentos de memoria, manejando cada uno de ellos diferentes direcciones pero haciendo referencia a exactamente el mismo contenido.

En realidad las implementaciones del estándar POSIX permite solicitar al crear o abrir un segmento de memoria compartida que este se encuentre en un punto concreto del espacio de direcciones de la memoria virtual, sin embargo, aunque esta posibilidad simplifica su gestión, hace muy difícil la codificación de programas dado que el montador (*linker*) de la plataforma ha de decidir, inmediatamente antes de la ejecución, en donde situar los segmentos de código y datos (del programa y de las librerías de enlace dinámico); pudiendo ocurrir que al solicitar una determinada dirección esta se solape con alguna otra ya usada[7].

Una técnica común, que hemos usado en nuestra implementación, es usar direcciones relativas entre los procesos, de forma que si una referencia a una dirección ha de ser pasada entre dos *threads*[8] cualesquiera, realmente se pasa el desplazamiento de dicha dirección desde la base de la memoria compartida.

Así pues disponemos de dos métodos para referenciar o desferenciar una dirección:

```
void*   plug   (dword_t off);
dword_t unplug(void*ptr);
```

El método *plug* recibe como parámetro un desplazamiento desde la base de la memoria compartida y devuelve el puntero absoluto a la dirección virtual que el proceso puede usar para referenciar el contenido. Por el otro lado, el método *unplug* realiza la tarea inversa, a partir de una referencia absoluta calcula su desplazamiento desde la base de la memoria compartida y la devuelve. Todas las estructuras conteniendo punteros en el interior de la memoria compartida, en nuestra implementación, tienen en cuenta este hecho y referencian/desferencian apropiadamente los valores de estas según sea conveniente.

[7] De hecho el API de nuestra implementación de memoria compartida soporta que el programador solicite una dirección base concreta. Sin embargo el nodo no usa esta capacidad, dado que hace muy difícil (de hecho es una situación no determinista) encontrar en tiempo de ejecución un segmento común libre.

[8] Cualquier *thread* en diferentes o igual procesos.

El método *base* permite en todo momento conocer el valor de la base de la memoria compartida.

4.4.2 Gestión de concurrencia

Dado que más de un proceso o *thread* pueden leer o cambiar el contenido de la memoria compartida es necesario implementar algún mecanismo de concurrencia que permita realizar secuencias de lectura/escritura coherentes. Nuestra implementación la lleva a cabo en dos niveles: un primer nivel permite el bloqueo de un objeto semáforo asociado a la memoria virtual en modo conmutado (si/no), permitiendo que solamente un *thread* adquiera el semáforo a la vez. Un segundo nivel, que usa al anterior, permite una forma de ínter bloqueo más elaborada, habilitando o bien un único escritor o bien múltiples lectores a la vez; todo ello recursivamente.

Un semáforo es una variable protegida o un tipo abstracto de dato, y constituye un método clásico para restringir el acceso a los recursos compartidos. Las implementaciones del estándar POSIX entregan dicha funcionalidad como un tipo especial de comunicación entre procesos (IPC) permitiendo la consulta y modificación de dicha variable de forma atómica. Nosotros hemos empleado esta funcionalidad para implementar el nivel más bajo del control de concurrencia.

El nivel más alto del control de concurrencia gestiona una lista[9] conteniendo una entrada por *thread*, la cual contiene a su vez la información relativa al número de bloqueos realizados (*lock.count*) y el tipo de bloqueo (lectura/escritura) actual (*lock.kind*).

Cuando un *thread*, representado por *pinfo*, desea adquirir un bloqueo para lectura, lleva a cabo el siguiente proceso:

```
if(pinfo.lock.counter==0)
  {
   semop(semid,-1);
   pinfo.lock.kind=O_RDONLY;
   pinfo.lock.counter+=1;
  }
else
  {
   pinfo.lock.counter+=1;
  }
```

Si no tiene actualmente ningún bloqueo, invoca el bloqueo del semáforo a bajo nivel (*semop*) indicándole que desea restar su valor en una unidad. Como esta operación es

[9] Situada en la memoria compartida y por tanto accesible a todos los procesos.

atómica, esta llamada sólo retornará cuando haya sido posible dicha operación (esto es, cuando no haya nadie escribiendo[10]). Así pues, seguidamente, almacena el tipo de bloqueo que ha adquirido (O_RDONLY) e incrementa el contador de bloqueos propio.

En caso de poseer en este momento al menos un bloqueo, este ha de ser de lectura o escritura (necesariamente) con lo cual, se limita a incrementar su contador privado de bloqueos. Aquí es indistinto si el bloqueo anterior fue de lectura o escritura, en cualquiera de los dos casos se permite volver a bloquear para leer. No ocurrirá lo mismo en el caso contrario, bloqueado para lectura y petición de bloqueo para escritura, como veremos más adelante.

Cuando un *thread* desea adquirir un bloqueo para escritura, lleva a cabo el siguiente proceso:

```
if(pinfo.lock.counter==0)
  {
   semop(semid,-MAXCURRENT);
   pinfo.lock.kind=O_RDWR;
   pinfo.lock.counter+=1;
  }
else
  {
   if(pinfo.lock.kind==O_RDONLY)
     {
      semop(semid,+1          );
      semop(semid,-MAXCURRENT);
      pinfo.lock.kind=O_RDWR;
      pinfo.lock.counter+=1;
     }
   else
   if(pinfo.lock.kind==O_RDWR)
     {
      pinfo.lock.counter+=1;
     }
  }
```

En este caso, si este *thread* no soporta actualmente ningún bloqueo de ningún tipo, se invoca una llamada al nivel más bajo de bloqueo solicitando que el semáforo se decremente en el valor máximo posible[11]. Debido a la semántica del semáforo, esta llamada retornará únicamente cuando el semáforo haya sido decrementado en MAXCURRENT unidades[12], lo que ocurrirá cuando no haya ningún otro *thread* ni leyendo ni escribiendo. Por tanto el

[10] Y siempre y cuando no se haya rebasado el número máximo de lectores (MAXCURRENT).

[11] MAXCURRENT es una constante que limita el número de *threads* que simultáneamente pueden acceder a la memoria compartida. Su valor es establecido en tiempo de diseño y actualmente posee el valor 512.

[12] La semántica del semáforo POSIX es tal que una operación (incremento, decremento) puede llevar a cabo si y sólo si esta no ocasiona un valor del semáforo negativo.

acceso exclusivo está garantizado. Seguidamente el código se limita a almacenar el tipo de bloqueo e incrementar su contador individual.

En caso de que hubiera un bloqueo establecido por parte del *thread*, tenemos que dilucidar si es compatible con el tipo de bloqueo solicitado, esto es: si está bloqueado para escribir y solicitamos volver a bloquear para escribir, no hay problema; sin embargo, si hemos bloqueado para leer y estamos solicitando para escribir, hay que transformar el bloqueo anterior de lectura en otro de lectura/escritura.

Para hacer esto primero comprobamos el tipo de bloqueo que actualmente está activo, si es de lectura/escritura (O_RDWR) el código se limita a incrementar el contador de bloqueos del *thread*. La otra posibilidad es que el bloqueo se encuentre establecido para lectura y estemos intentando un bloqueo del tipo lectura/escritura. Entonces estamos obligados a cambiar el tipo de bloqueo. Para ello liberamos el bloqueo a bajo nivel (+1) y seguidamente lo adquirimos para lectura/escritura decrementando el valor del semáforo en el máximo valor posible. Cada una de estas operaciones es atómica y el *thread* puede ser bloqueado durante más tiempo, por lo que es una situación a evitar. Una vez que ambas operaciones sobre el semáforo han sido realizadas registramos el hecho modificando el tipo de bloqueo (O_RDWR) e incrementando el contador individual de bloqueos para el *thread*.

El proceso inverso, esto es, el desbloqueo (*unlock*) es similar al descrito anteriormente.

Para desbloquear hemos de tener en cuenta si tenemos algo bloqueado, y hemos de diferenciar si el contador (*pinfo.lock.counter*) es estrictamente la unidad o superior.

En caso de ser superior a 1, nos limitaremos a decrementar el contador del *thread*, tanto si el bloqueo actual sea exclusivamente para lectura o para lectura/escritura.

```
if(pinfo.lock.counter==1)
  {
   semop(semid,+1);
   pinfo.lock.kind=0;
   pinfo.lock.counter=0;
  }
else
if(pinfo.lock.counter>1)
  {
   pinfc.lock.counter-=1;
  }
```

El desbloqueo, en el caso de ser el contador igual a uno, es similar en ambos casos. Si estuviese actualmente bloqueado para lectura, incrementamos el valor del semáforo en una unidad y ponemos a cero tanto el contador de bloqueos como el tipo de bloqueo. Similarmente, en el caso de estar bloqueado para escritura, incrementamos el semáforo en el valor máximo posible.

```
if(pinfo.lock.counter==1)
  {
   semop(semid,+MAXCURRENT);
   pinfo.lock.kind=0;
   pinfo.lock.counter=0;
  }
else
if(pinfo.lock.counter>1)
  {
   pinfo.lock.counter-=1;
  }
```

4.4.3 Estrategia de reserva de espacio

Hasta ahora hemos considerado las técnicas para establecer un segmento de memoria compartida y un método de ínter bloqueo de procesos. Ahora hemos de establecer la estrategia usada para asignar o reservar porciones discretas de esta memoria y poder así gestionar eficazmente el espacio limitado.

La reserva dinámica de memoria es una técnica de gestión del almacenamiento (típicamente RAM) durante la ejecución de uno o más programas. Es una forma de distribuir la propiedad temporal de una porción de los recursos de memoria limitados (objeto) entre, en nuestro caso, varios *threads*. Un objeto reservado dinámicamente permanece en este estado hasta que es explícitamente liberado bien directamente por el *thread* o bien por medio de un mecanismo de *garbage collection*. Durante su tiempo de vida este objeto es propiedad exclusiva del *thread* propietario (que no será necesariamente el creador del objeto).

Son varias las formas en la que podemos llevar a cabo esta tarea, de entre estas hemos seleccionado una de las más simples, consistente en particionar el espacio en unidades discretas de idéntico tamaño (*chunk*), de 128 bytes, y mantener un índice binario indicando el estado de cada *chunk* de la memoria compartida. Así pues el segmento (que siempre es múltiplo del tamaño de página del sistema, en nuestro caso 4KB) es dividido en tres partes bien diferenciadas: la cabecera (HEAD), el índice (CAT) y los datos (DATA). El primero almacena los datos de carácter general de la memoria compartida (identificadores, tamaños, desplazamientos del CAT y el HEAD, siguiente *chunk*, contador de identificadores únicos y la lista de procesos, entre otros). La zona CAT almacena 2 bits por cada *chunk* en que dividimos la zona DATA. Según el valor de estos dos bits podremos conocer el estado del *chunk*: libre (0b00) y ocupado (0b01, 0b11)[13]. Un *chunk* puede estar ocupado, pero de dos formas diferentes, formando parte del primer *chunk* de un segmento reservado o formando parte de los siguientes *chunks*.

[13] El valor 0b10 no es usado en este contexto y es considerado como un error.

Esta estrategia permite una búsqueda rápida de espacio libre, pero ocasiona un porcentaje de fragmentación interna y externa. Esta desventaja queda parcialmente difuminada al analizar el tipo de procesos que usarán la memoria compartida. El uso típico que harán los servicios de esta herramienta será reservar y liberar porciones pequeñas de memoria con tiempos de vida muy pequeños. La influencia de reservar pequeños segmentos de memoria sobre la fragmentación interna es significativa positivamente, y respecto a la fragmentación externa, esta es disminuida dado que los segmentos de memoria (típicamente paquetes) sobreviven durante un periodo muy pequeño de tiempo, el necesario para procesarlos y reenviarlos.

El sistema de gestión de la memoria compartida mantiene en la cabecera de cada segmento una información formada por el contador de referencias y el propietario del segmento. El primero se encarga de llevar la cuenta del número de referencias realizadas sobre el segmento, se incrementa a petición del usuario y se decrementa también a petición del usuario o por una llamada al método *free*. El segmento se libera definitivamente sólo cuando este contador alcanza el valor cero. Por el otro lado la cabecera del segmento también incluye la identificación del propietario del segmento, usado para liberar los segmentos reservados por un *thread* que ha desaparecido descontroladamente[14].

Las primitivas usadas para gestionar esta memoria son básicamente:

```
void*    import (void*pointer, unsigned size);
void*    malloc (unsigned size);
void*    realloc(void*pointer, unsigned size);
void     free   (void*pointer);
void*    memdup (void*pointer);
unsigned sizeOf (void*pointer);
```

La semántica de los métodos *malloc*, *realloc* y *free* son idénticas a las que podemos encontrar en el estándar POSIX: reserva, redimensionamiento y liberación de segmentos de memoria, respectivamente. Sin embargo hemos añadido otras tres primitivas muy útiles: *import* permite importar un segmento de memoria de un determinado tamaño desde el *heap* privado del *thread* a la memoria compartida. El método *memdup* permite duplicar un segmento de la memoria compartida, tanto en su geometría como en su contenido. Finalmente, *sizeOf*, permite conocer el tamaño del segmento dado.

[14] Es, por tanto, un tipo simple de *garbage collection*.

4.5 Implementación del modelo de comunicaciones interno

Internamente, los diferentes componentes que forman un nodo del tipo que estamos describiendo, se comunican mediante el paradigma de paso de mensajes punto a punto. Cada entidad (en general, servicio) que desee comunicarse debe, en primer lugar, crear un objeto de tipo *Mailbox* (Ilustración 27).

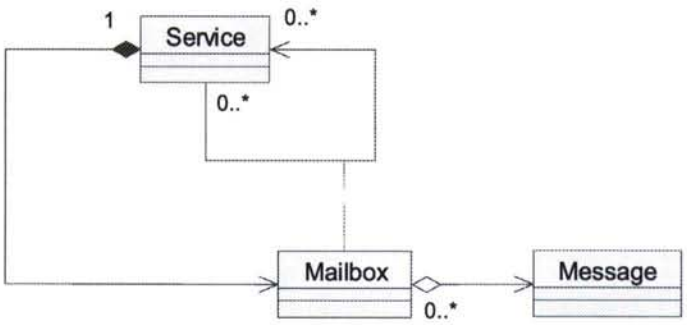

Ilustración 27: Modelo UML de comunicaciones interna

Cada servicio posee un *mailbox*, el cual es usado por otros servicios para mandar mensajes con destino a este servicio. Cada *mailbox* se implementa en un segmento de memoria compartida como una lista cíclica con la estructura mostrada en la Ilustración 28.

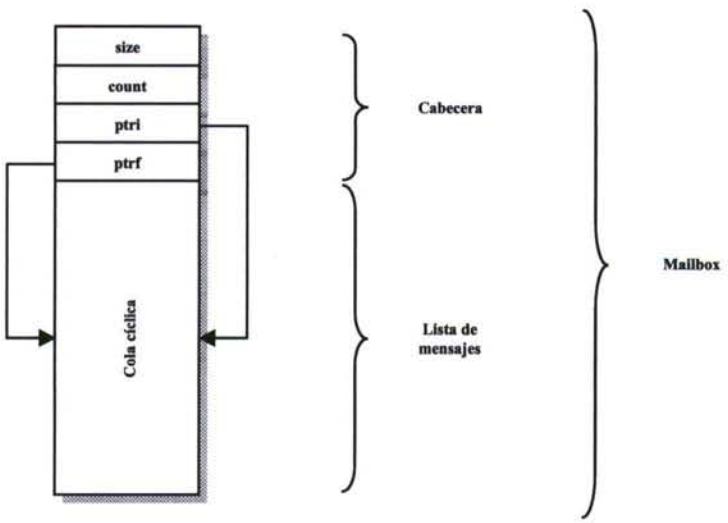

Ilustración 28: Esquema de un mailbox

En la cabecera (HEAD) de la memoria compartida se encuentra la lista estática de los *threads* registrados en un momento dado. En esta lista podemos encontrar por cada proceso registrado: su estado, identificador índice, identificador de proceso (*pid*) y thread (*tid*), su

nombre, datos acerca de sus bloqueos y el desplazamiento de su *mailbox*. Esta lista es el punto de entrada que otro servicio usa para buscar su destinatario, dado el nombre.

El *mailbox* de un servicio puede contener hasta 255 mensajes de tamaño arbitrario[15]. Cada mensaje está formado por: tipo, remitente, id. de mensaje, id. de respuesta, valor y contenido.

El tipo del mensaje (ver δ7.10.2) permite manejar apropiadamente el mensaje, indicando alguna combinación de las siguientes características: texto, binario, comando (implica texto), paquete (implica binario), confirmación, urgente, valor, y ACK. Si la característica texto está activada, entonces el contenido del mensaje ha de ser tratado como una lista de caracteres ASCII terminada en cero binario. En caso contrario estará marcado como binario y el contenido del mensaje es tratado como un *array* de bytes sin formato predefinido (emisor y receptor deberán conocer dicho formato). Si ninguno de estos dos está presente en las características del mensaje, entonces este no posee un contenido.

Los tipos paquete y comando son un tipo especial que determinan un formato predefinido del contenido, respectivamente un paquete de datos (IPv4, por ejemplo) y un comando de tipo texto (en lenguaje CL, siempre). Ambas características implican binario y texto pertinentemente.

El tipo confirmación indica que el emisor quedará a la espera de una respuesta y por tanto el receptor está obligado de generar una respuesta, aunque sea un simple ACK. De manera similar la etiqueta urgente indica al sistema que este mensaje ha de ser tratado con prioridad alta. Este modificador implica que el mensaje será encolado en el *mailbox*, no al final de la lista, si no al principio.

Finalmente el tipo valor y ACK están relacionados. La etiqueta valor indica que el mensaje tiene un valor (entero de 32 bits sin signo) incorporado. El tipo ACK implica la etiqueta valor y indica que este contiene una confirmación (¡=0) o un rechazo (==0).

Además de estos tipos, el usuario del mailbox puede indicar otras etiquetas (dispone de 8 bits) las cuales no influirán en el sistema, dado que este se limita a pasarlas tal cual, sin interpretarlas.

Cada mensaje al ser introducido en un mailbox recibe un número único que lo identifica sin ambigüedades. Este es el *id* del mensaje. Cada mensaje registra su *id* y el *id* del mensaje del

[15] Cada *mailbox* ocupa cerca de 5KB de memoria compartida, unos 40 *chunks*.

que es respuesta, de esta manera es posible seguir un diálogo complejo con varios servicios al mismo tiempo, dado que las primitivas de gestión del *mailbox* permiten extraer un mensaje no sólo del principio del *mailbox*, si no que habilita extraer aquel que es respuesta de uno dado.

Respecto al interface de gestión del *mailbox*:

```
bool      open(const char*path);
bool      close();
bool      isOpen();
```

Dado un servicio, estas primitivas[16] permiten abrir un *mailbox*, cerrarlo o saber si está abierto (o existe). Cada servicio posee un nombre (por ejemplo *deploy*, el servicio encargado de distribuir entre el resto de los servicios el tráfico después de clasificarlo) y su *mailbox* hereda dicho nombre, complementándolo con la trayectoria a la que pertenece (siguiendo con el ejemplo anterior */users/root/deploy*).

```
sid_t     self();
sid_t     lookFor(const char*name, long timeout);
char*     lookFor(const sid_t sid, long timeout);
```

El tipo *sid_t* caracteriza las variables que contienen un identificador de servicio (*service identifier*). Un servicio puede conocer su identificador mediante la primitiva *self*. Para buscar el identificador de un servicio, dado el nombre (o viceversa), el API entrega los métodos *lookFor* (sobrecargados).

```
bool      dependOn(char*path,bool enable);
bool      waitForDependencies();
```

Habitualmente los servicios dependerán unos de otros, así que es necesario habilitar un método para registrar estas dependencias y esperar por ellas. Estas primitivas permiten incorporar los servicios de que depende el servicio invocante (o eliminarlos del árbol de dependencias si el parámetro *enable* es *false*) y esperar a que estas sean satisfechas. En general la eliminación de un servicio implica la eliminación de aquellos que dependen de él.

```
bool    wait();
mid_t   send(sid_t sid,Message&message,mid_t mid,long timeout,sid_t from);
bool    waitForReply(Message&message,sid_t&sid,mid_t&rid,mid_t mid,long tm);
bool    confirm(sid_t sid,mid_t mid,long timeout);
bool    receive(Message&message,sid_t&sid,mid_t&mid,mid_t&rid);
mid_t   sendAndWait(sid_t sid,Message&message,long timeout,sid_t from);
```

Finalmente las primitivas principales del interface, aquellas relacionadas con los mensajes. El método *wait* espera a que haya algún mensaje en el *mailbox*, bloqueando

[16] Todas ellas son métodos de la clase *mailbox*.

indefinidamente el servicio en caso negativo. La señalización se hace mediante las primitivas POSIX *kill* y *pause*: la primera envía una señal al proceso indicado (en el caso de nuestra implementación SIGURG[17]) y la segunda provoca que el proceso interrumpa su ejecución hasta que reciba un mensaje cualquiera.

La función posiblemente más utilizada es *send*, encargada de enviar el mensaje *message* con destino al servicio identificado por *sid*, como respuesta a un mensaje anterior identificado como *mid* (cero si no es respuesta de otro). El parámetro *from* es útil cuando queremos que el mensaje sea enviado en nombre de otro servicio. En este caso el receptor recibirá el mensaje tal y como si hubiera sido enviado por el servicio identificado por *from*. Esta función devuelve el identificador del mensaje enviado, el cual puede ser usado para confirmar que el destinatario ha leído el mensaje por medio del método *confirm*, el cual retorna *true* si el mensaje *mid* ha sido extraído del *mailbox* del servicio *sid*.

El método *receive* comprueba si hay algún mensaje al principio del mailbox del servicio, de haberlo lo retorna en message, junto con el emisor (*sid*), el identificador del mensaje (*mid*) y el identificador de serlo del mensaje del que es respuesta (*rid*). Si no hay mensajes la primitiva retorna inmediatamente devolviendo *false*.

Una forma más refinada de recibir mensajes es esperar por una respuesta a un mensaje enviado previamente. La primitiva *waitForReply* hace precisamente esto, espera por la respuesta del mensaje *mid* y entrega en *message*, *sid* y *rid* respectivamente el mensaje, el servicio emisor y el identificador del mensaje respuesta.

Finalmente el método *sendAndWait* unifica las primitivas *send*, *wait* y *waitForReply* en una sola instrucción. Esta función envía el mensaje *message* hacia *sid* en nombre de *from* y espera un tiempo determinado a que haya una respuesta explícita de este mensaje, introduciéndolo en *message*.

Todas los métodos que han de esperar bien por una respuesta o mensaje (*waitForReply*, *confirm*, *sendAndWait*), bien por que haya espacio en el mailbox destino (*send*) tienen como parámetro un tiempo máximo a esperar (*timeout*) codificado en un entero largo con signo. El valor de este parámetro indica la disposición a esperar indefinidamente (valor 0),

[17] La elección de esta señal no es arbitraria, dado que es usada como señal de recepción de datos urgentes a través de un socket TCP. Dado que el nodo no usa la pila TCP del sistema operativo anfitrión, queda libre para nuestro uso, sin embargo sigue siendo usada por funciones POSIX como *select* o *poll*, y es útil para implementar coherentemente los servicios de entrada de datos.

no esperar (valor negativo) o esperar un número determinado de microsegundos (valor estrictamente mayor que cero).

Para completar esta sección dedicada al modelo de comunicaciones interna del nodo vea la δ7.10.2 en donde se puede encontrar una completa de la clase *Message*.

5 Evaluación del modelo

5.1 Introducción

En el capítulo 3 hemos diseñado un conjunto de filtros con el propósito de usarlos para disminuir los resultados obtenidos por los servidores de consultas (QS) en ámbitos de recuperación de información (IR) y así disminuir el ancho de banda y los tiempos de procesamiento necesarios para producir el resultado final esperado. Los capítulos 2 y 4 detallan conceptual y detalladamente la arquitectura de los nodos VAIN, dispositivos que han de ser dispuestos en el flujo de datos entre los QS y el *broker* de usuario, para albergar los filtros anteriormente expuestos y dotar así a la subred de una capacidad de procesamiento semántico o valor añadido, susceptible de alcanzar los objetivos planteados en el capítulo 1.

Este capítulo detalla los experimentos realizados y muestra los resultados, además de albergar el análisis de los mismos.

5.2 Diseño de los experimentos

Una vez explicados los conceptos en que nos basamos para diseñar los filtros IR, y habiendo obtenido una cuantificación teórica de la calidad en términos de eficiencia o complejidad temporal y espacial; pasaremos a describir los experimentos realizados y las consideraciones relativas al cálculo de los intervalos de tiempo.

5.2.1 Metodología de medición del tiempo

La medición de intervalos de tiempo entre dos sucesos en la ejecución de un programa es un proceso complicado, más aun cuanto menor sea el tiempo empleado en ejecutarse. Para medir el intervalo de tiempo transcurrido entre dos puntos cualesquiera de una secuencia de

código se toman dos muestras, una al inicio de la secuencia y otra al final, y se restan. La fiabilidad del resultado depende tanto de la resolución de la técnica empleada para medir el tiempo como de la escala esperada.

Otro aspecto a tener en cuenta, en sistemas operativos multitarea, es que el proceso convivirá seguramente con otros en un entorno de procesador compartido, esto es, la secuencia de código a medir con probabilidad será interrumpida y restaurada para permitir la sensación de que múltiples programas se ejecutan al mismo tiempo, más aún si en la secuencia existen llamadas al S.O. o de entrada/salida. Además de ello, en SS.OO. con memoria virtual, la ejecución de código puede conllevar la recuperación, desde el sistema de almacenamiento secundario, de páginas a las que se hace referencia y que no se encuentran en la memoria principal. Si bien sabemos que cuanto más pequeña es la secuencia a medir menos influencia tendrán estos hechos, habrá que tenerlos en cuenta a la hora de seleccionar una herramienta de medición.

En nuestro caso, dado que el S.O. anfitrión pertenece a la familia Unix y está basado en el *kernel* Linux 2.6, el cual no es un S.O. en tiempo real, disponemos básicamente de la herramienta *gettimeofday*, la cual tiene como máximo una resolución de microsegundos[1]. Nosotros supimos desde el principio que los tiempos esperados de los experimentos a realizar estarían en algunos casos por debajo de la escala de microsegundos, así pues necesitamos buscar un método con mayor resolución para llevar a cabo las mediciones.

Con la experiencia adquirida al desarrollar la herramienta RTASM (δ4.2) decidimos poner a prueba la resolución real de la función *gettimeofday* y contrastarla con la que podíamos obtener usando la instrucción ensamblador *rdtsc*.

En sus últimos procesadores la casa Intel [Intel, URL] ha proporcionado a los programadores de sistemas una instrucción que devuelve el números de *ticks* de reloj transcurridos desde el último arranque del procesador. Cada instrucción ejecutada en el procesador consume un número determinado y bien conocido de *ticks* (así pues podría decirse que realmente mide el número de instrucciones) cuantas más instrucciones posea una secuencia de código, mayor será el número de *ticks* consumidos, y mayor será el tiempo empleado.

[1] El estándar POSIX no especifica oficialmente una resolución estricta de *gettimeofday*, si bien sí su máxima resolución, dado que devuelve los segundos y microsegundos desde una fecha de inicio concreta.

La velocidad de un procesador, el numero de *ticks* por segundo, es un valor conocido; por tanto es posible, a partir del numero de *ticks* que consume una secuencia de código, conocer el tiempo empleado con una resolución máxima de $1/v$ segundos, con v la velocidad del ordenador en hercios.

El procesador empleado en los experimentos posee una velocidad de 2.8 GHz, en concreto[2] 2801859000 Hz, por tanto su resolución es de aproximadamente 3.57e10 segundos o 35.7 nanosegundos.

El código (ensamblador embebido del compilador GNU/GCC) encargado de leer este valor es el siguiente:

```
__inline__
unsigned long long int rdtsc(void)
{
 unsigned long long int x;
 __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
 return x;
}
```

El cual devuelve el número de *ticks* como un entero de 64 bits sin signo. Al declarar la función *inline*, conseguimos disminuir el número de instrucciones consumidas al realizar la llamada y con la optimización propia del compilador, esta puede considerarse despreciable[3].

Para poner a prueba esta metodología, hemos llevado a cabo el siguiente experimento: creamos una función (*spendTime*) que consume ciclos de reloj en una escala teóricamente cercana a los microsegundos, esta función es invocada 100000 veces obteniendo el mínimo, máximo y media de los tiempos consumidos por medio de las funciones *gettimeofday* y posteriormente *rdtsc*. Repitiendo este experimento 100 veces obtenemos las gráficas que podemos observar en la Ilustración 29 e Ilustración 30.

Los ejes de ambas gráficas se muestran el microsegundos, el máximo referenciado a la izquierda y tanto la media y el mínimo a la derecha. Las líneas que representan el máximo valor obtenido muestran los errores cometidos al medir no sólo la función *spendTime*, si no los debidos a cambios de contexto, interrupciones y fallos de página, y, aunque en magnitud muy similares, se aprecia una disminución considerable de estos al usar el método *rdtsc*. De hecho las medias de los máximos son, respectivamente, 24,860 frente a 8,689 microsegundos, para *gettimeofday* y *rdtsc*.

---

[2] Dato extraído del *kernel* Linux de la distribución usada.

[3] Al contrario que la llamada a la función *gettimeofday* que conlleva un gasto intrínseco de tiempo al realizar las mediciones, dado que es una función que realmente invoca una llamada, interrupción, al *kernel* Linux.

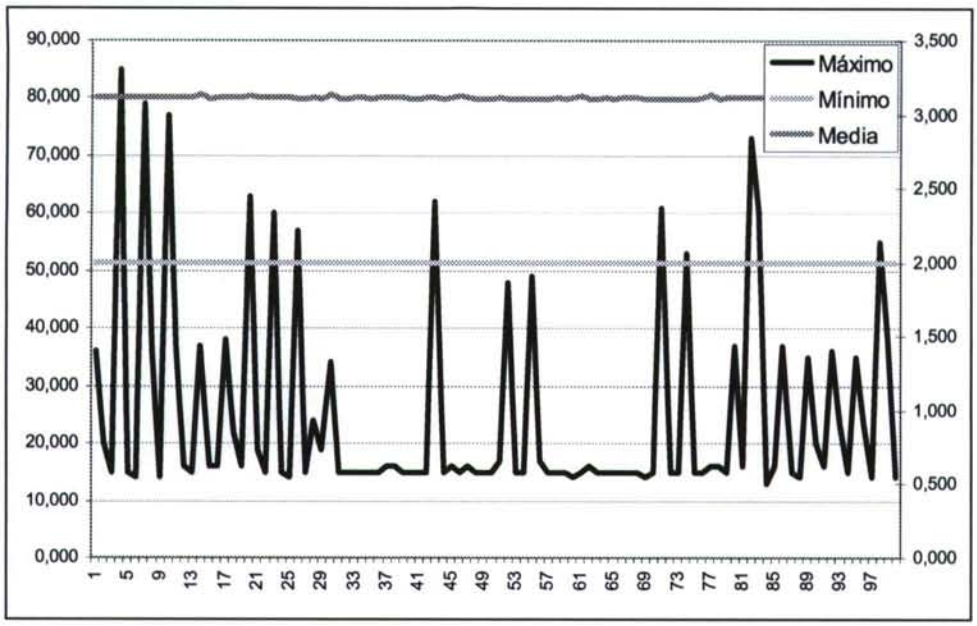

Ilustración 29: Medida de intervalos de tiempo con *gettimeofday*

De la misma manera podemos observar como *gettimeofday* produce un mínimo constante igual a 2 microsegundos, mientras que *rdtsc* oscila entre 0.355 y 0.360 microsegundos (0.358 de media), lo que nos lleva a suponer que *gettimeofday*, no alcanza una resolución inferior a los 2 microsegundos en esta implementación, mientras que *rdtsc* la supera ampliamente.

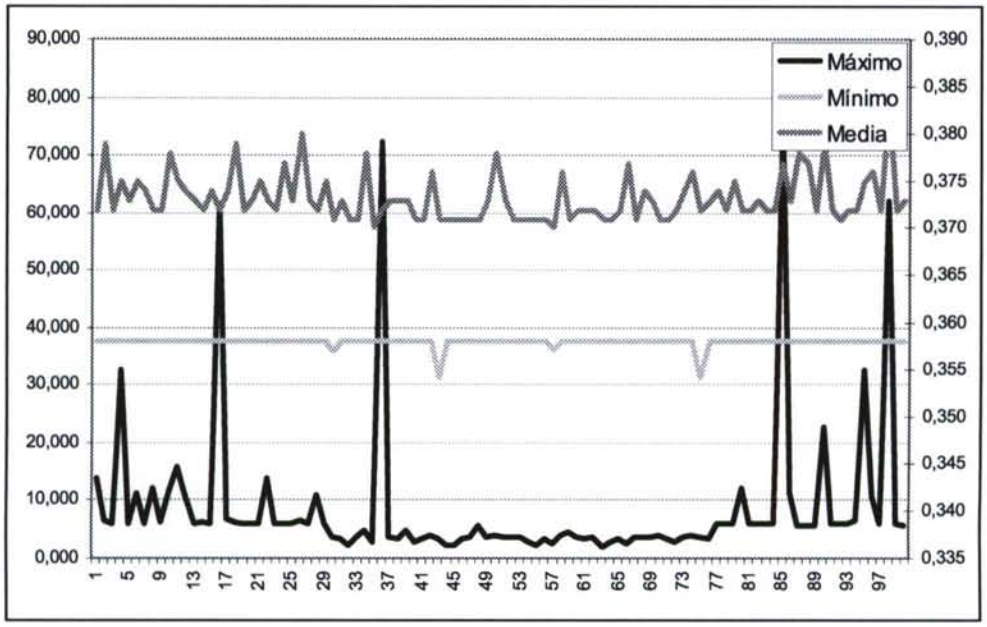

Ilustración 30: Medida de intervalos de tiempo con *rdtsc*

5.2.2 Descripción de los experimentos

La Ilustración 31 muestra la infraestructura empleada en los experimentos, en ella puede observarse como dos ordenadores, emisor y receptor, hacen las veces, respectivamente, de 1024 QS y *broker*. Ambos están unidos por una red en donde podemos encontrar dos segmentos *Ethernet* 100BaseT y un nodo VAIN, ejecutando los servicios *ip/ethernet* (δ7.11.2), *deploy* (δ7.11.1) y *fir*, este último implementando los filtros vistos en el capítulo 3.

El ordenador etiquetado como emisor, transmite hacia el denominado receptor 1024 paquetes UDP conteniendo cada uno de ellos 1000 respuestas, pares (documento, peso) de 8 bytes cada una de ellas. El nodo programable situado entre estos dos hace las veces de *router*, capturando, filtrando y retransmitiendo el resultado hacia el receptor, el cual comprueba que entre los documentos recibidos se encuentran los 1000 de mayor peso generados originalmente. Este proceso se repite 500 veces por cada filtro IR tomado en consideración.

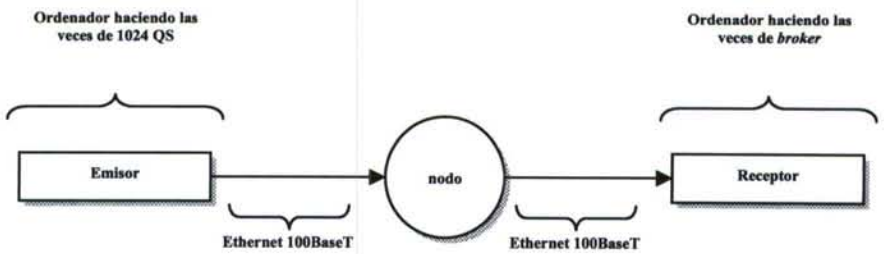

Ilustración 31: Esquema de la infraestructura de soporte de los experimentos

El emisor posee un fichero conteniendo un test de prueba, formado por los 1024x1000 documentos, de forma que es exactamente el mismo juego de datos el que pasa por el nodo para filtros diferentes. El test de prueba es generado aleatoriamente y distribuye los pesos de los documentos uniformemente. Así pues, antes de realizar los experimentos se conocen de antemano los 1000 mejores resultados, de esta manera el receptor puede comprobar la funcionalidad del filtro.

El número de filtros probados es de 12 (*noFilter*, *HardBroker*, *SoftBroker*, *SoftBroker*-BGFAST, *simple* BGFAST, *AcumulativeSL* BGFAST, *AcumulativeUL* BGFAST, *AcumulativeSU* BGFAST, *AcumulativeUU* BGFAST, *BinaryTree* y *BinaryTree* BGFAST)

y para los que incorporan un filtro BGFAST se prueba con el factor *f* de 0 a 10. En total 82 filtros[4].

Es en el nodo en donde se toman las medidas, las cuales son: número de paquetes de salida (q), número de documentos de salida (n_{out}), media del tiempo total empleado para filtrar 1024 paquetes en 500 repeticiones (t_f), media de tiempo total en retransmitir q paquetes (t_t); para los filtros basados en BGFAST, además se mide: documentos que entran en el filtrado (*in*), documentos que pasan positivamente el filtro (*out*), número de documentos que influyen en la heurística (H), número de documentos rechazados por el filtro debido a f_{th} y teniendo en cuenta a m_{th}.

En total el juego de pruebas para cada filtro generó 3.81 GB de tráfico, teniendo sólo en cuenta el tamaño de los paquetes IPv4 y despreciando la fragmentación, en total 348 GB; esto es, aproximadamente 8 horas de ejecución[5].

5.3 Resultados de las mediciones

En la Tabla 4 puede observarse los totales de todas las mediciones realizadas. La primera columna es el nombre del filtro, en el caso de estar basado en BGFAST se indica la escala (s), siempre 1, y el factor (f), además de los parámetros T y w que también son constantes para todos los experimentos, respectivamente 1000 y 8M.

Los grupos de columnas *factores* y *calidad* muestran los parámetros calculados: factor de filtrado (F_f), factor de tráfico (F_t), balance del filtro (r), eficacia (E_f), eficiencia (E_{ff}) y calidad (Q).

Existen algunas excepciones en esta tabla. Los filtros *AcumulativeSU* y *AcumulativeUU*, para el valor de *f* igual a 0 no han arrojado resultados para q, n_{out} y t_t, dado que generaban paquetes de tamaño superior a 64KB, sobrepasando el tamaño máximo de un datagrama UDP, por lo que sus respectivas celdas están vacías y por consiguiente no existen cálculos para los factores y parámetros de calidad.

[4] Realmente más, dado que el filtro *BrokerBinaryTree* fue probado sin y con control de balanceo. En los datos sólo incluiremos la versión realizada con árboles AVL, sin embargo en el análisis de los resultados incluiremos la versión sin control de balanceo.

[5] O lo que es lo mismo, el tiempo mínimo empleado en transmitir los resultados de 41000 consultas si sólo dispusiéramos de un segmento *Ethernet* conmutado y un único broker para procesar las respuestas.

	Complejidad temporal				Complejidad espacial
	α	β	δ	ψ	
NoFilter	p			n_{in}	
HardBroker	p	n_{in}	n_{in} $Log_2(n_{in})$	T	n_{in}
SoftBroker	p	$T^2H_{p,0.988}$		T	T
BrokerBinaryTree	p	$2Log_2(T)T$ $H_{p,0.988}$	T	T	T
Simple BGFAST	p	T/s $(H_{p+f}-H_f)$		T/s $(H_{p+f}-H_f)$	c+fT
Acumulative SL	p	T^2/s $(H_{p+f}-H_f)$		T/s $(H_{p+f}-H_f)$	c+(f+2)T
Acumulative UL	p	T/s $(H_{p+f}-H_f)$		T/s $(H_{p+f}-H_f)$	c+(f+2)T
Acumulative SU	p	T^2/s $(H_{p+f}-H_f)$		T/s $(H_{p+f}-H_f)$	c+fT+T/s $(H_{p+f}-H_f)$
Acumulative UU	p	T/s $(H_{p+f}-H_f)$		T/s $(H_{p+f}-H_f)$	c+fT+T/s $(H_{p+f}-H_f)$
HardBroker BGFAST	p	T/s $(H_{p+f}-H_f)$	T/s $(H_{p+f}-H_f)$ $Log_2(T/s$ $(H_{p+f}-H_f))$	T	c+fT+T/s $(H_{p+f}-H_f)$
SoftBroker BGFAST	p	T^2/s $(H_{p+f}-H_f)$		T	c+(f+1)T
BrokerBinaryTree BGFAST	p	$2Log_2(T)T/s$ $(H_{p+f}-H_f)$	T	T	c+(f+1)T

Tabla 3: Resumen de las complejidades temporales y espaciales de los filtros analizados

Nuestro interés en el análisis se centrará en la mejoría experimentada al incorporar un filtrado BGFAST a los filtros tradicionales vistos en el capítulo 3, sobre todo en términos temporales, esto es, el tiempo empleado en filtrar 1024 paquetes; aunque también estamos interesados en la eficacia de los filtros. En el análisis de estos datos tendremos además en cuenta la relación entre t_f y q para los filtros basados en BGFAST, dependiendo del parámetro *f*; también analizaremos la evolución de la calidad (Q) dependiendo de *f*.

Finalmente compararemos los resultados medidos con el cálculo teórico efectuado de complejidad temporal por filtro.

5.4 Análisis de los resultados

En la Tabla 3 tenemos el resumen de las complejidades temporales y espaciales de los filtros analizados. La complejidad temporal se muestra según sus factores dependientes y su factor independiente.

La metodología de análisis consistirá en comparar los filtros que hemos denominado tradicionales con sus respectivas versiones basadas en el filtro GFAT para, seguidamente, analizar la mejoría introducida al usar el algoritmo BGFAST con diferentes valores del parámetro *f*. También contrastaremos los resultados medidos con los teóricos en forma de complejidad temporal. La finalidad es demostrar la mejoría introducida por el uso del algoritmo BGFAST, en general, en contextos de recuperación de la información.

					BGFAST					Factores			Calidad		
Filtro	q	Nout	tf	tt	In	Out	H	fth	mth	Ff	Ft	r	Ef	Eff	Q
No Filter	1024	1E+06	227	11697	-	-	-	-	-	0,000	0,000	2%	0,000	0,56	0,000
Hard Broker	1	1000	174943	92	-	-	-	-	-	1,000	1,000	100%	1,000	0,78	0,776
Hard Broker															
BGFAST(1,0)	1	1000	1602	51	9282	8259	0	1023	0	1,000	1,000	97%	1,000	1,03	1,031
BGFAST(1,1)	1	1000	1558	51	9282	4278	0	5004	0	1,000	1,000	97%	1,000	1,03	1,034
BGFAST(1,2)	1	1000	1341	51	9282	2698	0	6584	0	1,000	1,000	96%	1,000	1,04	1,045
BGFAST(1,3)	1	1000	1275	51	9282	2120	0	7162	0	1,000	1,000	96%	1,000	1,05	1,048
BGFAST(1,4)	1	1000	1247	50	9282	1923	0	7359	0	1,000	1,000	96%	1,000	1,05	1,050
BGFAST(1,5)	1	1000	1247	50	9282	1864	0	7418	0	1,000	1,000	96%	1,000	1,05	1,050
BGFAST(1,6)	1	1000	1250	51	9282	1847	0	7435	0	1,000	1,000	96%	1,000	1,05	1,050
BGFAST(1,7)	1	1000	1243	51	9282	1844	0	7438	0	1,000	1,000	96%	1,000	1,05	1,050
BGFAST(1,8)	1	1000	1233	51	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,05	1,051
BGFAST(1,9)	1	1000	1378	55	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,04	1,042
BGFAST(1,10)	1	1000	1394	54	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,04	1,042
Soft Broker	1	1000	7592	50	-	-	-	-	-	1,000	1,000	99%	1,000	0,93	0,930
Soft Broker															
BGFAST(1,0)	1	1000	8020	51	9282	8259	82	505	436	1,000	1,000	99%	1,000	0,93	0,927
BGFAST(1,1)	1	1000	6443	51	9282	4278	0	5004	0	1,000	1,000	99%	1,000	0,94	0,940
BGFAST(1,2)	1	1000	4074	51	9282	2698	0	6584	0	1,000	1,000	99%	1,000	0,97	0,968
BGFAST(1,3)	1	1000	2931	55	9282	2120	0	7162	0	1,000	1,000	98%	1,000	0,99	0,990
BGFAST(1,4)	1	1000	2496	51	9282	1923	0	7359	0	1,000	1,000	98%	1,000	1,00	1,000
BGFAST(1,5)	1	1000	2385	51	9282	1864	0	7418	0	1,000	1,000	98%	1,000	1,00	1,004
BGFAST(1,6)	1	1000	2340	51	9282	1847	0	7435	0	1,000	1,000	98%	1,000	1,00	1,005
BGFAST(1,7)	1	1000	2357	51	9282	1844	0	7438	0	1,000	1,000	98%	1,000	1,00	1,004
BGFAST(1,8)	1	1000	2466	52	9282	1843	0	7439	0	1,000	1,000	98%	1,000	1,00	1,001
BGFAST(1,9)	1	1000	2621	52	9282	1843	0	7439	0	1,000	1,000	98%	1,000	1,00	0,997
BGFAST(1,10)	1	1000	2590	52	9282	1843	0	7439	0	1,000	1,000	98%	1,000	1,00	0,998
Simple BGFAST(1,0)	1024	8259	584	8106	9282	8259	0	1023	0	0,993	0,000	7%	0,496	0,66	0,328
Acumulative SL															
BGFAST(1,0)	9	8933	6794	369	9282	8259	0	1023	0	0,992	0,992	95%	0,992	0,73	0,727
BGFAST(1,1)	5	4289	4107	192	9282	4278	0	5004	0	0,997	0,996	96%	0,996	0,81	0,803
BGFAST(1,2)	3	2699	3047	115	9282	2698	0	6584	0	0,998	0,998	96%	0,998	0,87	0,864
BGFAST(1,3)	3	2120	2582	92	9282	2120	0	7162	0	0,999	0,998	97%	0,998	0,89	0,886
BGFAST(1,4)	2	1923	2342	94	9282	1923	0	7359	0	0,999	0,999	96%	0,999	0,92	0,919
BGFAST(1,5)	2	1864	2236	73	9282	1864	0	7418	0	0,999	0,999	97%	0,999	0,92	0,924
BGFAST(1,6)	2	1847	2191	72	9282	1847	0	7435	0	0,999	0,999	97%	0,999	0,93	0,926
BGFAST(1,7)	2	1844	2209	72	9282	1844	0	7438	0	0,999	0,999	97%	0,999	0,93	0,925
BGFAST(1,8)	2	1843	2324	72	9282	1843	0	7439	0	0,999	0,999	97%	0,999	0,92	0,922
BGFAST(1,9)	2	1843	2403	72	9282	1843	0	7439	0	0,999	0,999	97%	0,999	0,92	0,920
BGFAST(1,10)	2	1843	2400	72	9282	1843	0	7439	0	0,999	0,999	97%	0,999	0,92	0,920
Acumulative UL															
BGFAST(1,0)	9	8933	808	371	9282	8259	0	1023	0	0,992	0,992	69%	0,992	0,82	0,813
BGFAST(1,1)	5	4289	1096	192	9282	4278	0	5004	0	0,997	0,996	85%	0,996	0,87	0,865
BGFAST(1,2)	3	2699	1077	110	9282	2698	0	6584	0	0,998	0,998	91%	0,998	0,92	0,920
BGFAST(1,3)	3	2120	1062	67	9282	2120	0	7162	0	0,999	0,998	94%	0,998	0,94	0,936
BGFAST(1,4)	2	1923	1075	63	9282	1923	0	7359	0	0,999	0,999	94%	0,999	0,97	0,966
BGFAST(1,5)	2	1864	1090	62	9282	1864	0	7418	0	0,999	0,999	95%	0,999	0,97	0,968
BGFAST(1,6)	2	1847	1090	62	9282	1847	0	7435	0	0,999	0,999	95%	0,999	0,97	0,968
BGFAST(1,7)	2	1844	1114	61	9282	1844	0	7438	0	0,999	0,999	95%	0,999	0,97	0,967
BGFAST(1,8)	2	1843	1161	62	9282	1843	0	7439	0	0,999	0,999	95%	0,999	0,97	0,964
BGFAST(1,9)	2	1843	1188	62	9282	1843	0	7439	0	0,999	0,999	95%	0,999	0,96	0,963
BGFAST(1,10)	2	1843	1179	61	9282	1843	0	7439	0	0,999	0,999	95%	0,999	0,96	0,963
Acumulative SU															
BGFAST(1,0)	-	-	5448	-	9282	8259	0	1023	0	-	-	-			
BGFAST(1,1)	1	3208	10534	79	9282	4256	0	5056	0	0,998	1,000	99%	0,999	0,85	0,850
BGFAST(1,2)	1	2267	5477	69	9282	2618	0	6718	0	0,999	1,000	99%	0,999	0,90	0,902
BGFAST(1,3)	1	2119	4362	64	9282	2120	0	7162	0	0,999	1,000	99%	0,999	0,92	0,918
BGFAST(1,4)	1	1922	3554	62	9282	1923	0	7359	0	0,999	1,000	98%	1,000	0,94	0,936
BGFAST(1,5)	1	1864	3293	76	9282	1864	0	7418	0	0,999	1,000	98%	1,000	0,94	0,942
BGFAST(1,6)	1	1847	3206	62	9282	1847	0	7435	0	0,999	1,000	98%	1,000	0,95	0,945
BGFAST(1,7)	1	1844	3201	66	9282	1844	0	7438	0	0,999	1,000	98%	1,000	0,95	0,945
BGFAST(1,8)	1	1843	3255	69	9282	1843	0	7439	0	0,999	1,000	98%	1,000	0,94	0,944
BGFAST(1,9)	1	1843	3446	64	9282	1843	0	7439	0	0,999	1,000	98%	1,000	0,94	0,940
BGFAST(1,10)	1	1843	3410	65	9282	1843	0	7439	0	0,999	1,000	98%	1,000	0,94	0,941
Acumulative UU															
BGFAST(1,0)	-	-	947	-	9282	8259	0	1023	0	-	-	-			
BGFAST(1,1)	1	4278	1187	92	9282	4278	0	5004	0	0,997	1,000	93%	0,998	0,95	0,953
BGFAST(1,2)	1	2698	1155	70	9282	2698	0	6584	0	0,998	1,000	94%	0,999	0,99	0,985
BGFAST(1,3)	1	2120	1133	63	9282	2120	0	7162	0	0,999	1,000	95%	0,999	1,00	1,002
BGFAST(1,4)	1	1923	1128	61	9282	1923	0	7359	0	0,999	1,000	95%	1,000	1,01	1,010
BGFAST(1,5)	1	1864	1112	60	9282	1864	0	7418	0	0,999	1,000	95%	1,000	1,01	1,013
BGFAST(1,6)	1	1847	1128	59	9282	1847	0	7435	0	0,999	1,000	95%	1,000	1,01	1,012
BGFAST(1,7)	1	1844	1119	59	9282	1844	0	7438	0	0,999	1,000	95%	1,000	1,01	1,013
BGFAST(1,8)	1	1843	1114	59	9282	1843	0	7439	0	0,999	1,000	95%	1,000	1,01	1,013
BGFAST(1,9)	1	1843	1270	63	9282	1843	0	7439	0	0,999	1,000	95%	1,000	1,00	1,004
BGFAST(1,10)	1	1843	1274	63	9282	1843	0	7439	0	0,999	1,000	95%	1,000	1,00	1,004
Broker Binary Tree	1	1000	6349	51	-	-	-	-	-	1,000	1,000	99%	1,000	0,94	0,941
Broker Binary Tree															
BGFAST(1,0)	1	1000	6207	51	9282	7913	0	505	0	1,000	1,000	99%	1,000	0,94	0,942
BGFAST(1,1)	1	1000	4099	51	9282	4278	0	5004	0	1,000	1,000	99%	1,000	0,97	0,968
BGFAST(1,2)	1	1000	2789	51	9282	2698	0	6584	0	1,000	1,000	98%	1,000	0,99	0,993
BGFAST(1,3)	1	1000	2228	51	9282	2120	0	7162	0	1,000	1,000	98%	1,000	1,01	1,008
BGFAST(1,4)	1	1000	2058	51	9282	1923	0	7359	0	1,000	1,000	98%	1,000	1,01	1,014
BGFAST(1,5)	1	1000	1995	53	9282	1864	0	7418	0	1,000	1,000	97%	1,000	1,02	1,016
BGFAST(1,6)	1	1000	1949	58	9282	1847	0	7435	0	1,000	1,000	97%	1,000	1,02	1,018
BGFAST(1,7)	1	1000	1942	53	9282	1844	0	7438	0	1,000	1,000	97%	1,000	1,02	1,018
BGFAST(1,8)	1	1000	1940	51	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,02	1,018
BGFAST(1,9)	1	1000	1927	53	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,02	1,018
BGFAST(1,10)	1	1000	1927	65	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,02	1,018

Tabla 4: Resultados de las mediciones de los filtros IR

5.4.1 NoFilter

Este filtro posee una semántica nula, no obra procesamiento alguno sobre los paquetes que lo atraviesan y por supuesto, podemos esperar un tiempo de procesamiento por paquete constante, de hecho inferior al microsegundo (Ilustración 32), de media 0.452, con 78.371 μs empleados para retransmitir cada paquete[6].

El tiempo total para procesar los 1024 paquetes fue de 227 μs, empleando 11697 μs para retransmitirlos. Como es obvio el factor de filtrado es cero, al igual que el factor de tráfico, arrojando un balance de filtrado del 2%, esto es, consumió el 98% del tiempo retransmitiendo tráfico. Según nuestro criterio de calidad este filtro posee un valor de Q igual a cero.

La complejidad temporal de este algoritmo la calculamos (δ3.3.1) como:

$$O(\alpha p+\psi \sum_{i=1}^{p} p(i))$$

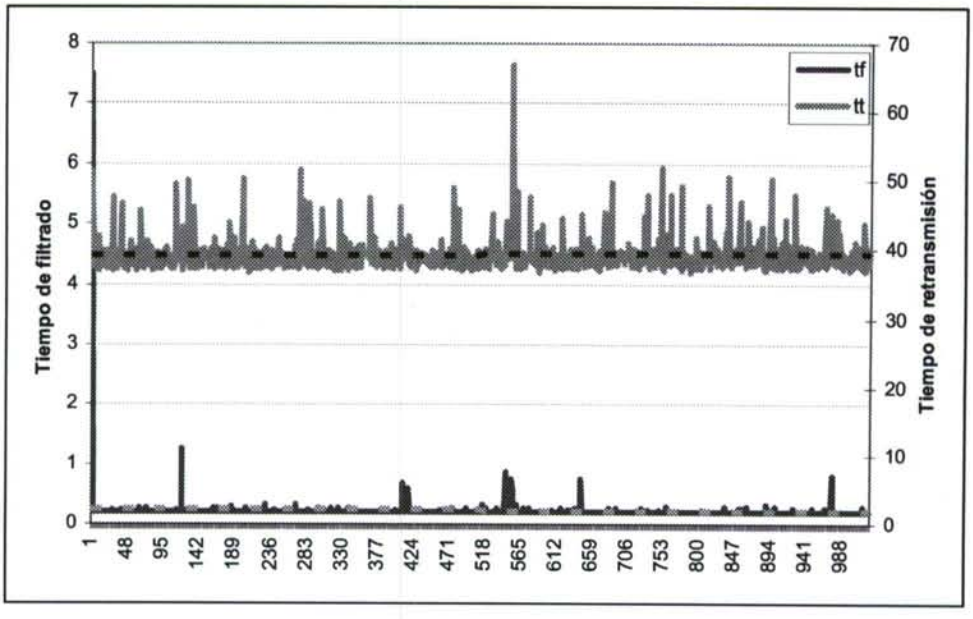

Ilustración 32: Evolución temporal del filtro NoFilter: $t_f(i)$ y $t_t(i)$

[6] Los tiempos de retransmisión de paquetes no serán, en general, muy fiables, dado que consisten en al menos una llamada de I/O manejada por el *kernel* del S.O., y siendo este un S.O. multitarea puede entregar durante la transmisión la ejecución a otro proceso del sistema, quedando este hecho reflejado en los tiempos.

Y dado que $p(i)$ es constante igual a T, y que p es igual a 1024, tenemos:

$$O(\alpha 1024+\psi \sum_{i=1}^{1024} T)=O(\alpha 1024+\psi 1024T)$$

Por tanto la mayor parte lo invierte retransmitiendo, lo que coincide con los datos medidos, en donde t_f es muy inferior a t_t.

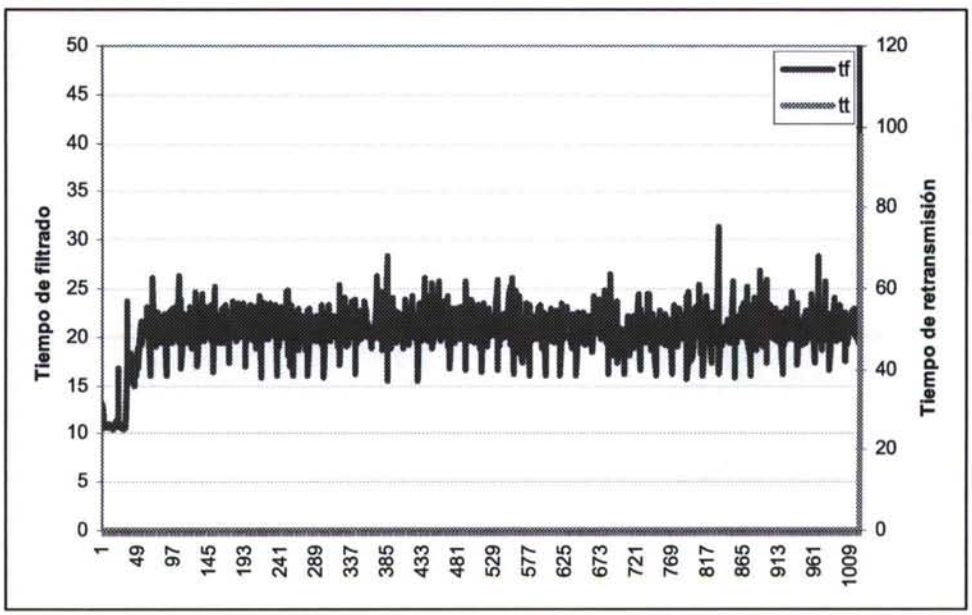

Ilustración 33: Evolución temporal del filtro HardBroker: $t_f(i)$ y $t_t(i)$

5.4.2 HardBroker

Este filtro está basado en almacenar, ordenar, truncar y retransmitir (δ3.3.2). El factor de filtrado y el de tráfico son 1, dado que emite un único paquete, al finalizar el proceso, conteniendo 1000 documentos. Su eficacia es la mayor posible, sin embargo su eficiencia no es muy buena, 0.78, dado que consume gran parte del tiempo total procesado los documentos (t_f): 174943 µs (~175 ms).

La Ilustración 33 muestra una evolución en el tiempo de este filtro, el tiempo de retransmisión es cero hasta la llegada del último paquete, en donde invierte 92 µs; según el balance del filtro (r) casi el 100% del tiempo lo invierte procesado los documentos del paquete, y este tiempo es consumido en su inmensa mayoría al final del procesamiento, cuando tiene que ordenar los 1024x1000 documentos que ha ido almacenando en las etapas previas.

Su complejidad temporal la habíamos definido como:

$$O(\alpha p+\beta n_{in} + \delta n_{in}\log_2(n_{in}) + \psi T)$$

Sustituyendo:

$$O(\alpha 1024+\beta 1024T+\delta 1024T\log_2(1024T)+\psi 1000) \approx O(\alpha 1024+\beta 1024T+\delta 20444963T+\psi T)$$

Según este resultado el mayor gasto temporal lo llevará acabo preparando (δ) los datos (ordenar y truncar) antes de ser retransmitidos (ψ).

5.4.3 SoftBroker

El filtro denominado *SoftBroker*, consiste en ir insertando los documentos de forma ordenada en una lista de tamaño máximo *T*. Al finalizar el procesamiento de todos los paquetes esta lista contendrá los *T* mejores documentos que el filtro ha canalizado.

Es una mejora respecto al filtro anterior, tanto en tiempo como en espacio usado. Este filtro es perfecto, posee un factor de tráfico y de filtrado igual a 1; y su eficiencia es superior al anterior (0.93). Su calidad la hemos calculado en 0.930.

La evolución en el tiempo ya no es lineal, tiende a cero debido a que al introducir un documento en una lista de tamaño limitado, si es rechazado y dado que los documentos en el paquete están ordenados de mayor a menor, sabemos que el resto del paquete será igualmente rechazado. Esto ocasiona que a medida que procesemos los paquetes, tardemos menos tiempo, y de hecho tendamos a cero a lo largo del tiempo.

Su complejidad temporal fue calculada como:

$$O(\alpha p+\beta T^2 H_{p,0.988}+\psi T)$$

Sustituyendo:

$$O(\alpha 1024+\beta 7805T+\psi T)$$

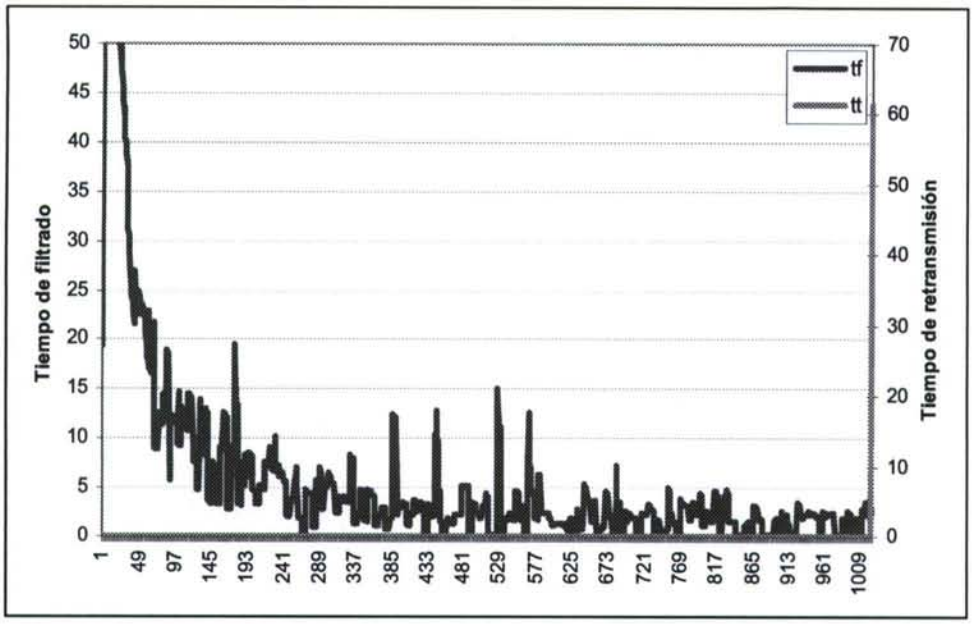

Ilustración 34: Evolución temporal del filtro SoftBroker: $t_f(i)$ y $t_t(i)$

Del más de un millón de documentos de entrada, realmente 7824 (~0.76% del total) son introducidos en la lista ordenada, el resto son descartados. Respecto al anterior, este filtro consume más tiempo procesando los paquetes (una 8 veces más), sin embargo ya no es necesario ordenar los documentos y el factor δ desaparece. Esto ocasiona que este filtro sea unas 23 veces más rápido que el anterior.

5.4.4 BrokerBinaryTree

Este filtro es un refinamiento del anterior donde, en vez de usar una lista limitada a T elementos, usamos un árbol que, igualmente, limitamos a T elementos.

Hereda los factores de tráfico y filtrado del anterior, y prácticamente es igual en calidad (0.928 frente a 0.930 del anterior).

El tiempo empleado en procesar toda su entrada es inferior a la del filtro *SoftBroker*, 6349 µs frente a 7592 µs. En la Ilustración 35 podemos ver como la evolución en el tiempo es muy similar.

Su complejidad temporal varía respecto al anterior filtro, disminuimos el tiempo empleado en añadir los documentos a una lista (ahora los añadimos a un árbol), pero introducimos un coste asociado a volcar los T mejores elementos del árbol a la lista que formará el paquete de salida:

$$O(\alpha p+\beta 2\mathrm{Log}_2(T)TH_{p,0.988}+(\delta+\psi)T) \approx O(\alpha 1024+\beta 156T+(\delta+\psi)T)$$

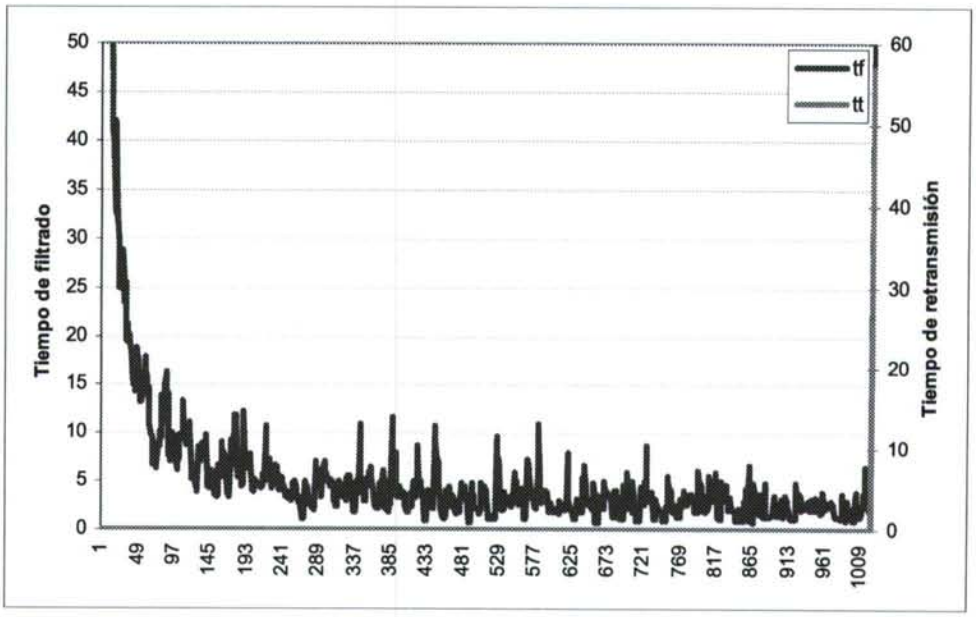

Ilustración 35: Evolución temporal del filtro BrokerBinaryTree: $t_f(i)$ y $t_t(i)$

Como podemos ver en las mediciones tomadas, ha supuesto una ligera ventaja el usar un árbol binario en vez de una lista limitada. El coste asociado a insertar los documentos en la estructura ha disminuido, sin embargo se ha visto parcialmente compensado con el coste de recorrer el árbol de mayor a menor, hasta *T* elementos, añadiéndolos en el paquete de salida.

Anteriormente (δ3.3.4) hemos explicado como el uso de árboles sin control de balanceo ocasionaba la realización de árboles degenerados, comportándose realmente como listas, con una complejidad temporal muy parecida a la del filtro *SoftBroker*:

$$O(\alpha p+\beta 7805T+(\delta+\psi)T)$$

Las mediciones han arrojado un tiempo de 7931 μs para este tipo de árboles, mientras que su versión usando listas consumió 7592 μs, confirmando los datos establecidos teóricamente.

5.4.5 Filtros basados en GFAT

En la sección anterior hemos mostrado el análisis temporal de los filtros denominados tradicionales. Esta sección analizará, en primer lugar, los filtros basados en GFAT *puros* para, posteriormente, llevar a cabo el análisis de las versiones GFAT de los filtros tradicionales vistos anteriormente.

5.4.5.1 Simple GFAT

Este filtro es la implementación del algoritmo *NoFilter* combinado con el algoritmo GFAT con el objetivo de reducir el número de documentos reenviados. Su estrategia consiste en filtrar cada paquete conteniendo T elementos y si este continua conteniendo documentos reenviarlo.

Podemos ver que este filtro no es determinista, su eficacia es de 0,496, mientras que su eficiencia se ha calculado en 0,66. Finalmente su calidad ha quedado establecida en 0,328, superior que su versión tradicional (con calidad 0,00).

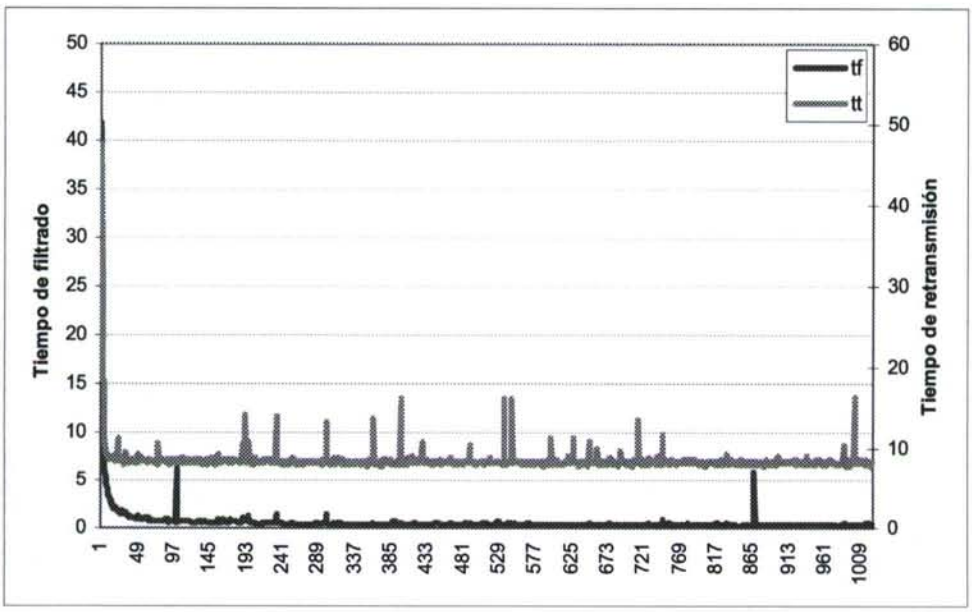

Ilustración 36: Evolución temporal del filtro SimpleGFAT: $t_f(i)$ y $t_t(i)$

A primera vista observamos como el balance del filtrado es del 7%, esto es, el 93% del tiempo lo emplea reenviando el tráfico. Así pues sus resultados arrojan un tiempo de 584 μs filtrando y 8106 μs retransmitiendo los 1024 paquetes con una media de 8 documentos por paquete, frente a los 1000 iniciales. Su complejidad temporal queda así:

$$O(\alpha+(\beta+\psi)TH_{1024}) \approx O(\alpha+\beta 8T+\psi 8T)$$

Según esto cabría esperar un tiempo de retransmisión muy inferior al medido para *NoFilter*. Las diferencias respecto a los tiempos de retransmisión de este filtro y su versión tradicional (8106 μs frente a 11697 μs) se explican observando la Ilustración 36, en donde podemos observar que independientemente del tamaño del paquete de salida el S.O. invierte retransmitiendo un mínimo de, aproximadamente, 8 μs.

En esta implementación hemos podido observar una ligera mejora introducida al usar el algoritmo GFAT. Los siguientes filtros de la familia *Acumulative*, son variaciones del que acabamos de ver. Variaciones con dos grados de libertad: limitando el número de paquetes transmitidos (*Limited/Unlimited*) y ordenando (o no) el contenido de los paquetes (*Sorted/Unsorted*).

5.4.5.2 AcumulativeSL GFAT

El primer filtro que analizaremos de la familia *Acumulative* es aquel que limita el tamaño del paquete de salida a T elementos y además lo mantiene ordenado (*Sorted, Limited*). Recordemos que este algoritmo, para mantener ordenado los paquetes de salida, mantiene una lista de tamaño finito $2T$, insertando en la misma de forma ordenada los documentos que pasan el filtro. Cuando esta lista supera el tamaño de T elementos, transmite un paquete de dicho tamaño y recorta la lista en T documentos. Como consecuencia de su funcionamiento este algoritmo forma un filtro no perfecto, con calidad medida en 0,727, dado que hemos mejorado el factor de tráfico (0,992 frente a 0) y empeorado ligeramente el factor de filtrado (0,992 frente a 0,993) respecto a *SimpleGFAT*. Las medidas arrojan que esta implementación transmite 8933 documentos en 9 paquetes, consumiendo para ello 369 µs. El tiempo de filtrado, como cabía esperar, ha aumentado a 6794 µs.

En la Ilustración 37 podemos observar los tiempos en que los paquetes son transmitidos como pulsos en la gráfica, más frecuentes cuanto más al principio del caudal de entrada. También es posible ver como los tiempo de filtrado de media disminuyen entre los pulsos debido a que el filtro avanza en su aprendizaje.

Su complejidad temporal queda establecida como:

$$O(\alpha p+(T\beta+\psi)TH_{1024}) \approx O(\alpha 1024+\beta 7509T+\psi 8T)$$

Como indica esta fórmula el tiempo de filtrado ha aumentado en una proporción de 1000 (T) respecto a *SimpleGFAT*, dado que ahora existe un coste inherente a mantener una lista finita ordenada. Respecto al tiempo de retransmisión, este puede llevarnos a engaño, ya que predice que consumirá el mismo tiempo que su antecesor. Sin embargo esto es consecuencia de que la fórmula está planificada en términos de documentos, y no de paquetes (ambos filtros emiten el mismo número de documentos, pero en un número diferente de paquetes). Sin embargo la calidad (a través del factor de tráfico) corrige esta situación, entregándonos una mejor estimación de la bonanza del filtro.

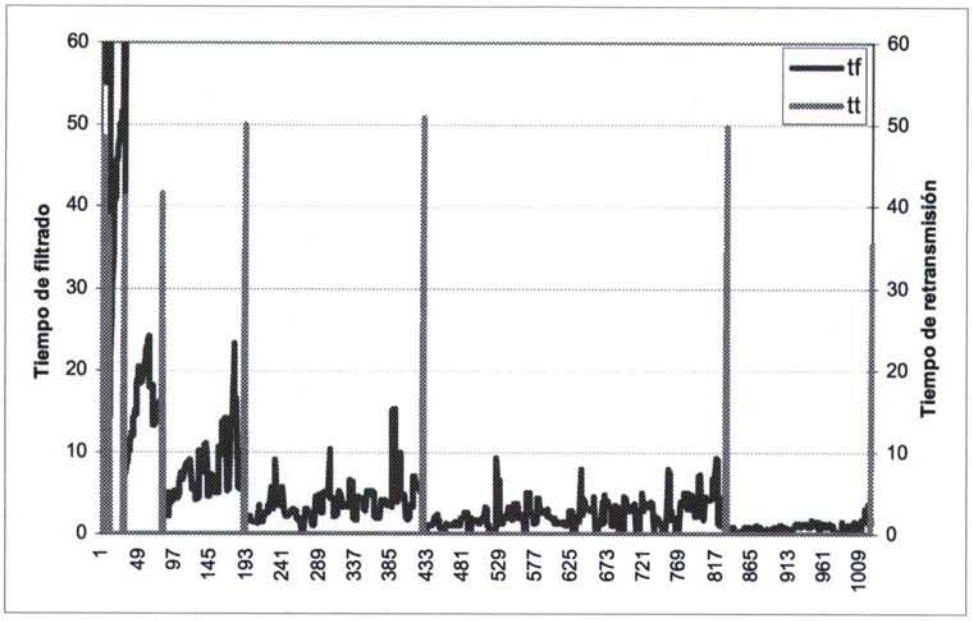

Ilustración 37: Evolución temporal del filtro AcumulativeSL GFAT: $t_f(i)$ y $t_t(i)$

5.4.5.3 AcumulativeUL GFAT

Respecto al anterior, este filtro limita igualmente el tamaño de los paquetes de salida, pero no mantiene el contenido de estos ordenado (*Unsorted, Limited*).

Respecto al anterior este filtro muestra una mejoría en su calidad (0,813 contra 0,727) debido al un aumento de su eficiencia (0,82 frente a 0,73). Esta eficiencia aumenta gracias a que ahora este filtro emplea el 69% del tiempo filtrando (el anterior empleaba el 95%) lo que representa 808 μs frente a 6794 μs.

Su complejidad temporal:

$$O(\alpha p+(\beta+\psi)TH_{1024}) \approx O(\alpha 1024+\beta 8T+\psi 8T)$$

Nos predice un tiempo de procesamiento (β) cercano al encontrado en *SimpleGFAT* y mejor que el calculado para *AcumulativeSL*, como confirman las medidas.

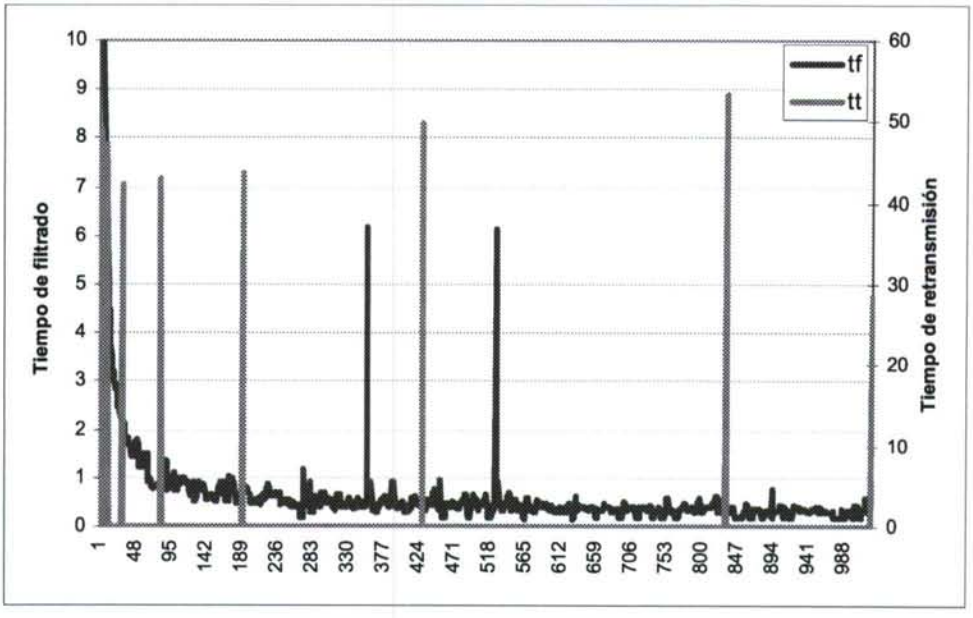

Ilustración 38: Evolución temporal del filtro AcumulativeUL GFAT: $t_f(i)$ y $t_t(i)$

5.4.5.4 AcumulativeSU GFAT y AcumulativeUU GFAT

Los dos filtros anteriores limitaban el tamaño de los paquetes de salida a *T* documentos por paquete. Los dos filtros que vamos a ver a continuación, en esta misma sección, no limitan el números de documentos por paquete, si no que permiten que este crezca hasta el tamaño máximo impuesto por el filtro (no por el protocolo de comunicaciones empleado). Ambos filtros emitirán, por tanto, un único paquete con todos los documentos que hayan pasado el filtro.

Sin embargo para el filtro GFAT (recordemos que es equivalente a BGFAST con el parámetro *f* igual a 0) y para el juego de entrada empleado, este arroja un resultado de 8259 documentos en un solo paquete, esto es 66080 bytes a transmitir, en este caso, en un datagrama UDP, el cual permite como máximo datagramas de 64KB. Esta situación desaparecerá cuando aumentemos el valor del parámetro *f*.

Por tanto lo único que podemos analizar por ahora de estos filtros es su tiempo de procesamiento, respectivamente 5448 µs y 947 µs, ambos muy similares a sus versiones limitadas (*Limited*) tal y como predijeron los cálculos de sus complejidades temporales respectivas[7] (6794 µs y 808 µs respectivamente).

[7] La complejidad temporal de ambos filtros son idénticas a sus versiones limitadas.

5.4.5.5 HardBroker GFAT

Con este filtro entramos en el análisis de las versiones GFAT de los filtros tradicionales. El primero en estudiar es la versión GFAT de *HardBroker*, el filtro basado en el almacenamiento masivo de los documentos y su posterior ordenación antes de reenviarlos.

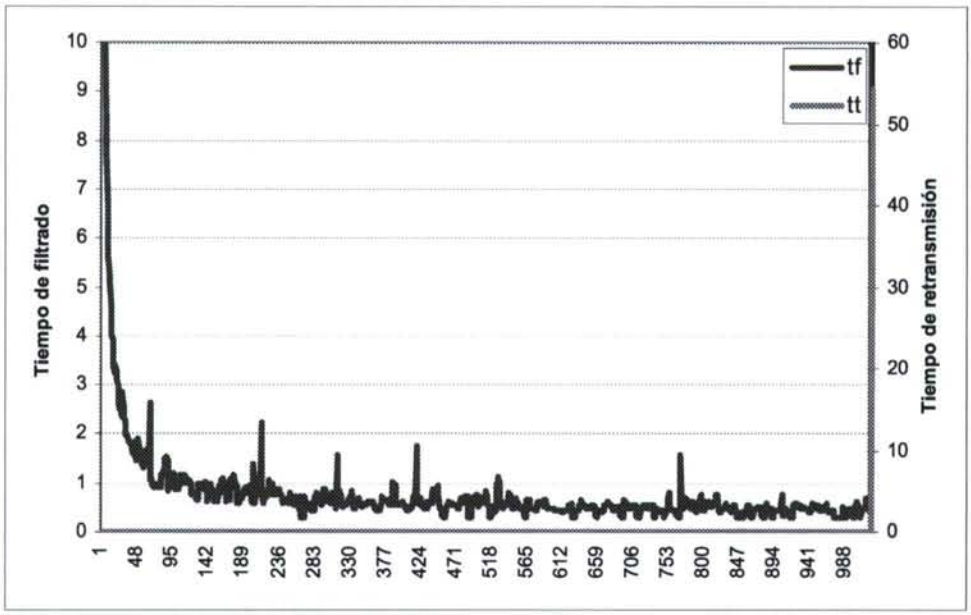

Ilustración 39: Evolución temporal del filtro HardBroker GFAT: $t_f(i)$ y $t_t(i)$

Podemos observar en la gráfica (Ilustración 39) como ahora el tiempo de filtrado ya no es constante (ver Ilustración 33) y disminuye a medida que avanzamos en el procesamiento de los paquetes de entrada. De hecho esta gráfica se asemeja a la vista para el filtro *SimpleGFAT* (Ilustración 36) dado que el comportamiento del filtro es idéntico en ambos casos.

El filtro GFAT a reducido el número de elementos a almacenar de 1024x1000 a 8259 (unas 124 veces) y por tanto el número final de documentos a ordenar. Este hecho se refleja en los tiempos de filtrado y preprocesamiento (1,6 ms frente a 175 ms) y queda registrado en el aumento de la calidad del filtro (1,031), siendo la mejor calidad vista hasta el momento.

La complejidad temporal del algoritmo *HardBroker* fue calculada aproximadamente como:

$$O(\alpha 1024+\beta 1024T+\delta 20444963T+\psi T)$$

Siendo la de su versión GFAT:

$$O(\alpha 1024+\beta 8T+\delta 8T\log_2(8T)+\psi T) \approx O(\alpha 1024+\beta 8T+\delta 103T+\psi T)$$

Muy superior a la de su predecesor, tal y como detallan las medidas realizadas.

5.4.5.6 SoftBroker GFAT

La versión GFAT de su filtro predecesor *SoftBroker*, aprovecha la disminución del caudal de documentos de entrada proporcionado por el filtro GFAT para disminuir el número de inserciones realizadas en la lista de tamaño finito *T*.

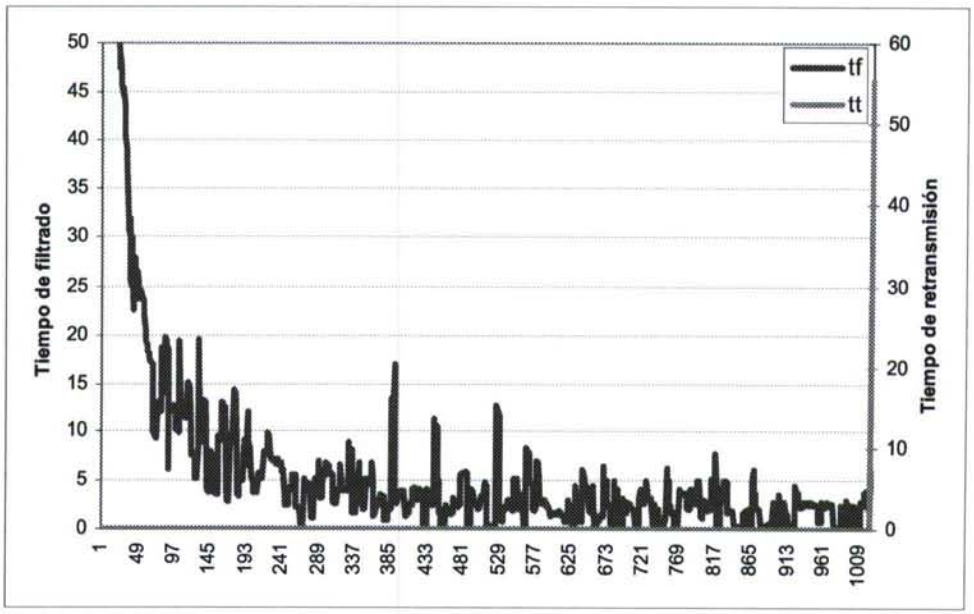

Ilustración 40: Evolución temporal del filtro SoftBroker GFAT: $t_f(i)$ y $t_t(i)$

Tal y como deducimos en la δ3.4.1esta disminución era irreal, dado que los documentos que el filtro GFAT eliminaba de la inserción eran precisamente, en su mayor parte, elementos que la propia inserción no tendría en cuenta. Así pues dedujimos que los tiempos (y en general la calidad) deberían ser muy parecidos, como así ha sido.

Según este juego de prueba, el filtro GFAT permite pasar 8259 documentos, mientras que *SoftBroker* realmente inserta en la lista 7824 elementos (435 documentos de diferencia). Además el primero es un superconjunto del segundo. Por lo tanto no existe ventaja significativa al incorporar un filtro GFAT a la implementación predecesora[8].

[8] De hecho es casi equivalente a filtrar dos veces con el mismo filtro GFAT.

Los tiempos de filtrado y la calidad son casi idénticos (8020 μs y 0.927 frente a 7592 μs frente a 0.930, respectivamente). La complejidad temporal fue calculada, igualmente, arrojando el mismo resultado (ver δ3.7.7).

5.4.5.7 BinaryTree GFAT

Recordemos que el filtro antecesor del actual no estaba basado en una mejoría del algoritmo de filtrado si no de otra forma de mantener una lista ordenada (factor β) consiguiendo así unos tiempos de inserción, búsqueda y borrado logarítmicos. Así pues los datos respaldan los cálculos realizados sobre este filtro y su antecesor, mostrando valores muy similares en tiempos y calidad, al igual (y por las mismas razones) que el filtro *SoftBroker* GFAT.

Sin embargo podemos mostrar la mejoría introducida al usar árboles AVL en vez de listas finitas. La calidad de este filtro ha aumentado (0,942) respecto a *SoftBroker* GFAT (0,927), debido a la eficiencia la cual aumenta debido a un menor tiempo de filtrado (6207 μs frente a 8020 μs). Su complejidad temporal mejora debido a la reducción del tiempo necesario para añadir (y eliminar) un elemento en el árbol:

$$O(\alpha p+\beta 2\mathrm{Log}_2(T)TH_{p,0.988}+(\delta+\psi)T) \approx O(\alpha 1024+\beta 156T+(\delta+\psi)T)$$

5.4.6 Filtros basados en BGFAST

Para finalizar el apartado de análisis de los datos esta sección detalla, de los resultados obtenidos para cada subfamilia de filtros, la bonanza en relación al parámetro *f* del filtro BGFAST. Hemos de recordar que los apartados anteriores se limitaban a analizar los resultados solamente para el valor 0 de dicho parámetro, o lo que es lo mismo, para el filtro de la familia GFAT. Este apartado relaciona los resultados obtenidos para su rango de valores de 0 a 10, y así podemos establecer una cadena de análisis entre los filtros tradicionales, BGFAT con $f = 0$ y BGFAT con f entre 0 y 10.

El análisis, basado por completo en gráficas, tiene dos puntos de vista: por una lado la relación de $t_f(i)$ y $q(i)$ en relación a f y por el otro la evolución de la eficacia, eficiencia y calidad, también en relación a f. Esta última está sometida por medio del umbral de eficiencia (ω), un valor arbitrario escogido de forma oportunista, para establecer una clasificación en términos de calidad. El valor escogido inicialmente para ω es de 6,4.

La relación entre t_f y q muestra la relación que existe entre el tiempo empleado filtrando el caudal de entrada y el resultado de dicho esfuerzo. La evolución de q a lo largo de los

diferentes valores que f va tomando (de 0 a 10) va a ser igual para todos los filtros analizados. Esto es debido al hecho de haber usado para todas estas pruebas el mismo juego de ensayo, consistente el 1024 paquetes conteniendo 1000 pares, identificación de documento y peso, con este último generado aleatoriamente. Cada paquete tiene su contenido ordenado de mayor a menor según su peso.

5.4.6.1 AcumulativeSL BGFAST

Para este filtro, en la Ilustración 41, podemos observar como q va disminuyendo paulatinamente a medida que incrementamos el valor de f, pasando de un valor igual a 8259 documentos de salida y estabilizándose en 1843 documentos para un valor de f igual a 8.

Recordemos que anteriormente hemos calculado el valor óptimo teórico de f como:

$$f_{opt} \approx \lceil (Ln(p) + \gamma) \rceil$$

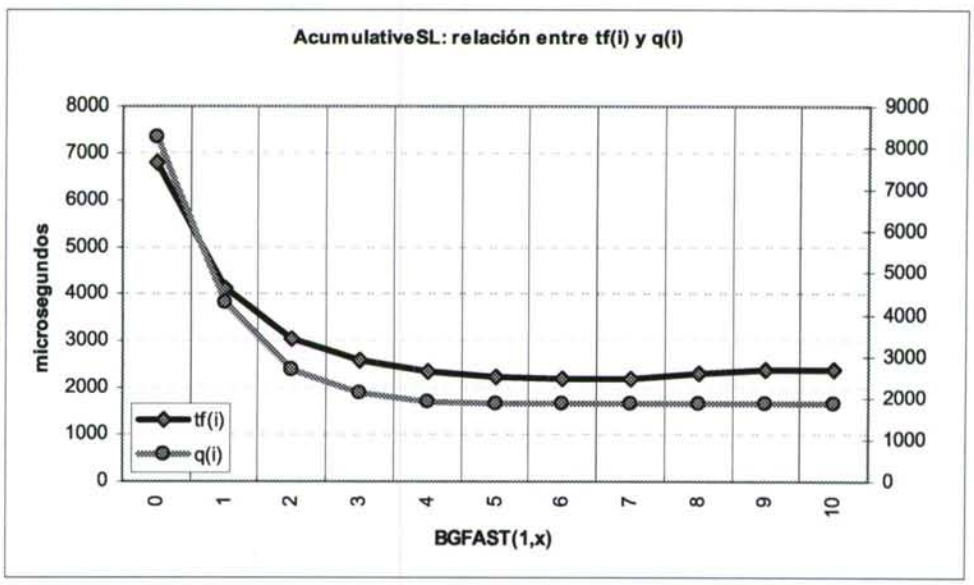

Ilustración 41: Evolución f del filtro AcumulativeSL BGFAST: t_f(i) y q(i)

El cual entrega un valor de f óptimo 8 para p = 1024 (véase la Tabla 4 para observar mejor como para $f \geq 8$ el filtrado no mejora y se mantiene estable).

Sin embargo el tiempo empleado filtrando tiene un repunte a partir del valor de f igual a 6, valor para el cual se alcanza el mínimo (2191 μs).

La eficacia de este filtro es cercana a 1, entre 0.992 y 0.999 dado que genera entre 9 y 2 paquetes.

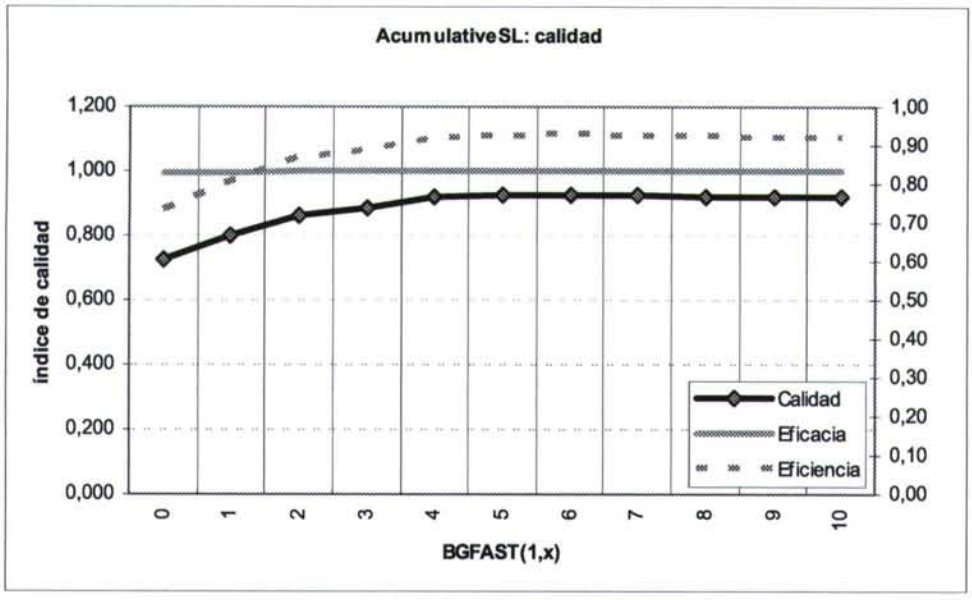

Ilustración 42: Evolución *f* del filtro AcumulativeSL BGFAST: parámetros de calidad

El valor máximo de eficacia lo alcanza a partir del valor 4 de *f*, sin embargo la eficiencia tiene sus picos en los valores 6 y 7 haciendo que la calidad sea máxima para *f* igual a 6.

5.4.6.2 AcumulativeUL BGFAST

De funcionamiento muy similar al anterior, este filtro se comporta respecto al tiempo de manera casi lineal.

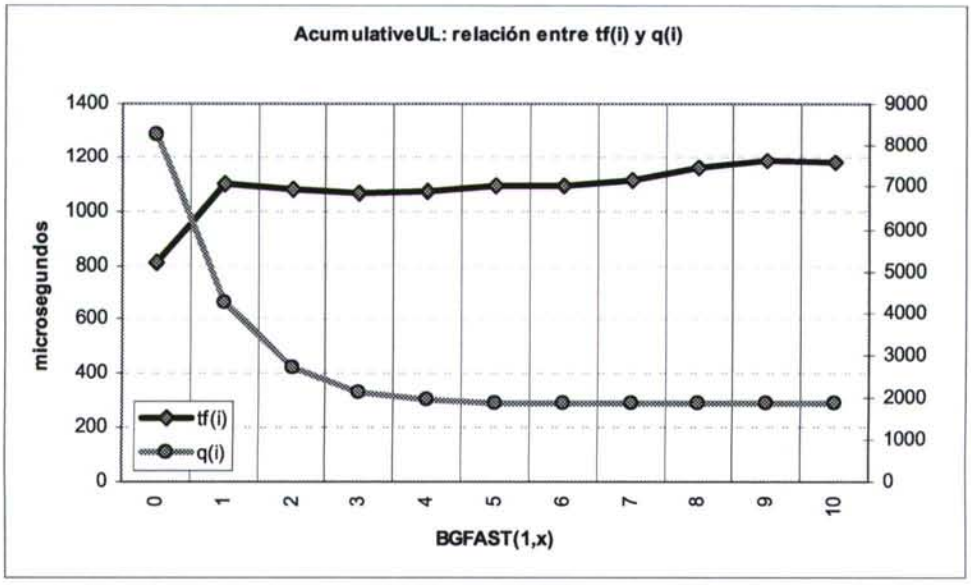

Ilustración 43: Evolución *f* del filtro AcumulativeUL BGFAST: t_f(i) y q(i)

El tiempo mínimo y máximo (808 μs y 1179 μs) empleado depende en menor medida del número de documentos efectivamente filtrados, dado que el valor relacionado con el factor β es muy pequeño.

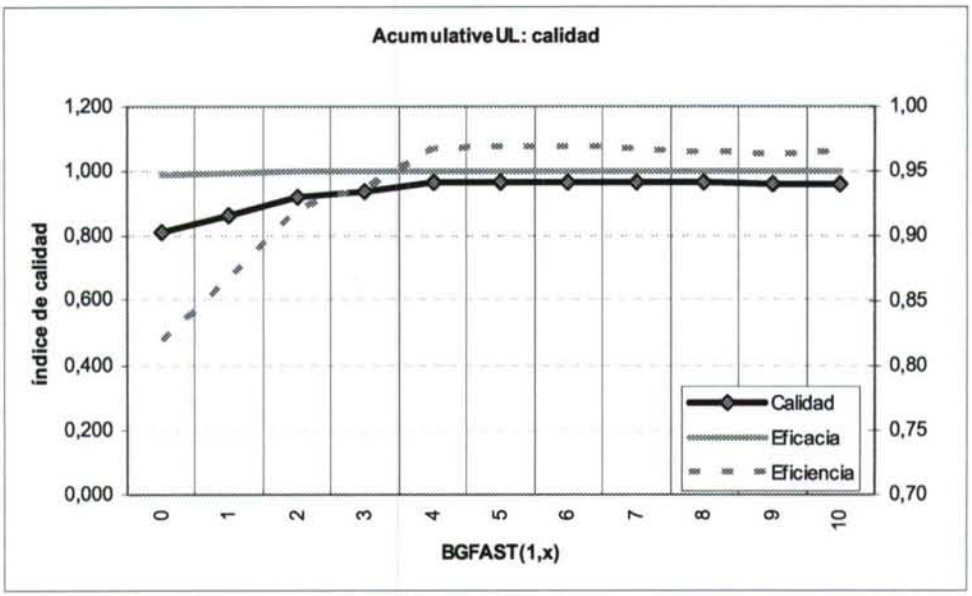

Ilustración 44: Evolución *f* del filtro AcumulativeUL BGFAST: parámetros de calidad

Esto significa que apenas se requiere más esfuerzo en filtrar mejor, al menos en relación al filtro anterior (y al siguiente).

La calidad alcanza su máxima para los valores de *f* igual a 5 y 6, en gran medida debido al aumento de la eficiencia (0.999) y en menor medida de la eficacia (0,97).

5.4.6.3 AcumulativeSU BGFAST

La evolución de t_f respecto a *f*, muestra como a medida que reducimos los documentos de salida, es necesario realizar un menor esfuerzo en términos de tiempo para insertar dichos elementos en lista con el objeto de mantenerla ordenada.

En general la curva t_f de la gráfica anterior es razonablemente similar para los filtros *Acumulative* de tipo *Sorted*, dado que requieren insertar los elementos que pasen el filtro en una lista finita y por tanto el esfuerzo temporal se reducirá a medida que menos documentos necesite añadir.

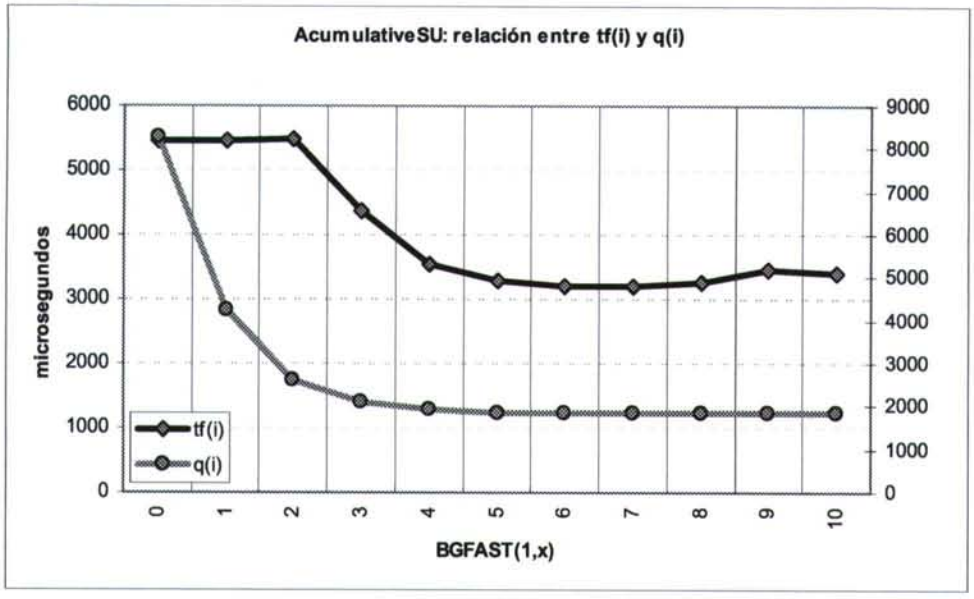

Ilustración 45: Evolución *f* del filtro AcumulativeSU BGFAST: t_f(i) y q(i)

La eficacia de este algoritmo es cercana a la unidad, dado que emite un único paquete (*Unlimited*) conteniendo entre 3208 y 1843 documentos. Respecto a su eficiencia, esta aumenta rápidamente, alcanzando un pico en los valores 6 y 7 de *f*.

Su calidad, modulada por la eficiencia y la eficacia, alcanza su máximo para los valores de *f* igual a 5 y 6, disminuyendo ligeramente a partir de este momento.

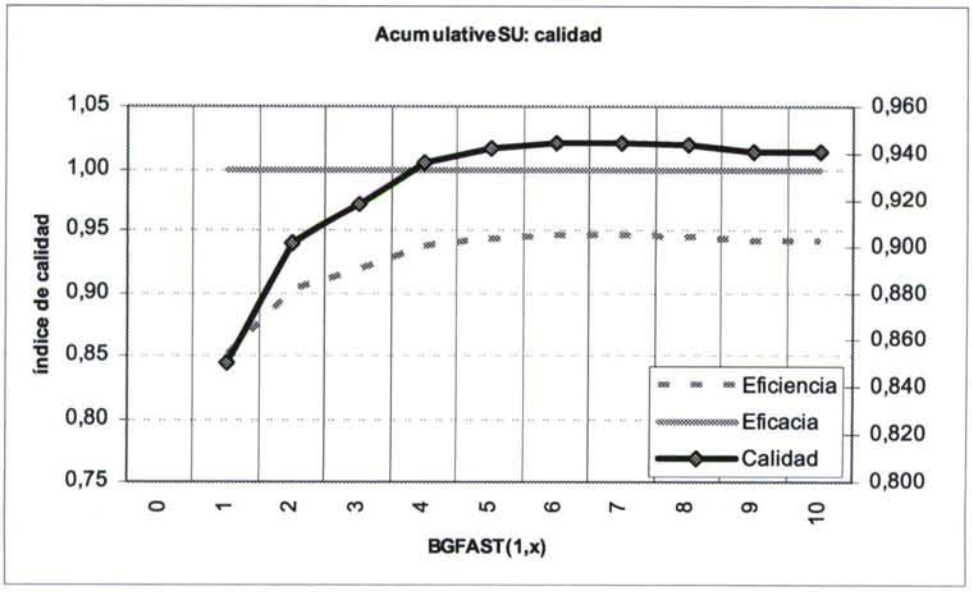

Ilustración 46: Evolución *f* del filtro AcumulativeSU BGFAST: parámetros de calidad

La razón de que la calidad (y la eficiencia) decaiga ligeramente a partir de un máximo hay que buscarla en el tiempo de filtrado, dado que comenzando con un valor de *f*, añadir más poder de filtrado en forma de un mayor valor de este, no supone mejora alguna en la eficacia, pero implica un ligero incremento en el número de instrucciones a ejecutar y por tanto de tiempo.

5.4.6.4 AcumulativeUU BGFAST

Para acabar el análisis de la familia *Acumulative*, mostramos los gráficos correspondientes al filtro *Unlimited/Unsorted*, el cual arroja los mejores tiempo de procesamiento emitiendo un único paquete conteniendo entre 4278 y 1843 documentos.

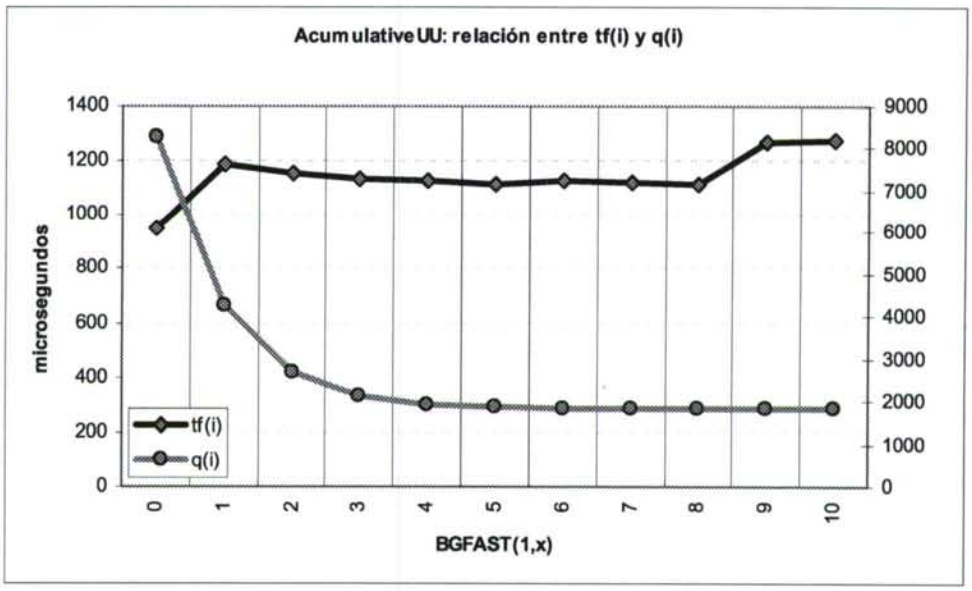

Ilustración 47: Evolución *f* del filtro AcumulativeUU BGFAST: $t_f(i)$ y q(i)

Como cabría esperar el tiempo es relativamente lineal respecto a la productividad del filtro, dado que el coste de procesar cada paquete es muy pequeño.

Sin embargo su calidad depende de otros factores, la eficacia aumenta hasta casi alcanzar la unidad y, por tanto, la calidad depende casi por completo de la eficiencia la cual, a su vez, depende del número de documentos de salida. Por esta razón la calidad es una curva con varios máximos para f igual al 5, 7 y 8 (para *f* igual a 6 es ligeramente inferior debido quizá a algún ruido en la muestra, debido a lo pequeño de estas).

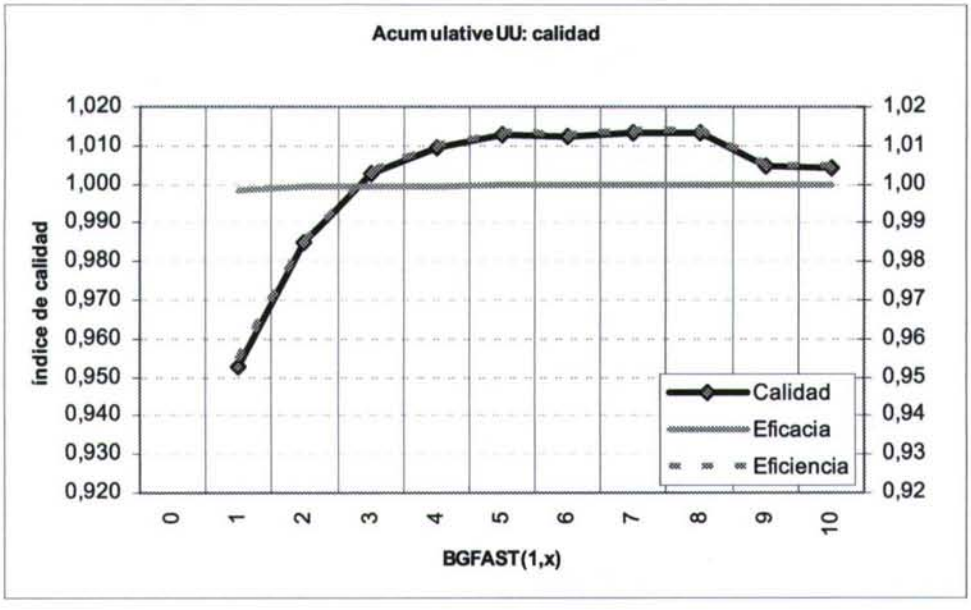

Ilustración 48: Evolución *f* del filtro AcumulativeUU BGFAST: parámetros de calidad

5.4.6.5 HardBroker BGFAST

Este filtro es el que ha arrojado, en términos de calidad, los mejores resultados, con un pico de 1,051 para el valor de *f* igual a 8.

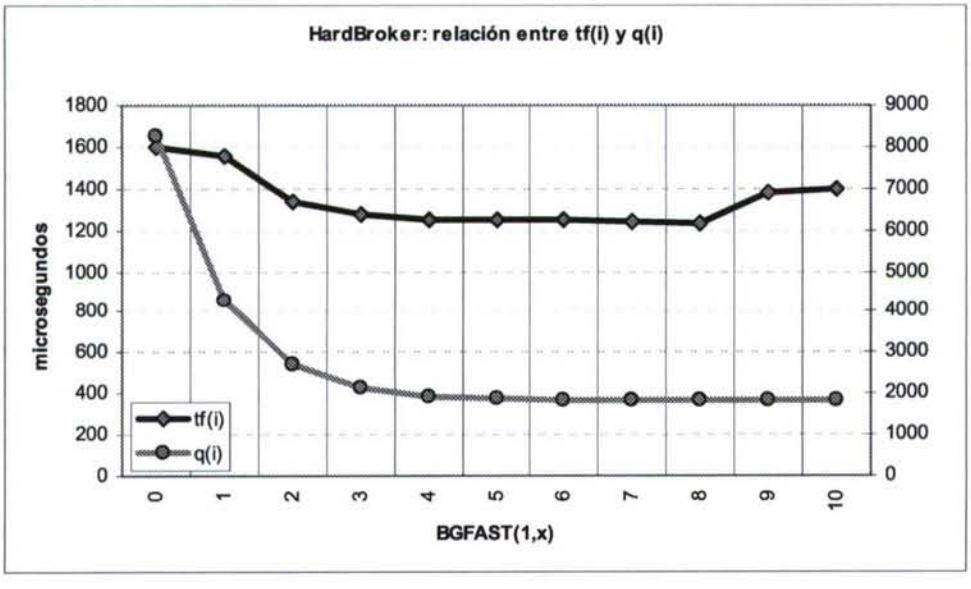

Ilustración 49: Evolución *f* del filtro HardBroker BGFAST: $t_f(i)$ y q(i)

El comportamiento del tiempo respecto a los valores que toma *f* es, globalmente hablando, lineal; moviéndose entre 1602 µs y 1233 µs, apenas 400 µs de diferencia. Estamos ante un filtro perfecto, que genera un único paquete de 1000 documentos.

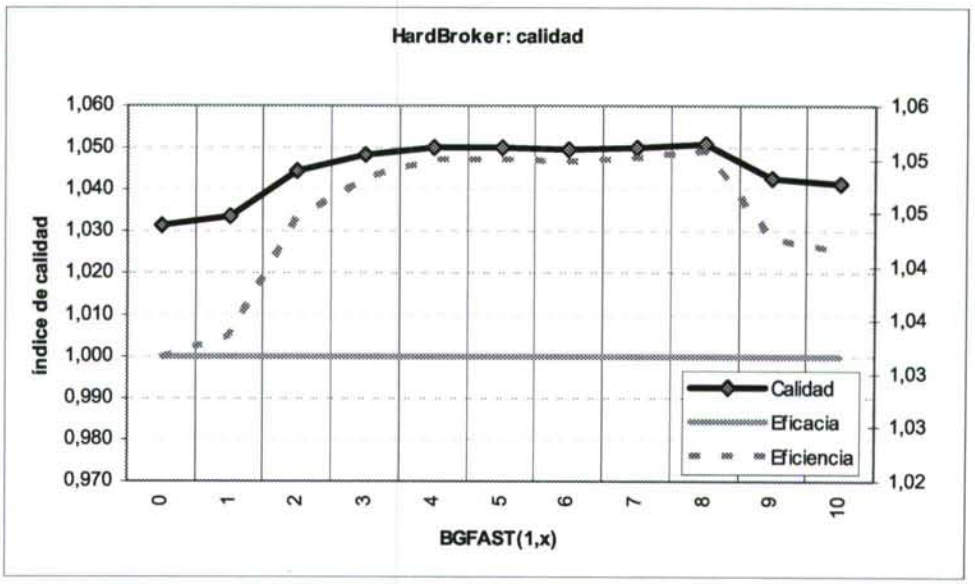

Ilustración 50: Evolución *f* del filtro HardBroker BGFAST: parámetros de calidad

Su eficacia es constante igual a 1 y la eficiencia varía acorde a la calidad; de hecho esta depende exclusivamente de la primera. La máxima calidad la encontramos para el valor, esperado, de *f* igual a 8. Y el algoritmo de ordenación *QuickSort*, en este caso, sólo ha de ordenar 1843 elementos, frente a los 1024x1000 originales. La reducción del tiempo total de procesamiento (t_f+t_t) es muy importante.

5.4.6.6 SoftBroker BGFAST

Las gráficas de este filtro y del siguiente (*BrokerBinaryTree* BGFAST) se comportan tal y como cabría esperar, se pueden considerar gemelas, dado que sólo difieren en la forma de mantener ordenados un conjunto de elementos (y por supuesto en sus medidas).

Recordemos como para *f* igual a 0, BGFAST (o lo que es lo mismo GFAT), no producía apenas mejora respecto a la versión tradicional del filtro. Esto era debido a que GFAT eliminaba del proceso aquellos documentos que de por sí eliminaría, sin necesidad de filtro alguno, el algoritmo de inserción en una lista de tamaño limitado a *T*.

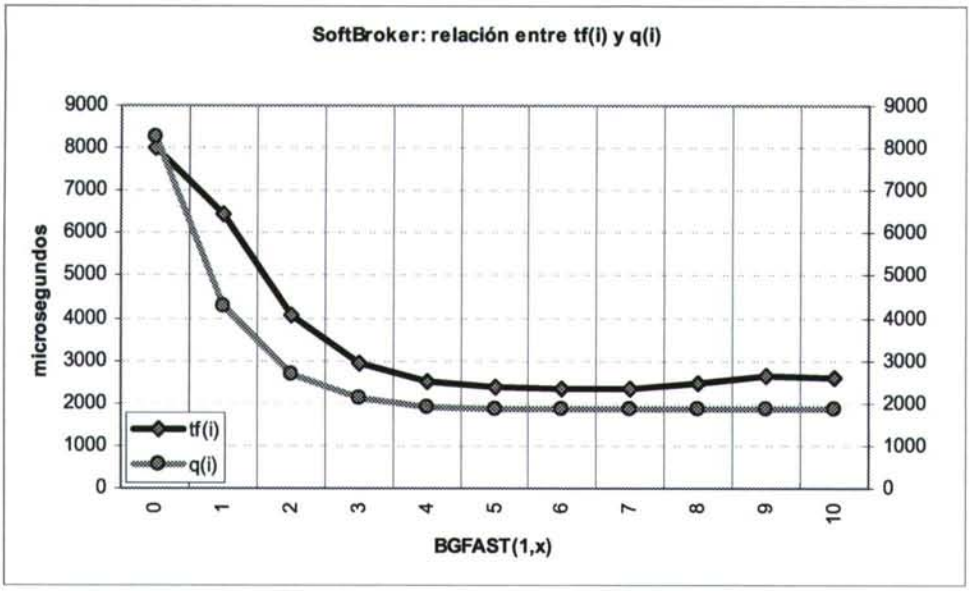

Ilustración 51: Evolución *f* del filtro SoftBroker BGFAST: t_f(i) y q(i)

Sin embargo al aumentar el valor de *f*, el filtrado aumenta su productividad y debemos insertar cada vez menos elementos. La evolución de t_f respecto a *f* demuestra esta situación, haciendo que a medida que incrementamos el filtrado, el tiempo necesario para procesar el caudal de entrada disminuya.

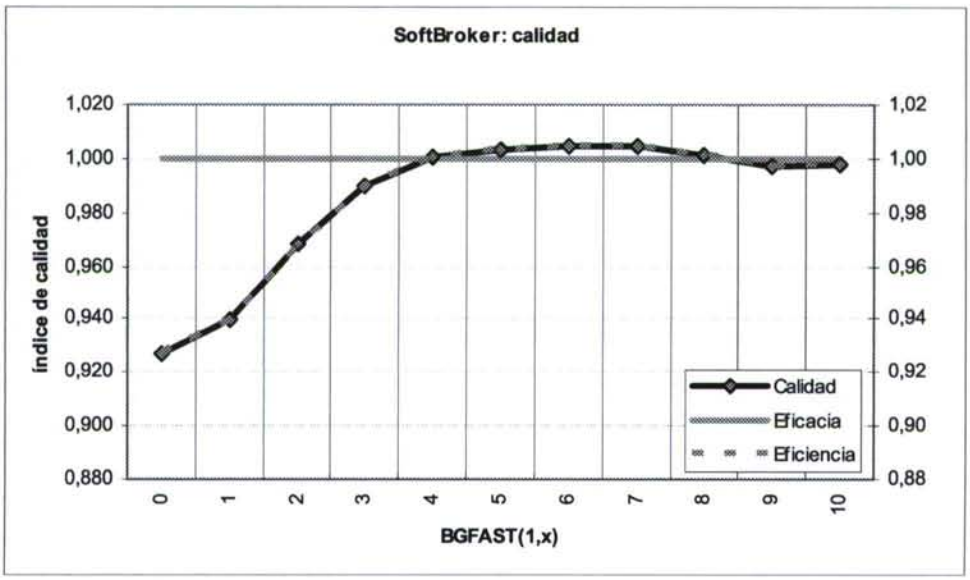

Ilustración 52: Evolución *f* del filtro SoftBroker BGFAST: parámetros de calidad

En términos temporales el mínimo lo encontramos para *f* igual a 6 (2340 µs), que se corresponde con una calidad e 1,005.

5.4.6.7 BinaryTree BGFAST

De manera muy similar al anterior este filtro presenta el perfil esperado, disminuyendo el tiempo necesario para procesar los paquetes de entrada a medida que aumenta la productividad del filtrado.

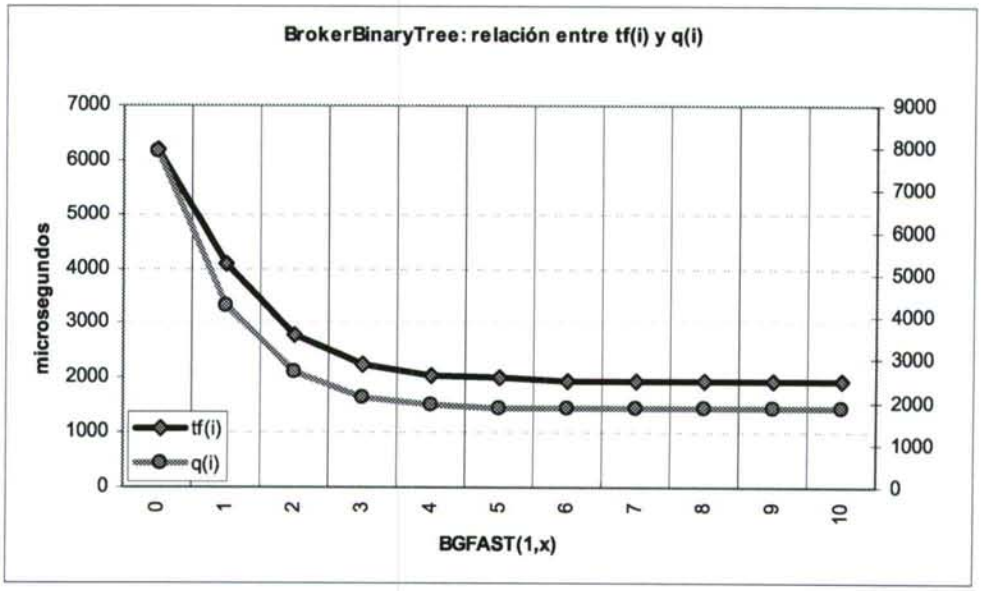

Ilustración 53: Evolución *f* del filtro BrokerBinaryTree BGFAST: $t_f(i)$ y q(i)

La calidad e este filtro parece estabilizarse para los valores de *f* igual a 9 y 10, alcanzando una calidad de 1,018 (1927 μs) y emitiendo un único paquete de 1000 documentos.

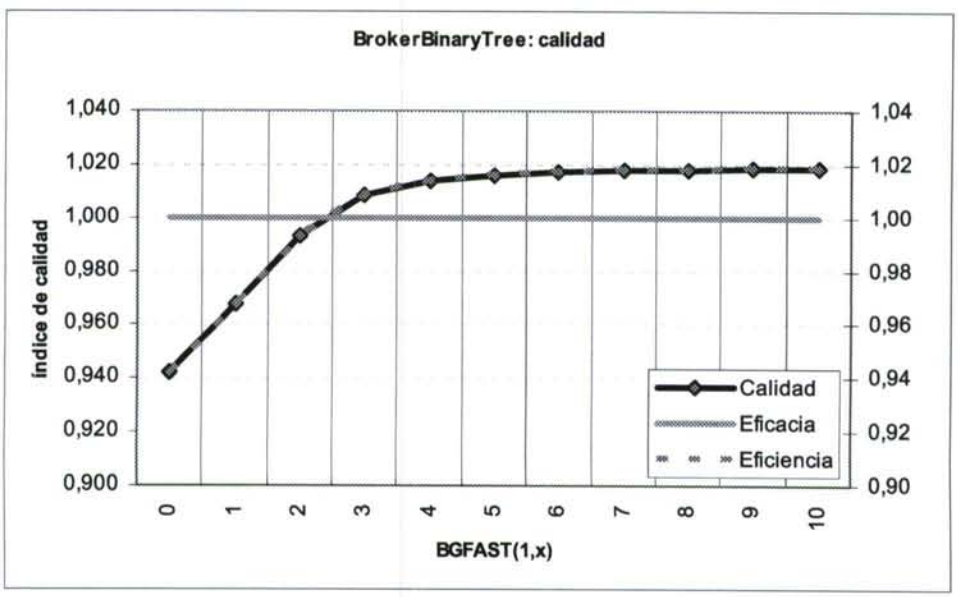

Ilustración 54: Evolución *f* del filtro BrokerBinaryTree BGFAST: parámetros de calidad

La calidad depende por entero de la eficiencia, dado que la eficacia es constante igual a 1. Esta es creciente y se estabiliza en los últimos valores de *f*.

6 Discusión, conclusiones y trabajo futuro

6.1 Introducción

El trabajo presentado en esta memoria ha sido orientado al estudio, análisis, diseño, implementación y prueba de un prototipo de nodo programable y de la lógica asociada al procesamiento embebido en el canal de respuesta a consultas en entornos de recuperación de información distribuida.

El punto de partida ha sido el trabajo de Cacheda, F. [Cacheda, 02], en donde se plantean los problemas inherentes al encauzamiento de las respuestas desde los QS a los *brokers* de usuario. En este y otros trabajos posteriores [Cacheda, 05] de los mismos autores, se detallan los cuellos de botella encontrados, destacando entre ellos dos: el propio canal de comunicaciones, y los *brokers* de usuario.

La solución que hemos planteado ha sido diseñar un modelo de filtrado en entornos de DIR usando técnicas de redes programables, o dispositivos de conmutación de paquetes de valor añadido embebido. No hemos encontrado trabajos de investigación relacionados con el estudio de la semántica en el canal de respuesta en ámbitos de IR distribuida[1], dado que los estudios existentes se centran en el canal de consulta y las técnicas y tecnologías implicadas en la obtención de la respuesta. Así pues consideramos el resultado de este trabajo novedoso y como una contribución al ámbito científico e investigador.

---

[1] Sin embargo sí existen estudios del rendimiento de las comunicaciones en el canal de respuesta tal y como indicamos en la introducción de este trabajo.

Las conclusiones se basan principalmente en los resultados obtenidos experimentalmente (capítulo 5), gracias al desempeño del nodo VAIN (capítulos 2 y 4) y de la arquitectura descrita como modelo de filtrado en ámbitos de IR (capítulo 3).

6.2 Discusión

Los datos mostrados en la Tabla 4 han sido calculados con un umbral de eficiencia (ω) igual a 6.4; gracias a este umbral podemos posicionar a voluntad el valor de la calidad (Q) de los filtros, permitiendo un corte por encima e igual y por debajo del valor unidad. En dicha tabla 32 filtros obtuvieron un valor igual o por encima de 1.

Para restringir el conjunto de los mejores filtros hemos modificado el valor de ω, haciéndolo igual a 6,3, con lo que obtenemos diferentes valores para la calidad (aunque manteniendo las relaciones) y así obramos un corte donde se seleccionan los mejores 17 filtros (ver Tabla 5).

					BGFAST					Factores			Calidad		
Filtro	q	Nout	tf	tt	In	Out	H	fth	mth	Ff	Ft	r	Ef	Eff	Q
HardBroker															
BGFAST(1,0)	1	1000	1602	51	9282	8259	0	1023	0	1,000	1,000	97%	1,000	1,02	1,015
BGFAST(1,1)	1	1000	1558	51	9282	4278	0	5004	0	1,000	1,000	97%	1,000	1,02	1,017
BGFAST(1,2)	1	1000	1341	51	9282	2698	0	6584	0	1,000	1,000	96%	1,000	1,03	1,028
BGFAST(1,3)	1	1000	1275	51	9282	2120	0	7162	0	1,000	1,000	96%	1,000	1,03	1,032
BGFAST(1,4)	1	1000	1247	50	9282	1923	0	7359	0	1,000	1,000	96%	1,000	1,03	1,034
BGFAST(1,5)	1	1000	1247	50	9282	1864	0	7418	0	1,000	1,000	96%	1,000	1,03	1,034
BGFAST(1,6)	1	1000	1250	51	9282	1847	0	7435	0	1,000	1,000	96%	1,000	1,03	1,033
BGFAST(1,7)	1	1000	1243	51	9282	1844	0	7438	0	1,000	1,000	96%	1,000	1,03	1,034
BGFAST(1,8)	1	1000	1233	51	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,03	1,034
BGFAST(1,9)	1	1000	1378	55	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,03	1,026
BGFAST(1,10)	1	1000	1394	54	9282	1843	0	7439	0	1,000	1,000	96%	1,000	1,03	1,025
BrokerBinaryTree															
BGFAST(1,5)	1	1000	1995	53	9282	1864	0	7418	0	1,000	1,000	97%	1,000	1,00	1,000
BGFAST(1,6)	1	1000	1949	58	9282	1847	0	7435	0	1,000	1,000	97%	1,000	1,00	1,002
BGFAST(1,7)	1	1000	1942	53	9282	1844	0	7438	0	1,000	1,000	97%	1,000	1,00	1,002
BGFAST(1,8)	1	1000	1940	51	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,00	1,002
BGFAST(1,9)	1	1000	1927	53	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,00	1,002
BGFAST(1,10)	1	1000	1927	65	9282	1843	0	7439	0	1,000	1,000	97%	1,000	1,00	1,002

Tabla 5: Selección de los mejores filtros según su calidad (ω=6,3)

Por familias podemos observar como *HardBroker* combinado con BGFAST ha obtenido la máxima calidad, seguido por la versión filtrada del *broker* implementado con árboles AVL (*BrokerBinaryTree*).

El filtro que podemos considerar óptimo es *HardBroker* BGFAST con un valor de *f* igual a 8 (para 1024 QS) y una calidad relativa calculada en 1.03420 (para ω=6.3)

La mejora de este filtro, comparada con la de su versión tradicional *HardBroker*, es elevada, 1.2 ms frente a 175 ms, más de 140 veces más rápido, con un coste en términos de memoria de aproximadamente 52 KB. La versión tradicional consumiría hasta 8 MB de memoria en las mismas condiciones. Tal y como podemos constatar en las complejidades temporales de ambos algoritmos (δ3.3.2 y δ3.7.6 respectivamente), la mejoría se ha debido, sobre todo, a la reducción de elementos a ordenar por el algoritmo de ordenación empleado.

La mejoría es aún importante al comparar el mejor resultado obtenido usando algoritmos tradicionales y el mejor filtro usando BGFAST. El denominado *BrokerBinaryTree* usando árboles AVL arrojó una calidad de 0.941 frente a *HardBroker* BGFAST con $f = 8$ que obtuvo una calidad de 1.051. En términos temporales el primero ha invertido un total de 6.4 ms frente a 1.2 ms, esto es, unas 5 veces más rápido. Por lo que queda patente la mejoría introducida por el uso del algoritmo BGFAST.

En la introducción de esta tesis (capítulo 1) declaramos como objetivo principal el de minimizar el término t_i mediante la reducción del término $tr_{i,j}$ (respuestas producidas por el QS j para la consulta i) presente en la siguiente fórmula (para 1024 QS):

$$t_i = \max_{j=1}^{1024}(t_{i,j}) + \max_{j=1}^{1024}(ra_{i,j}) + tc(\sum_{j=1}^{1024} tr_{i,j})$$

Suponiendo el uso de BGFAST(T, w, s=1, f) sólo en el *broker* de usuario (modelo de brokers centralizados) tenemos que ahora el tiempo empleado para procesar la consulta es:

$$t_i = \max_{j=1}^{1024}(t_{i,j}) + \max_{j=1}^{1024}(ra_{i,j}) + tc(T(H_{p+f} - H_f))$$

Dado que hemos reducido el juego de elementos que el *broker* debe ordenar. Esto significa que hemos alcanzado una solución que reduce considerablemente uno de los dos cuellos de botella, los *brokers* de usuario, dado que reducimos considerablemente el número de elementos que este debe procesar para obtener la respuesta definitiva.

Al situar, al menos, una capa formada por n nodos VAIN (modelo de filtros distribuido) implementando el filtrado *HardBroker* BGFAST (Ilustración 2) conseguimos, por una parte segmentar el tráfico en n caudales de respuesta, aumentando el ancho de banda total a disposición de los QS en aproximadamente la misma proporción. Profundizando el cálculo del tiempo total empleado, supongamos un canal simétrico de respuesta de orden n, en donde existe una etapa intermedia de filtrado formada por n nodos y de cada uno de ellos dependen n servidores de consulta (el número total de QS es por tanto n^2). Por tanto ahora tenemos que:

$$t_i^{BGFAST} = \max_{k=1}^{n}(\max_{j=n(k-1)+1}^{n(k-1)+n}(t_{i,j}) + \max_{j=n(k-1)+1}^{n(k-1)+n}(ra_{i,j,k}) + tc(r_{s,n,f}) + rn_{i,k}) + tc(r_{s,n,f})$$

En donde $t_{i,j}$ es el tiempo empleado por el QS j en procesar la consulta i; $ra_{i,j,k}$ es el tiempo empleado por el QS j en mandar el resultado parcial de la consulta i al nodo k; y $rn_{i,k}$ es el tiempo empleado en mandar, para la consulta i, el resultado parcial producido por el nodo k; y, finalmente, $r_{s,n,f}$ es:

$$r_{s,n,f} = \frac{T}{s}(H_{n+f} - H_f)$$

Esto es, la estimación de documentos que un filtro BGFAST(T, w, s, f) permite pasar después de procesar n paquetes con T documentos cada uno de ellos, usando w como ancho de ranura.

Las medidas mostradas en la sección 5.3 no son susceptibles de ser extrapoladas a este ámbito y, por razones obvias, es muy difícil su implementación, así pues hemos recurrido al trabajo de Cacheda et al. [Cacheda, 05] los cuales han accedido a modificar el simulador que describen en sus trabajos con el objetivo de medir, y comparar, la eficiencia de un modelo centralizado frente a la eficiencia del modelo distribuido descrito en los párrafos anteriores. Dichos autores muestran que sus experimentos sobre el modelo centralizado simulando 1024 QS con uno o más brokers, arrojan un *throughput* de 0.68371 respuestas por segundo, lo que en términos temporales significa, en el mejor de los casos, 4286.52 ms por consulta (tiempo de respuesta).

Una vez modificada la simulación para adaptarla a una arquitectura distribuida con 32 filtros, cada uno de los cuales gestiona el tráfico de 32 QS (1024 en total), el modelo arroja un *throughput* de 0.70192 respuestas por segundo, que en términos temporales representa 1584.26 ms, por tanto un 64% de mejora del modelo de filtros distribuidos respecto al modelo de brokers centralizados.

Hemos citado anteriormente en la introducción que la principal ventaja al usar nodos programables es la transparencia, dado que estos dispositivos activos de la subred entregan como valor añadido de su funcionalidad el procesamiento del tráfico que los atraviesa. La ventaja secundaria derivada a partir de esta, es la facilidad de modificación de la topología de la red. Esta tiene una especial importancia en el canal de respuesta de una arquitectura de recuperación de información.

Anteriormente hemos puesto como ejemplo de infraestructura distribuida un canal de respuesta simétrico unicapa, formado por una única capa de n nodos, encargado cada uno de ellos de procesar el tráfico de n servidores de consulta (32x32 para 1024 QS). Esta

topología simétrica es, a priori, la óptima si suponemos (como así lo hemos hecho a lo largo de este trabajo) que cada QS emite el mismo número de resultados parciales por consulta (*T*) y que todos los pesos emitidos por los QS se encuentran igualmente distribuidos. En circunstancias reales los QS responderán con resultados parciales de diferente cardinalidad y diferente distribución de pesos dependiendo principalmente de dos factores: la distribución temática de las páginas (proyección sobre los QS) en sus índices locales invertidos y las modas (debido a que estas determinan el contenido de las consultas y por tanto influyen en las respuestas de los QS).

Estos dos factores son más acusados si tenemos en cuenta arquitecturas IR tipo cluster, donde cada uno de los clusters alberga una o más temáticas de entre un conjunto de temáticas predefinidas.

Con estos requerimientos, en donde unos QS responderán con resultados parciales diferentes a los demás (en cardinalidad o distribución de probabilidad) una topología simétrica puede no ser la más adecuada[2], dado que tendremos que hacer el mismo esfuerzo computacional y de comunicaciones sobre resultados parciales básicamente diferentes.

Consideremos el canal de respuesta como una estructura tipo árbol (jerárquica) en donde los QS son las hojas y el broker de usuario o final es la raíz. Bajo este prisma un canal de respuesta simétrico *n*x*n* posee tres niveles con un broker final como raíz, *n* hijos y *n*x*n* nodos hoja (QS). Los canales de respuesta asimétricos estarán balanceados hacia un lado o hacia otro, dependiendo de la profundidad de la estructura o el número de nodos que sean hijos de otro superior. Dicho canal de respuesta asimétrico permitiría adecuar mejor las respuestas de los QS en situaciones de operativa real.

6.3 Conclusiones

En el caso que hemos puesto como ejemplo a lo largo de este documento, 1024 servidores emiten *T* documentos cada uno de ellos como respuesta a una consulta. Esto conlleva que el *broker* de usuario tenga que procesar 1024T documentos mediante un algoritmo de ordenación y selección (*tc*) que en el mejor de los casos es $\mathbf{O}(1024T \log_2(1024T))$. Usando un filtro BGFAST con $f = 8$ y $s = 1$, la complejidad del algoritmo de ordenación es aproximadamente $\mathbf{O}(5T \log_2(5T))$, esto es, aproximadamente menos de un 5% del tiempo

[2] Aunque insistimos, a priori parece ser la topología óptima si la distribución de los pesos en las respuestas parciales de los QS se encuentran uniformemente distribuidas.

original. Este resultado aún mejoraría sustancialmente si usamos un valor de $s > 1$, pero introduciendo una probable, aunque pequeña, pérdida de precisión.

Con estos datos podemos concluir en primer lugar que al algoritmo BGFAST funciona con una eficiencia muy elevada cuando es combinado con otros algoritmos. Por sí sólo BGFAST funciona con un alto rendimiento (mejor incluso que *HardBroker* BGFAST en términos de tiempo), pero en el contexto que impone las características del propio filtro (recordemos que por sí mismo no es un filtro determinista y, aunque hayamos obtenido la forma de cuantificar el volumen de su salida - el número de documentos efectivamente filtrados - esta permanecerá sin ordenar).

La utilidad de los filtros BGFAST no sólo se limita a los nodos y el broker de usuario, si no que su uso puede ser extendido a los mismos QS, en donde, en la etapa de recolección (donde típicamente se obtienen del orden de 10 o 100 veces T como respuesta a una consulta) y antes de ordenar los resultados para obtener los T mejores, estos pueden someter los documentos que cumplen la consulta a un filtro de las características estudiadas, reduciendo así drásticamente el número de elementos a ordenar antes seleccionar y transmitir, y por tanto obteniendo un menor valor para $t_{i,j}$.

La solución que hemos planteado ha sido diseñar un modelo distribuido y determinista de filtrado en entornos de IR usando técnicas de redes programables. Hasta donde el conocimiento de los autores alcanza no se conoce un resultado parecido acerca de la semántica en el canal de respuesta en ámbitos de IR, centrándose todo el esfuerzo en las arquitecturas de comunicaciones y sobre todo en el canal de consulta y las técnicas y tecnologías implicadas en la obtención de la respuesta.

Según nuestros experimentos el algoritmo BGFAT, combinado con otros algoritmos, es superior a 140 veces más rápido que su versión tradicional (BGFAT combinado con HardBroker, el cual está a su vez basado en el algoritmo *quicksort*). Así mismo la huella en memoria de nuestra propuesta es sensiblemente inferior (52KB respecto a 2MB).

El algoritmo BGFAT en una arquitectura de canal de respuesta centralizada (usándolo sólo en los brokers de usuario) permite una mejoría sustancial, reduciendo el juego de respuestas parciales a ordenar de $\Sigma t_{i,j}$ a $T(H_{p+f} - H_f)$, que para el caso de 1024 QS representa una reducción de más del 99% (1024M frente a menos de 5K) del juego de documentos de entrada con el consiguiente impacto el la eficiencia del algoritmo de ordenación.

Respecto a la mejora del modelo de filtros distribuido usando BGFAST respecto al modelo de brokers centralizado, las simulaciones muestran una mejoría del 64,5%.

Con estos datos podemos concluir que el algoritmo GFAT funciona con una eficiencia muy elevada cuando es combinado con otros algoritmos. Por sí sólo GFAT proporciona un alto rendimiento pero en el contexto que impone las características del propio filtro (recordemos que por sí mismo no es un filtro determinista y, aunque hayamos obtenido la forma de cuantificar el volumen de su salida - el número de documentos efectivamente filtrados - esta permanecerá sin ordenar). Así mismo hemos constatado una mejoría de la eficiencia al usar un modelo de filtros distribuidos respecto a un modelo de brokers centralizado.

BGFAST puede ser usado en contextos donde exista un flujo (ordenado parcialmente o no ordenado) de datos, con un criterio de clasificación y en donde se requiera seleccionar un subconjunto de estos. BGFAST puede ser usado para seleccionar dicho subconjunto bajo el criterio de mayor, menor o rango, dado que es posible transformar el criterio de clasificación a nuestra voluntad[3].

Estos resultados han podido realizarse gracias a la construcción de un nodo de valor añadido embebido (VAIN) según los diseños mostrados en los capítulos 2 y 4 (y cuya funcionalidad, desde el punto de vista del usuario, se explica en el capítulo 7). El carácter programable en tiempo de ejecución de esta herramienta ha permitido probar exhaustivamente los diseños detallados en este documento y dado que se trata de una herramienta de propósito general, puede ser usada para extender los estudios experimentales sobre el tema o abordar otras investigaciones donde se requiera el uso de componentes activos en la subred. Así mismo esta tecnología es la clave de la transparencia de la arquitectura del canal de respuesta filtrado, dado que mediante el uso de, por ejemplo, brokers intermedios sería necesario que los QS conocieran detalladamente el destino inmediato de sus resultados parciales y por tanto cualquier modificación de la topología implicaría la reconfiguración de los QS.

Bajo este prisma los QS emiten tráfico IP con destino a uno o más brokers[4]. Los nodos programables, trabajando en modo promiscuo, capturan dicho tráfico y lo procesan,

[3] En el contexto que nos ocupa podríamos haber querido seleccionar los de menor peso, o los de peso dentro de un rango. Incluso cualquier tipo de clasificación, dado que podemos realizar una transformación del peso $Tr(x)$, tal que dicha función proyecte el espacio de pesos en un espacio arbitrario a nuestra voluntad.

[4] Los QS conocen de antemano las direcciones IP de los brokers finales, sin embargo, dependiendo de la estrategia usada, es posible que los QS envíen sus respuestas a diferentes brokers de cada vez con el objetivo de no sobrecargarlos. De esta manera se puede balancear la carga entre los brokers finales [Cacheda, 05].

emitiendo sus resultados como si se tratase de un QS[5] virtual. Además, los nodos deben capturar el tráfico ARP emitido[6] por los QS y respondiendo con la dirección MAC de los filtros reales[7]. De esta manera es posible construir una topología de filtrado tan compleja como sea necesario sin que los QS o los brokers finales necesiten ser reconfigurados.

Finalmente, los conceptos e ideas mostrados en este trabajo han sido compartidos, discutidos y analizados por medio de publicaciones científicas (revistas [Puentes, 05] y congresos [Puentes, 03a] [Puentes, 03b] [Puentes, 03c] [Puentes, 03d] [Puentes, 04a] [Puentes, 04b] [Puentes, 04c] [Puentes, 04d]) tanto nacionales como internacionales.

6.4 Trabajo futuro

GFAT reduce, en su papel de filtro, la cardinalidad del conjunto de entrada de nT (n paquetes con T elementos cada uno de ellos) a TH_n, garantizando que en su salida se encuentran los T mejores (también *peor* o *rango*) elementos según algún criterio de clasificación. Sin embargo su eficacia (capacidad de filtrado) no es perfecta y puede ser mejorada de dos formas: por un lado permitiendo algún tipo de perdida de precisión en el filtrado, por el otro mejorando el propio proceso de filtrado del algoritmo.

BGFAST muestra una capacidad muy superior a GFAT si situamos el parámetro f en su valor óptimo, con lo que consideramos que el objetivo que nos planteamos inicialmente ha sido alcanzado. Sin embargo nuestros experimentos han arrojado unos valores en términos temporales (eficiencia) no tan buenos como su capacidad de filtrado (eficacia). La primera ha aumentado un 20% en relación a GFAT, mientras que la segunda mejoró en un 75%. Esta última característica del filtro puede mejorarse con un valor del parámetro s superior a 1, pero bajo el riesgo de perder la precisión de la respuesta.

Debido a esto consideramos interesante proseguir el estudio de la eficiencia de esta familia de filtros, en términos temporales, combinando diferentes parámetros de s y f, buscando minimizar el segundo (incluso llegando a cero) y aumentar el primero. En el caso más extremo nos proponemos comparar la eficiencia de los filtros GFAST con $s > 1$ y BGFAST

[5] Los nodos pueden enviar su tráfico haciendo uso de su propia dirección IP y MAC, no necesitan enmascararlas dado que los brokers, a priori, no usan esta información.

[6] Nos referimos, en la etapa de descubrimiento, a la consulta de conversión de dirección IP a MAC, necesaria para establecer en enlace entre dispositivos en una red Ethernet, por ejemplo.

[7] De esta forma, al cambiar la topología de la red de filtrado, no necesitamos descartar la *caché* ARP de los QS. Esta no necesita ser cambiada a no ser que modifiquemos un broker real.

con $s = 1$ y $f = 0$, teniendo el cuenta la pérdida de precisión inducida al aumentar el parámetro s del filtro.

Respecto a la aplicabilidad de esta familia de filtros, aunque los hemos estudiado en un contexto de recuperación de la información, en concreto en el canal de respuesta, pueden ser usados en cualquier problema donde un algoritmo requiera ordenar un conjunto de elementos lo suficientemente grande para convertirse en un cuello de botella y tan sólo necesite un subconjunto de los mismos (mejor, peor o rango). Situando una etapa previa de preprocesamiento es posible reducir el juego de elementos a ordenar, disminuyendo la complejidad del algoritmo de ordenación, aumentando apenas el de la suma de ambos algoritmos. Por tanto otro de nuestros objetivos para el futuro es buscar nuevas áreas donde aplicar esta familia de algoritmos, bien en situaciones como las explicadas en este párrafo, bien en otros entornos que aún no hemos identificado.

En este documento hemos considerados que la arquitectura DIR implementa los filtros BGFAST en el *broker* de usuario, al menos una etapa intermedia de n nodos o incluso en los QS, durante la etapa de recolección. Centrándonos en el canal de respuesta filtrado, una vía de investigación que dejamos abierta es el estudio de la mejor topología de red, considerando como requisito tanto la capacidad de las líneas de comunicación como la carga de consultas a lo largo del tiempo. Esta vía abre las posibilidades a una arquitectura *real* de recuperación de información, filtrada en el canal de respuesta, bajo los parámetros de caudal disponible y capacidad escalar, siempre salvaguardando el objetivo deseado de la transparencia. El objetivo de esta investigación es crear una metodología de trabajo que permita diseñar el mejor canal de respuesta para una arquitectura dada bajo las restricciones impuestas por las comunicaciones y el ratio de consultas esperado.

La ventaja de la transparencia, que obtenemos al usar nodos programables, es vital si tenemos en cuenta las consideraciones vistas anteriormente más aún si tenemos en cuenta que estas, tanto la distribución temática de las páginas sobre los QS como las modas, cambian a lo largo del tiempo, y una arquitectura IR afinada deberá poder adaptarse a tales circunstancias.

La implementación real de dicha arquitectura puede llevarse a cabo mediante el uso de, por ejemplo, tecnologías de emulación de redes locales sobre ATM (LANE), que permiten la creación de redes locales virtuales mediante la formación de segmentos de la capa de enlace, habilitando incorporar o extraer dinámicamente elementos como hosts (QS y brokers) y dispositivos de conmutación (nodos programables) a voluntad. Mediante esta topología es posible tener un conjunto estático de QS y brokers, además de una "*granja*" de nodos programables de dimensión arbitraria y cambiar dinámicamente la topología de la red

adaptándola a las circunstancias de cada situación (cambios de modas, eventos y acontecimientos, etc.). Sin embargo esta arquitectura no carece de problemas; cada nodo programable (susceptible de ser gestionado) necesita ser sincronizado antes de resituarlo dentro de la topología (para que termine de procesar las consultas pendientes y aun no completadas).

Así pues la topología de la subred del canal de respuesta es un área de investigación abierta que promueve desarrollo de arquitecturas de comunicaciones óptimas a situaciones reales e implementables en el mundo real y para entornos de IR heterogéneos. Para ello es vital el desarrollo de herramientas que al ser embebidas en el interior de la subred, como elementos activos, permitan mejores desempeños y no se conviertan, a la larga, en cuellos de botella en potencia. La arquitectura de nodo programable VAIN, mostrada en este documento, permite experimentar con fluidez con nuevos diseños de algoritmo, sin la necesidad de complejos o costosos dispositivos de propósito determinado *a priori*. El diseño del nodo VAIN puede ser mejorado bajo dos prismas: por un lado la propia arquitectura, aunque arroja unos resultados excelentes en términos de flexibilidad o independencia, puede ser mejorada permitiendo reconocer tráfico con dependencias temporales, más seguridad y robustez. Por otro lado estos nodos pueden ser enriquecidos con servicios añadidos, soporte de estándares y arquitecturas complejas, que habiliten que una red VAIN pueda ser un auténtico laboratorio dentro de un laboratorio.

7 Apéndice: uso y programación del nodo

7.1 Introducción

Las siguientes secciones describen, a modo de manual de usuario y desarrollo, el punto de vista externo del sistema. No entraremos en detalles de cómo ha sido construido, si no que explicaremos en su lugar cómo ha de usarse. Para una detallada descripción del sistema y sus componentes véase el capítulo 4, "*Implementación del nodo programable*".

La primera subsección detalla la estructura de ficheros en el cual reside el sistema: sus ejecutables y librerías así como directorios. También mostramos como iniciar los contextos de usuario y finalizarlos, el fichero de configuración y las limitaciones del nodo.

La segunda subsección muestra el funcionamiento de la consola del nodo, la herramienta mediante la cual podemos ponernos en contacto con el sistema en su totalidad y operar administrativamente sobre él. Esta sección se enlaza con la siguiente, la correspondiente a la descripción del lenguaje CL, el cual es ampliamente usado dentro del sistema para el envío de comandos en formato texto.

La siguiente subsección introduce el tema de las expresiones de red, herramienta mediante la cual describimos el filtro de tráfico de red que deseamos capturar. De nuevo esta sección se encuentra íntimamente enlazada con la que la sigue, los esquemas de red.

Para finalizar las dos últimas secciones muestran el punto de vista del desarrollador de servicios, contextos de ejecución, la estructura de un módulo y las clases utilitarias. La última sección muestra como iniciar y finalizar un servicio desde la consola del nodo.

7.2 Directorios, Ejecutables y librerías

El nodo en sí está formado por varios ejecutables y librerías, compartidas entre los diferentes subsistemas. A diferencia de la metodología clásica de los entornos de la familia UNIX, los ficheros que forman el nodo no se encuentran distribuidos en los diferentes subdirectorios estándar del sistema de ficheros, si no que residen bajo un único subdirectorio. Es posible distribuir los ejecutables y librerías a lo largo de las trayectorias por defecto modificando los ficheros de configuración, sin embargo aquí mostraremos el punto de vista compacto, tal y como viene en la distribución.

La muestra la estructura básica del sistema de ficheros bajo el directorio raíz *./vain*. Los ficheros más importantes bajo este subdirectorio son el fichero *Makefile* y el fichero de configuración por defecto *vain.xml*. El primero es usado junto con la herramienta *make* - o *gmake* - para crear las dependencias (*make dep*), limpiar los ficheros binarios (*make clean*) o construir el nodo (*make*). Todos los ficheros binarios finales creados a partir de esta herramienta se almacenan en *./vain/bin* o en *./vain/lib* dependiendo de su naturaleza.

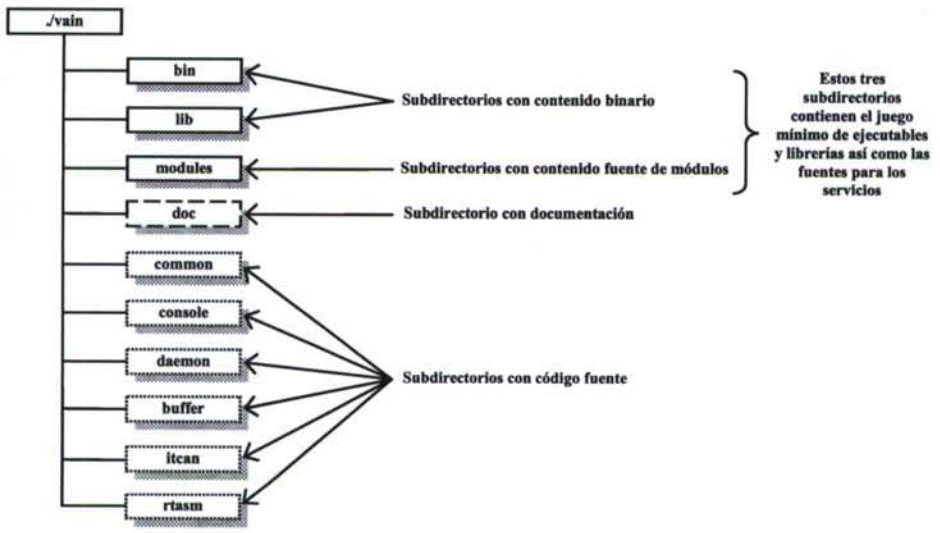

Ilustración 55: Jerarquía básica de ficheros del sistema

El fichero de configuración *./vain/vain.xml* ha de tomarse como ejemplo o como fichero de configuración por defecto, al arrancar el nodo – un contexto de usuario – habrá de indicarse que fichero de configuración usaremos. Aunque no es estrictamente necesario cada contexto se usuario debería ser arrancado con el mismo fichero de configuración.

Al ejecutar *make* se crean los ficheros binarios en *./vain/bin* y *./vain/lib*. De entre los ejecutables destacamos *./vain/bin/vaind* y *./vain/bin/vain-cl* ambos el *daemon* y la consola del nodo respectivamente.

Respecto a las librerías *./vain/lib/libvaind.so*, *./vain/lib/libitcan.so* y *./vain/lib/librtasm.so*: La primera contiene toda la funcionalidad del nodo y es usada tanto por el *daemon* como por la consola y los servicios. Las otras dos librerías forman el módulo de reconocimiento de tráfico de red (módulo ITCAN y ensamblador en tiempo de ejecución respectivamente) usados por algunos de los servicios.

Otras librerías que el sistema debe poseer son: *libpthread.so*, *libpcre.so*, *libdl.so* y *libxml2.so*; que se corresponden con los paquetes *PThread* (*threads* a nivel de usuario), PCRE (expresiones regulares al estilo *Perl*), DL (carga dinámica de librerías) y XML2 (procesamiento de fuentes XML) respectivamente. El nodo, actualmente, depende también en tiempo de ejecución del compilador GNU C/C++ dado que los módulos son compilados durante la operativa del nodo para formar los servicios usando este compilador.

La siguiente sección muestra el formato detallado del fichero de configuración.

7.3 El fichero de configuración

Como hemos indicado en la sección anterior el nodo, al arrancar cualquier contexto de usuario, necesita leer los parámetros iniciales de configuración. Estos datos se encuentran agrupados en formato XML en uno o más ficheros – normalmente uno sólo por nodo, aunque esto no excluye usar uno por usuario o cualquier otra combinación – que indicaremos al arrancar los contextos de usuario.

El siguiente código indica el formato (primer nivel) del fichero XML que el nodo espera encontrar.

```
<VAIN version="3.0">
  <!-- secciones -->
</VAIN>
```

De haber más de una etiqueta de primer nivel, el sistema lo ignora y busca una con nombre VAIN con la versión que se corresponda con la indicada en tiempo de compilación.

En el interior de la etiqueta de primer nivel podemos especificar varias secciones que detallamos seguidamente. De estas las únicas obligatorias son *users* y *drivers*.

7.3.1 Sección: *daemon*

```
<daemon>
    <shm size="8M" at="0x50000000"/>
</daemon>
```

La única entrada que por el momento reconoce el nodo es la relativa a la configuración de la memoria compartida. Los atributos *size* y *at* de la etiqueta *shm* hacen mención al tamaño deseado de la memoria compartida y a su localización dentro del espacio de direcciones del proceso. El atributo del tamaño sólo es usado cuando el nodo crea la memoria compartida y por defecto es "32M" [1]. Respecto al enlace en una dirección fija, de no existir - o ser 0 - cede al S.O. la elección de esta dirección. Por defecto es 0. El sistema usa la memoria compartida con punteros relativos, de forma que no es necesario que varios procesos compartan la misma dirección base de la memoria compartida.

7.3.2 Sección: *users*

```
<users>
    <user name="root" home="/root" />
    <user name="fran" home="/home/fran" />
</users>
```

Esta sección enumera los usuarios que están habilitados para crear contextos. El usuario debe existir y ser accesible usando *getpwnam*() así mismo es posible asignarle un subdirectorio *home* donde el usuario podrá acceder al sistemas de ficheros. De no indicar un subdirectorio *home*, el nodo toma el subdirectorio por defecto que indique el S.O. El contexto de usuario estará limitado al subárbol por debajo de su directorio *home*, no podrá acceder a otra parte del sistema de ficheros fuera de este subdirectorio. La única excepción a esta regla se produce cuando el usuario es administrador o miembro del grupo de administradores, en cuyo caso el acceso al sistema de ficheros es total.

El directorio *home* debe tener los derechos de lectura, escritura y acceso apropiados.

7.3.3 Sección: *drivers*

```
<drivers>
  <driver   name="c++"      bin="/usr/bin/g++">
    <value name="home"     text="/home/fran/vain"/>
    <value name="defines"  text="-D_REENTRANT -D_ACTIVE_LOG"/>
    <value name="includes" text="-I$(home)/common"/>
    <value name="includes" text="-I$(home)/daemon"/>
    <value name="libs"     text="$(home)/lib/libvaind.so"/>
    <value name="command"  text="$(bin) -shared -fPIC $(defines) $(includes)
                                $(sources) $(libs) -o $(target)"/>
  </driver>
</drivers>
```

[1] Se reconocen los postfijos M y K indicando respectivamente un millón y mil. Por ejemplo 8KK es equivalente a 8M el cual se calcula como 8*1024*1024.

Esta sección enumera y define el o los *drivers* de compilación del nodo. Por defecto se entrega el correspondiente al compilador GNU C++. El nodo exige que el *driver* de compilación, a partir de uno o más módulos (ficheros fuente) genere una librería de enlace dinámico con dos puntos de entrada enlazadas al estilo C. Dado que el nodo exige la creación de un objeto descendiente de la clase *context*, los servicios deberían desarrollase en C++.

La estructura de la definición de un *driver* se parece mucho a la funcionalidad de un fichero *Makefile*. Los atributos de la etiqueta *driver* - *name* y *bin* - declaran el nombre y el ejecutable asociado respectivamente al *driver*. La implementación de un *driver* es realmente una tabla se símbolos conteniendo una lista de pares (nombre, valor). El resto de las etiquetas *value*, se encargan de rellenar esta lista. De indicar un valor con un nombre que ya existe, este es añadido al valor original con un espacio en blanco entre ambos, de forma que así vamos formando un diccionario de pares (nombre, valor) no repetidos.

De entre estos valores hay tres que el sistema reconoce: *command*, *sources* y *target*. De hecho el primero es el único que exige el *driver* para operar con normalidad. La lista de fuentes y el fichero destino – la librería propiamente dicha – son rellenados en el momento de generar el servicio a partir de las fuentes o módulos. Para ello el sistema busca la entrada *command* y realiza las substituciones indicadas en su valor con el patrón $(*name*); en todo caso *name* debe existir en la tabla de símbolos. Dicha substitución generará un comando de sistema que al ser ejecutado correctamente con la función *execvp*() provocará la creación del fichero *target* (típicamente un fichero **.dll* en */tmp*) con el formato COFF de una librería de enlace dinámico.

La declaración de los servicios puede aumentar la tabla de símbolos con pares (nombre, valor).

7.3.4 Sección: *services*

```
<services>
     <base path="../modules"/>

     <!-- lista de servicios -->

</services>
```

La sección *services* enumera la lista de servicios por defecto que el nodo intentará arrancar al inicio de cada contexto de usuario. Son servicios por defecto, en cualquier momento podemos arrancar o cerrar servicios a nuestra voluntad.

La etiqueta *base* con el atributo *path* permiten indicar donde residen los módulos. Desde la aparición de esta etiqueta y hasta la finalización de la sección *services* (o la aparición de

otra etiqueta *base*) podemos limitarnos a indicar la trayectoria de los módulos de un servicio como una trayectoria relativa a esta indicada. El orden de aparición es, por tanto, significativo.

7.3.4.1 Subsección: *service*

```
<service name="deploy" driver="c++" user="root" config="">
      <module name="deploy.cpp"/>
      <value  name="libs"     text="$(home)/lib/libitcan.so"/>
      <value  name="libs"     text="$(home)/lib/librtasm.so"/>
      <value  name="libs"     text="$(home)/lib/libdsm32.so"/>
      <value  name="includes" text="-I$(home)/itcan"/>
      <value  name="includes" text="-I$(home)/buffer"/>
      <value  name="includes" text="-I$(home)/rtasm"/>
      <value  name="includes" text="-I$(home)"/>
</service>

<service name="ip/ethernet" user="root" driver="c++" config="">
      <module name="ethernet.cpp"/>
      <value  name="includes" text="-I$(home)/itcan"/>
      <value  name="includes" text="-I$(home)/buffer"/>
      <value  name="includes" text="-I$(home)/rtasm"/>
      <depend on="../../deploy"/>
</service>
```

La subsección *service*, dentro de la de más alto nivel *services*, permite declarar un servicio. Los tres atributos principales de la etiqueta; *name*, *driver* y *user*, indican el nombre del servicio, el *driver*[2] a utilizar y el usuario dueño del servicio. Al arrancar el contexto de usuario del denominado *root*, por ejemplo, sólo se cargarán los servicios del dicho usuario.

El contenido de la etiqueta *service* está compuesta por entradas *module*, *value*, y *depend*, todas ellas pueden aparecer más de una vez. La más importante y única obligatoria es *module*, dado que indica por que módulos está compuesto el servicio. En los ejemplos anteriormente expuestos los módulos se corresponden con ficheros fuente C++ situados en la trayectoria que indique la etiqueta *base*. Los nombres (trayectorias absolutas) de los módulos son añadidos a la entrada *sources* de la copia privada del *driver* declarado en la etiqueta *service*. Las etiquetas *value* poseen la misma funcionalidad que sus homónimas en la sección *driver*. Recordemos que declarando más de un valor para un nombre no eliminamos el valor anterior, si no que el nuevo valor se añade al final separado por un espacio en blanco.

El nombre final del servicio no es el indicado por el atributo *name*, si no que este es usado para formar el nombre completo del mismo. Este estará formado por la cadena "*/users/*", más el nombre del usuario, más "/", más el nombre del servicio. Por ejemplo: los servicios

[2] El *driver* debe estar definido en la sección *drivers*.

expuestos anteriormente se denominarán "*/users/root/deploy*" y "*/users/root/ip/ethernet*" respectivamente. Esta nomenclatura nos dará pié más adelante a tratar los servicios como una jerarquía. La excepción a esta regla de nombrado son las diferentes consolas que podemos abrir. Cada una de ellas se añadirá con el nombre "*/console/{pid}*" donde *{pid}* será el identificador del proceso obtenido con *getpid*().

Para finalizar mostraremos como es posible establecer dependencias por defecto de los servicios gracias a la etiqueta *depend* y su atributo *on*. El valor de su único atributo es la trayectoria absoluta o relativa al servicio del cual depende. Por ejemplo, para el servicio "*/users/root/ip/ethernet*" hemos establecido una dependencia "*../../deploy*" de trayectoria relativa. La resolución de esta trayectoria será "*/users/root/deploy*", indicando que el servicio debe esperar a que este se encuentre cargado y funcionando para poder recibir mensajes y por tanto ara entregar por completo su funcionalidad. Las dependencias son transitivas y no es posible formar abrazos mortales entre ellas. Dentro del código que forma el servicio es posible añadir dependencias y esperar a que sean satisfechas.

7.4 Arranque y finalización del nodo

Esta sección muestra como iniciar y finalizar el nodo, el cual está compuesto de uno o más contextos de usuario. Para finalizar explicaremos las limitaciones del mismo.

7.4.1 Contextos de usuario

Un contexto de usuario está formado por un proceso creado en tiempo de compilación y que se comporta a todos los efectos como un servicio. No es posible cambiarlo y es el mismo para todos los usuarios aunque sin embargo hereda las limitaciones que implemente el sistema para dicho usuario. La nomenclatura de los contextos de usuario es siempre igual "*/users/{name}*" donde *{name}* es el nombre del usuario. Cuando un contexto de usuario se inicia, inicia todos los servicios que encuentre en la sección *services* asignados a su nombre. De la misma manera, cuando un contexto de usuario finaliza su ejecución, este finaliza todos los servicios que esté administrado en ese momento, que pueden ser más o menos los que levanto en el momento de su inicialización.

Técnicamente un contexto de usuario está relacionado con un proceso, mientras que un servicio "normal" se relaciona con un *thread* o proceso a nivel usuario. Esto implica que todos los servicios de un usuario – administrado por su contexto – comparten el mismo espacio de direcciones, las mismas limitaciones y derechos, y, en el caso de caída del proceso, solamente los servicios de dicho usuario serán afectados.

Los contextos de usuario responden a mensajes como "*show services*", "*open service=<text>*", "*close service=<text>*", "*cancel service=<text>*", "*install service=<list>*" y "*uninstall service=<text>*"; además de los mensajes comunes a todos los servicios "*ping*" y "*quit*".

7.4.2 Arranque del nodo

El arranque o inicialización del nodo queda determinado por la ejecución de cualquier contexto de usuario. Podemos levantar tantos contextos de usuario como queramos, siempre y cuando no existan ya en memoria. Lo normal, aunque no necesario, es levantar en primer lugar el contexto del usuario *root*. Según el fichero de configuración del sistema este levantará los servicios "*deploy*" e "*ip/ethernet*" los cuales sí necesitan ser ejecutados como administrador o miembro de grupo de administradores. De no necesitar estos servicios, no es necesario ejecutar primero el contexto de usuario *root*. De cualquier manera el orden de inicio no es trascendente dado que los servicios no empezarán a funcionar hasta que sus dependencias sean satisfechas.

La ejecución del comando `./vaind --help` da como resultado la siguiente salida:

```
==========================================================================
VAIN daemon v3.0
(c) 2003-2005 Francisco Puentes
              fpuentes@acm.org
              fpuentes@udc.es
==========================================================================
Usage is: ./vaind [-h] [-v] [-u <username>] [(-c | -o)] -f <filename>
where: -h --help        Print help
       -v --verbose     Verbose mode
       -f --file        Configuration file (mandatory)
       -u --user        User name (current user is default)
       -c --create      Create shared memory (default)
       -o --open        Open shared memory
==========================================================================
```

La cual nos indica las diferentes opciones que tenemos para iniciar un contexto de usuario:

La opción *help* nos muestra esta salida y finaliza el ejecutable.

La opción *verbose* aumenta el nivel de mensajes que muestra el ejecutable antes de cerrar las entradas y salidas estándar y convertirse en *daemon*.

La opción *file* nos permite especificar el fichero de configuración a usar.

La opción *user* nos permite especificar el contexto de usuario a levantar.

La opción *create* y *open* son opciones disjuntas que nos permiten indicar si queremos crear o tan solo abrir la memoria compartida. Lo normal es que el primer contexto de usuario que

invoquemos cree también la memoria compartida, siguientes contextos de usuario deberán simplemente abrirla.

Cada contexto de usuario, en el momento inicial de su ejecución, crea varios ficheros útiles para posteriores operaciones; alguna de ellas, en caso de error, nos permite consultar el estado o razón del error y finalizar por completo el nodo. En concreto genera los ficheros "*vaind.{pid}.log*", "*ipcrm.{pid}.sh*", "*less.{pid}.sh*", "*kill.{pid}.sh*" y "*term.{pid}.sh*"; donde *{pid}* es el identificador del proceso del contexto de usuario. El fichero "*vaind.{pid}.log*" almacenará una información textual de las actividades que el contexto de usuario y sus servicios asociados han llevado a cabo. Con el *script* "*less.{pid}.sh*" podemos ver más cómodamente su contenido.

7.4.3 Finalización

La finalización del nodo consiste en finalizar todos sus contextos de usuario, uno por uno. Hay varias formas de hacerlo, aquí explicaremos algunas de ellas.

La forma más expeditiva es mandar una señal SIGTERM al proceso asociado al contexto de usuario, de esta forma finalizamos limpiamente el contexto. El *script* "*term.{pid}.sh*" lleva a cabo dicha funcionalidad. De todas formas es posible que debido a un error de sistema queramos finalizar abruptamente un contexto de usuario, para ello tenemos el *script* "*kill.{pid}.sh*" el cual manda la señal SIGKILL al proceso para que finalice. En este caso el contexto intenta finalizar limpiamente.

Habrá casos donde finalizando abruptamente todos los contextos de usuario el código no haya podido salir limpiamente de tal situación. En estos casos, para eliminar la memoria compartida – y su semáforo asociado – tendremos que recurrir al comando *ipcrm* para limpiar ambas entradas. El *script* "*ipcrm.{pid}.sh*" nos ayuda en este caso eliminando tanto el segmento de memoria compartida como el semáforo asociado.

7.4.4 Límites del nodo

El nodo hereda en su mayor parte los límites del S.O. sobre el que se ejecuta.

Respecto a los límites intrínsecos, achacables a su diseño, tenemos los límites siguientes: El número máximo de buzones de correo es 512 (cada buzón un servicio, contexto de usuario o consola). La unidad mínima que podemos reservar en la memoria compartida es de 128 bytes. El máximo segmento que podemos reservar es dependiente del tamaño máximo de la memoria compartida, la cual es a su vez dependiente del S.O. Cada buzón de correo puede albergar como máximo 255 mensajes sin procesar.

7.5 La consola del nodo

El método básico de gestión del nodo es la consola del sistema. Esta aplicación permite interactuar con el nodo usando comandos con un formato formal. Las siguientes subsecciones muestran cómo conectarse con el nodo, cómo enviar y recibir comandos y como finalizar la consola.

7.5.1 Conexión con el nodo

El ejecutable que contiene la consola del sistema es *./vain/bin/vain-cl*, el cual al ser ejecutado con el parámetro *--help* nos entrega la siguiente salida:

```
Usage: ./vain-cl [-h] [-v] [-n] [-w] [-a] [-e] [-s <service name>]
Where:
      -h --help              shows this help
      -v --verbose           active verbose mode
      -n --non-interactive   disable interactive mode
      -s --service           service name auto connect
      -w --waitfor           wait for message to be extracted
      -a --abort-if-fail     abort if an error happens
      -e --echo              enable echo
```

Todos los parámetros son opcionales y su significado es el siguiente:

El parámetro *--help* muestra esta salida y finaliza el ejecutable.

El parámetro *--verbose* aumenta el nivel de detalle en los mensajes de traza.

El parámetro *--non-interactive* deshabilita el modo interactivo en el cual se imprime un *prompt* esperado por la entrada del usuario y mostrando el servicio actual. Esta opción es necesaria cuando se ejecutan *scripts*.

El parámetro *--service* intenta conectarse inmediatamente a un servicio determinado dado por el valor del parámetro.

El parámetro *--waitfor* indica que queremos esperar un tiempo dado – actualmente un segundo – a que los mensajes que enviemos sean leídos y tengamos una respuesta[3].

El parámetro *--abort-if-fail* es de nuevo útil en caso de ejecutar *scripts*. Esta opción provocará que la consola finalice en caso de haber un error de cualquier tipo, incluidas las respuestas *not ack* de los servicios.

[3] En caso de no indicar esta opción la consola sigue esperando por una respuesta por un tiempo definido – un segundo – sin embargo no comprueba que el destino ha leído el mensaje enviado.

Finalmente el parámetro *--echo* provoca que se realice una copia de los comandos de entrada en la salida, útil en la ejecución de *scripts*.

La consola intenta localizar la memoria compartida buscando algún contexto de usuario activo[4]. En caso de encontrarla se conecta y registra con el nombre "*/console/{pid}*" donde *{pid}* es el identificador de proceso de la aplicación que ejecuta la consola del sistema. En caso contrario, si no es capaz de localizar la memoria compartida, la aplicación finaliza limpiamente con un mensaje de error.

7.5.2 Comandos locales y de sistema

Una vez ejecutada la consola sin, por ejemplo, un parámetro *--service*, tenemos la siguiente salida:

```
/? ■
```

Lo cual nos informa de donde estamos ("/" buzón raíz) y que espera a que introduzcamos un comando.

Los comandos pueden ser de dos tipos: locales y de sistema. Los comandos locales empiezan por el carácter "!" mientras que los comandos dirigidos al nodo empiezan por cualquier otro carácter.

Seguidamente enumeramos los comandos locales disponibles con una breve explicación:

```
!help                 muestra esta ayuda
!dir                  enumera la lista de servicios por debajo del actual
!tree                 muestra la jerarquía de servicios por debajo del
                      actual
!cd <path>            cambia el buzón actual
!connect <service>    conexión directa a un servicio (igual que !cd
                      <service>)
!disconnect           desconexión del servicio (igual que !cd /)
!sleep <time>         espera durante <time> segundos
!exit                 finaliza la consola
!quit                 finaliza la consola
!info header          muestra información sobre la cabecera de la memoria
                      compartida
!info processes       muestra información sobre los procesos registrados
```

[4] Al ejecutar un contexto de usuario, este añade – o crea – una entrada en el fichero */var/run/vain.pid* incorporando su identificador de proceso a la lista que existe en dicho fichero. Al mismo tiempo que la añade comprueba que el resto de los *pid*s indicados aún existen, de no hacerlo los elimina de la lista. De esta forma los contextos de usuario colaboran para mantener actualizada esta lista de procesos que forman el nodo. Cualquier aplicación externa que desee conectar con la memoria compartida (y por tanto con el nodo) necesita al menos un *pid* que represente un proceso ejecutándose de alguno de los contextos de ejecución.

```
!info process <index>  muestra información sobre un proceso determinado dado
                       su índice
!info statistics       muestra información estadística del sistema
!!<command>            ejecuta el comando externo <command>
```

En secciones anteriores hemos explicado como todos los servicios (del tipo que sean: contextos, servicios o consolas) tienen un nombre asociado. También hemos explicado como estos nombres tienen un formato semejante al nombre de un fichero en los sistemas de ficheros modernos: directorios, subdirectorios y nombres de ficheros separados por el carácter "/" en los S.O. de la familia *Unix*. Todos los nombres de servicios forman, así, una jerarquía de nodos. En esta subsección denominaremos nodo a cada uno de los elementos del árbol que forman los nombres de servicios, en siguientes secciones retomaremos de nuevo el significado de nodo – o simplemente sistema – como la ejecución de uno o más contextos de usuario.

Supongamos que hemos cargado todos los servicios estándar y ejecutamos en la consola el siguiente comando local:

```
/? ¡tree
```

La salida será:

```
/ <dir>
|-- console <dir>
|   |-- 2134 <srv>
|-- users <dir>
    |-- root <srv>
    |   |-- deploy <srv>
    |   |-- ip <dir>
    |   |   |-- ethernet <srv>
    |   |-- udp <srv>
    |   |-- tcp <srv>
    |   |-- ir <srv>
    |-- fran <srv>
        |-- dummy <srv>
```

Esta es la jerarquía de nodos por la que podemos movernos. Como podemos ver algunos de los nodos están etiquetados con el texto <srv> mientras otros los están con el texto <dir>. La primera etiqueta nos indica que este nodo está asociado a un servicio (posee un buzón de mensajes) mientras que la segunda no.

Si ejecutamos:

```
/? ¡cd /users/root
```

Seguido de:

```
/users/root> ping
```

Obtendremos la respuesta:

```
ack (1)
```

Estos dos comandos, el primero local y el segundo de servicio, nos hacen situar el nodo actual en el nodo llamado */users/root* para, seguidamente, enviarle a dicho servicio el comando *ping*. La respuesta de dicho comando es un *ack* con un valor 1 (un *not ack* tendría el valor 0).

Hagamos un inciso sobre el *prompt* de la consola. En los dos comandos anteriores podemos ver como el *prompt* ha cambiado de */?* a */users/root>*. En efecto, el comando *cd* nos sitúa en un nuevo nodo y el *prompt*, además de mostrar el nodo actual, indica si este nodo tiene o no un buzón asignado con los caracteres > y ? respectivamente.

El nodo raíz nunca tiene un buzón asignado.

Cualquier otro comando que no empiece por el carácter ¡ es tomado como un comando de sistema y enviado al servicio al que nos encontremos conectados. Los comandos de sistema han de ajustarse a un formato formal que detallaremos en la siguiente sección.

La consola, ante la llegada de un mensaje, quizá como réplica de un comando enviado, intentará mostrarlo lo más adecuadamente posible. La mayoría de los mensajes que envíe y reciba la aplicación serán de tipo texto o asentimientos, sin embargo es posible provocar la llegada de mensajes con formato binario.

7.6 El lenguaje CL

El lenguaje CL (*command line*) es un lenguaje formal orientado a describir estructuras arbóreas de datos. Es usado intensamente en el interior del nodo para enviar y recibir comandos tipo texto y desde la consola como un medio de interface hombre-máquina.

El operador del nodo debe conocer este lenguaje, bien por que usa la consola o bien por que desarrolla servicios. Las siguientes subsecciones detallan este lenguaje.

7.6.1 Analizador léxico

Los elementos léxicos que reconoce el analizador son:

Identificadores: deben empezar por un carácter alfabético (*isalpha*) o el carácter "_", seguidos de un número arbitrario de dígitos (*isdigit*) caracteres alfabéticos (*isalpha*) o el carácter "_".

Valores y constantes: números enteros con signo o sin él, en decimal octal, hexadecimal o binario. Números en punto flotante simples (no en notación científica). Direcciones IPv4. Cadenas de caracteres y caracteres simples, además de las constates *null*, *true* y *false*. Los

números octales y hexadecimales siguen la notación de los lenguajes C y C++, mientras que los números binarios son una extensión[5]. Sólo los números decimales y en punto flotante pueden tener signo.

Operadores: Los operadores inicio de lista ({), fin de lista (}), asignación (=) y coma (,).

7.6.2 Gramática

Seguidamente mostramos la gramática formal del lenguaje:

El objetivo a construir es un elemento *items*:

```
goal ::= items
```

El cual está formado por una lista de *item*:

```
items ::= items item
      |   item
```

Cada uno de ellos es o un identificador o un identificador seguido de una asignación y un valor:

```
item ::= IDENTIFIER = value
     |   IDENTIFIER
```

Una constante puede ser uno de los siguientes elementos:

```
constant ::= NULL
         |   TRUE
         |   FALSE
```

Y un valor de entre alguno de los siguientes:

```
value ::= constant
      |   INTEGER
      |   REAL
      |   TEXT
      |   CHARACTER
      |   ADDRESS
      |   structured
```

De entre los valores el más destacable es el denominado *structured*, el cual abre la posibilidad de que un valor pueda ser compuesto habilitando la estructura en árbol. Observar que el lenguaje determina una lista de listas (árbol n-ario):

```
structured ::= { values  }
           |   { values, }
```

[5] Por ejemplo, el número binario 0b1010 representa el decimal 10.

```
        |   {         }
```

Las listas pueden estar vacías o finalizar, incluso, con el carácter coma, en cuyo caso este carácter carece de significado, pero es útil para el desarrollador ya que impide que este tenga que llevar la cuenta de los operadores coma. Las listas pueden estar formadas por un valor, un identificador o un identificador con un valor asignado:

```
values ::= value
       |   IDENTIFIER = value
       |   IDENTIFIER
       |   values , value
       |   values , IDENTIFIER = value
       |   values , IDENTIFIER
```

7.6.3 Ejemplos

Como ejemplo pondremos un comando dirigido a un contexto de usuario con el objetivo de instalar un nuevo servicio:

```
install service={
   name="dummy/02",
   driver="c++",
   config="",
   modules= { "dummy.cpp"  },
   values=
     {
      libs     ="$(home)/lib/libitcan.so",
      libs     ="$(home)/lib/librtasm.so",
      includes="-I$(home)/itcan",
      includes="-I$(home)/rtasm",
      includes="-I$(home)"
     }
  }
```

En forma de árbol:

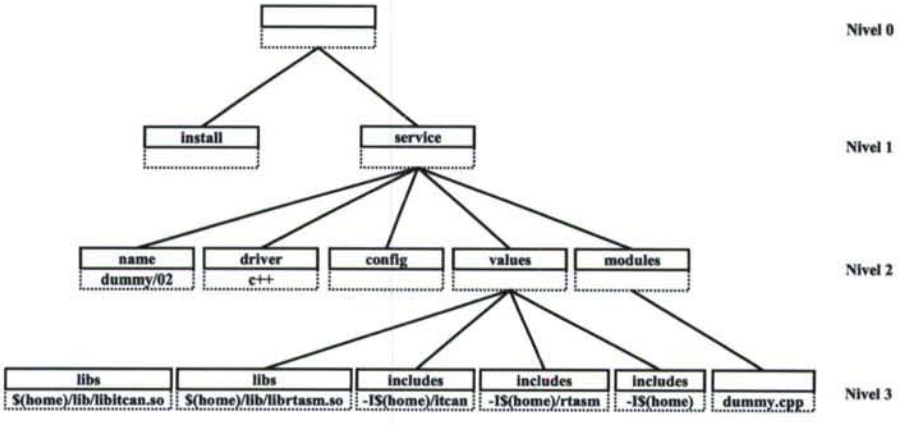

Ilustración 56: Estructura en árbol de un comando con estructura CL

La ilustración mostrada más atrás muestra la estructura en árbol del comando anterior. En él podemos observar como cada nodo del árbol está formado por el par (nombre, valor)

donde ninguno de los dos componentes del par es obligatorio. El nivel 0 es intrínseco e inaccesible, forma la raíz del árbol; siempre existe pero no podemos tener acceso a él. El nivel 1 lo forman los nodos *install* y *service*, sin valores. Este segundo contiene cinco nodos de los cuales *values* y *modules* tienen a su vez nodos hijos.

El lenguaje CL tiene como objetivo representar datos estructurados y para ello es importante que los veamos como árboles n-arios, dado que las herramientas que el sistema proporciona (*class* CLI) así los manejan.

7.6.4 Sub lenguaje CL-match

El sistema proporciona una herramienta para iterar por la estructura que forma un texto en formato CL. Como ya hemos dicho esta herramienta es la clase CLI (*command line iterator*), la cual nos entrega la posibilidad de compilar un texto en dicho lenguaje, e iterar por sus miembros.

Sin embargo la iteración puede ser bastante compleja y sobre todo no genera código fácil de interpretar. Era importante en aras de la flexibilidad y la transparencia construir una sub herramienta que ayudase al desarrollador a hacer código muy fácil de leer, simple y al menos igual de eficiente que la clase CLI. Esta herramienta es un analizador que compara una estructura (*class* CLI) con un patrón textual escrito en un subconjunto del lenguaje anfitrión CL. Su nombre CL-match.

La gramática del lenguaje CL-match es un subconjunto del lenguaje CL; simplemente eliminamos de este último los valores (elemento *value* en la gramática CL) y lo sustituimos por el elemento *type*, el cual indica el tipo del valor que el nombre del nodo del árbol en cuestión debe poseer. La gramática queda así:

```
goal ::= items
     |   items …

items ::= items item
      |   item

item ::= IDENTIFIER = < type >
     |   IDENTIFIER

type ::= list
     |   items
     |   array
     |   address
     |   char
     |   character
     |   text
     |   string
     |   real
     |   float
     |   double
     |   int
     |   integer
```

La comparación se hace por niveles; uno cada vez. Pongamos un ejemplo:

```
install service=<list>
```

Es un patrón en lenguaje CL-match que "encaja" con el comando que estamos usando de ejemplo. En él estamos indicando que el nivel 1 debe estar compuesto por dos nodos, *install* y *service*, el primero de ellos sin valor y el segundo con un valor de tipo *list*, lista de elementos. Tenemos varios tipos definidos en tiempo de diseño, algunos de ellos sinónimos:

list, *items*, *array*: el tipo ha de ser estructurado.

address: el tipo ha de ser una dirección IPv4.

char, *character*: el tipo ha de ser un carácter.

text, *string*: el tipo ha de ser un valor cadena de caracteres.

real, *float*, *double*: el tipo ha de ser un valor en punto flotante.

int, *integer*: El valor debe ser de tipo numérico con o sin signo.

El lenguaje CL-match posee una extensión que no soporta su lenguaje anfitrión. Se trata de poder indicar en el patrón que no sólo aceptamos, por ejemplo, dos elementos, si no dos o más. Así el texto:

```
install service=<list> ...
```

Reconocerá comandos CL que empiecen con *install* sin valor, le siga *service* con un valor tipo *list* y acepte que haya, opcionalmente, más valores a continuación en el mismo nivel. De no añadir los puntos suspensivos al final del patrón, este sólo reconocería los comandos con dos únicos nombres en este nivel.

7.7 Expresiones de red

Las expresiones de red son expresiones aritmético-lógicas con una sintaxis semejante a la de los lenguajes C o C++. En el nodo las usaremos para básicamente tres fines en dos contextos. El los esquemas de red usamos las expresiones para los valores *mustbe* y *length* de campos, opciones y protocolos. Su ejecución sobre un paquete devuelve un número entero sin signo que puede ser interpretado como un *boolean* para las expresiones *mustbe* y como un tamaño para las expresiones *length*. Estas expresiones también son usadas para demultiplexar el tráfico de datos; en este contexto cada expresión de red determina un canal,

el cual es una vía de comunicación bi-direccional de paquetes entre el nodo y el servicio que crea el canal.

Sea cual sea el contexto en donde estas expresiones son usadas su sintaxis es la misma. Las siguientes secciones muestran la gramática de las expresiones de red, su semántica dependiendo del contexto en donde se usen y finalmente se explican los conceptos de canal y coherencia de canal.

7.7.1 Analizador léxico

El analizador léxico de las expresiones de red reconoce los siguientes elementos:

Identificadores: deben empezar por un carácter alfabético (*isalpha*), seguidos de un número arbitrario de dígitos (*isdigit*) o caracteres alfabéticos (*isalpha*).

Valores y constantes: números enteros con signo o sin él, en decimal octal o hexadecimal. Direcciones IPv4. Caracteres simples, además de las constates *true* y *false*. Los números octales y hexadecimales siguen la notación de los lenguajes C y C++.

Operadores:

```
.       punto
,       coma
(       apertura de paréntesis
)       cierre de paréntesis
[       apertura de corchetes
]       cierre de corchetes
:       dos puntos
?       interrogante
%       operador aritmético módulo
/       operador aritmético división
*       operador aritmético multiplicación
+       operador aritmético suma
-       operador aritmético resta o negación
>>      operador aritmético RIGHT SHIFT
<<      operador aritmético LEFT SHIFT
&       operador binario AND
|       operador binario OR
~       operador binario NOT
^       operador binario XOR
==      operador lógico EQUAL
¡       operador lógico NOT
&&      operador lógico AND
||      operador lógico OR
<       operador relacional GREATER THAN
>       operador relacional LESS THAT
¡=      operador relacional NOT EQUAL
>=      operador relacional GREATER OR EQUAL THAN
<=      operador relacional LESS OR EQUAL THAN
```

7.7.2 Gramática

El siguiente código muestra un resumen comentado de la gramática de las expresiones de red:

El objetivo (*goal*) es formar una expresión bien formada según la gramática.

```
goal ::= expression

identifier ::= IDENTIFIER

literal ::= TRUE
        |   FALSE
        |   CHARACTER
        |   INTEGER
```

Podemos formar identificadores más largos uniéndolos por medio del carácter '.' (punto).

```
path ::= identifier
     |   path . identifier
```

Los elementos primarios no terminales son: literales, trayectorias (lista de identificadores unidos por puntos), trayectorias con rangos (desplazamiento, tamaño), expresiones entre paréntesis e invocación de funciones.

```
primary ::= literal
        |   path
        |   path [ expression : expression ]
        |   ( expression )
        |   functionInvocation
```

La gramática acepta el uso de funciones con un número indeterminado de parámetros. En número y formato de estas funciones es limitado y controlado desde la aplicación que use la librería en tiempo de compilación.

```
functionInvocation ::= path arguments

arguments ::= ( argumentList )
          |   (              )

argumentList ::= expression
             |   argumentList , expression
```

Las líneas siguientes muestran la gramática de las expresiones:

```
postfixExpression ::= primary

trapOverflowCornerCase ::= unaryExpressionNotPlusMinus
                       |   + unaryExpression

unaryExpression ::= trapOverflowCornerCase
                |   - trapOverflowCornerCase

unaryExpressionNotPlusMinus ::= postfixExpression
                            |   ~ unaryExpression
                            |   ! unaryExpression

multiplicativeExpression ::= unaryExpression
                         |   multiplicativeExpression * unaryExpression
```

```
                        |   multiplicativeExpression / unaryExpression
                        |   multiplicativeExpression % unaryExpression

additiveExpression ::= multiplicativeExpression
                    |   additiveExpression + multiplicativeExpression
                    |   additiveExpression - multiplicativeExpression

shiftExpression ::= additiveExpression
                 |   shiftExpression << additiveExpression
                 |   shiftExpression >> additiveExpression

relationalExpression ::= shiftExpression
                      |   relationalExpression <  shiftExpression
                      |   relationalExpression >  shiftExpression
                      |   relationalExpression <= shiftExpression
                      |   relationalExpression >= shiftExpression

equalityExpression ::= relationalExpression
                    |   equalityExpression == relationalExpression
                    |   equalityExpression != relationalExpression

andExpression ::= equalityExpression
               |   andExpression & equalityExpression

exclusiveOrExpression ::= andExpression
                       |   exclusiveOrExpression ^ andExpression

inclusiveOrExpression ::= exclusiveOrExpression
                       |   inclusiveOrExpression | exclusiveOrExpression

conditionalAndExpression ::= inclusiveOrExpression
                          |   conditionalAndExpression &&
inclusiveOrExpression

conditionalOrExpression ::= conditionalAndExpression
                         |   conditionalOrExpression ||
conditionalAndExpression

conditionalExpression ::= conditionalOrExpression
                       |   conditionalOrExpression ? expression :
conditionalExpression
```

Finalmente declaramos una expresión como una expresión condicional.

```
expression ::= conditionalExpression
```

7.7.3 Ejemplos

El objetivo de la expresiones de red es evaluarse sobre un esquema de red (ver δ7.8). El valor que la expresión devuelva es usada dependiendo del contexto. De ser tomado como un *boolean* 0 es *false* y diferente de 0 es *true*. Vamos algunos ejemplos:

```
ipv4.version==4
```

Esta expresión se hace *true* si el campo *ipv4.version* (definido en el esquema de red) posee el valor 4.

```
8*((ipv4.hlen*4)-(5*4))
```

Esta expresión se evaluará como un número entero sin signo y es usada para calcular la longitud del campo *options* en la cabecera *ipv4*. Los tamaños de los campos siempre se miden en bits.

```
ipv4.target==127.0.0.1 && ipv4.udp.port.target==666
```

Ahora estamos ante una expresión que se evalúa como *boolean* y forma un filtro UDP. Esta expresión es *true* si el paquete sobre el que se aplica posee la dirección destino 127.0.0.1 y el puerto destino UDP 666.

7.8 Esquemas de red

Los esquemas de red son estructuras jerárquicas que determinan las relaciones entre protocolos y campos.

Aunque el sistema permite introducir los esquemas de red de varias maneras y en varios formatos (API, CL y XML), realmente es en CL el lenguaje para el que fue hecho. Veamos un ejemplo completo de esquema de red:

```
install scheme=
        {
         protocol=
           {
            name="ipv4",
            field={ name="version", length=4,mustbe="ipv4.version==4"},
            field={ name="hlen",    length="4"  },
            field={ name="TOS",     length="8"  },
            field={ name="tlen",    length="16" },
            field={ name="ident",   length="16" },
            field={ name="flags",   length="3"  },
            field={ name="offset",  length="13" },
            field={ name="TTL",     length="8"  },
            field={ name="protocol",length="8"  },
            field={ name="checksum",length="16" },
            field={ name="origin",  length="32" },
            field={ name="target",  length="32" },
            field={ name="options", length="8*((ipv4.hlen*4)-(5*4))"},
            payload
           }
        }
```

Este código no es un esquema de red. Es, realmente, un comando de sistema que el servicio "*/users/root/ip/ethernet*" envía a "*/users/root/deploy*" tan pronto como inicia su actividad, instándole a instalar el esquema de red que contiene el valor de *scheme*. Como podemos ver el esquema contiene un único protocolo, de nombre "*ipv4*" el cual a su vez contiene 13 campos y un *payload*. Este último marca el final del protocolo (el reconocimiento finaliza correctamente al encontrar este elemento) y de no incluirse en el esquema se añade al final de cada protocolo instalado.

Tomemos como ejemplo el campo *version*. Este contiene su nombre y dos atributos, *length* y *mustbe*. Tal y como sus nombres indican estos atributos determinan la longitud y la

condición para el reconocimiento. Observar que ambos valores son expresiones de red. En caso de evaluarse como *false* la expresión *mustbe* de *version*, el reconocimiento del protocolo *ipv4* finaliza incorrectamente y se continúa con el protocolo siguiente que se encuentre al mismo nivel que *ipv4*. De no haber más protocolos en su mismo nivel el reconocimiento finaliza con una respuesta negativa.

Veamos otro ejemplo:

```
install scheme=
        {
         protocol=
           {
            name="udp",
            mustbe="ipv4.protocol==17",
            field=
              {
               name="port",
               field={ name="origin",length="16" },
               field={ name="target",length="16" }
              },
            field={ name="length", length="16" },
            field={ name="checksum", length="16" },
            payload
           }
        }
        anchor="ipv4"
```

En este caso se trata del comando de sistema que el servicio "*/users/root/ip/udp*" envía a "*/users/root/deploy*" al iniciar su actividad.

De nuevo el esquema propiamente dicho se encuentra como valor del identificador *scheme*. Sin embargo ahora tenemos varios elementos nuevos. Conjuntamente a *install* y *scheme*, en el primer nivel, tenemos el identificador *anchor* con el valor "*ipv4*". Esto indica que queremos instalar el esquema dentro del protocolo *ipv4* de esta forma establecemos las jerarquías de protocolos.

Dentro del esquema UDP propiamente dicho, tenemos que el protocolo, además de un nombre y campos, podemos añadirle una expresión *mustbe*. En este caso para que se inicie el reconocimiento de este protocolo es necesario que se cumpla esta condición. De no hacerlo el reconocimiento de la cabecera UDP falla y pasaría a reconocerse el siguiente protocolo de su mismo nivel (por ejemplo TCP o ICMP, de haber sido instalados).

Así pues los esquemas de red son la representación jerárquica de los protocolos (y campos) que el nodo es capaz de reconocer. Su flexibilidad radica en que no es necesario parar el nodo para compilar el código necesario con el fin de reconocer un nuevo protocolo, si no que instalando un nuevo servicio (o modificando uno existente) en tiempo de ejecución podemos hacer que el sistema reconozca tráfico con el nuevo esquema.

De hecho el nodo en sí no reconoce ningún protocolo, son los servicios los que instalan los esquemas: el servicio "*/users/root/ip/ethernet*" instala los protocolos IPv4 e ICMP. El servicio "*/users/root/ip/udp*" instala el protocolo UDP y por supuesto TCP es instalado por "*/users/root/ip/tcp*".

En cualquier momento enviando un comando "*show scheme*" al servicio "*/users/root/deploy*", este responde con el comando "*report scheme=<list>*" mostrando todo el esquema de red instalado conjuntamente con los canales abiertos (estos se instalan como elementos debajo de los campos *payload* de los protocolos). Así mismo el comando "*does exist=<text>*" dirigido al mismo servicio nos informa si el protocolo *<text>* está instalado, por ejemplo:

```
does exist="ipv4.tcp"
```

devuelve *ack* si el protocolo *tcp*, dentro de *ipv4* está instalado (y por tanto ambos existen en el esquema).

7.9 Canales y coherencia

Una vez instalados los esquemas de red adecuados, desde cualquier servicio es posible abrir y cerrar canales. Los canales son el equivalente a los *sockets* en programación de redes. En este caso al abrir un canal se entrega una expresión de red; esta se instala sobre el esquema de red actual y a la llegada de cada paquete se evalúa el esquema (y sus canales/expresiones). Si una expresión asignada a un canal se hace *true* sobre un paquete determinado, entonces el paquete es enviado al servicio que creó el canal. Esta es la forma de instalar filtros de tráfico desde los servicios.

Retomemos un ejemplo anterior de filtro y veamos como se instala sobre el esquema correspondiente:

```
ipv4.target==127.0.0.1 && ipv4.udp.port.target==666
```

Este filtro captura el tráfico UDP sobre IPv4 con destino a la dirección 127.0.0.1 y puerto 666. Al abrir el canal con este filtro, el servicio "*/users/root/deploy*" busca un nodo *payload* donde la expresión sea coherente. La coherencia de los filtros (o expresiones) es un tema muy importante, dado que indica donde va a actuar un filtro y por consiguiente si funciona correctamente o no. Podemos definir que una expresión es coherente con un nodo determinado de un esquema de red sí y sólo sí cada uno de los campos a los que hace referencia la expresión es reconocido antes de ser evaluada.

En el ejemplo que nos ocupa la expresión hace referencia a los campos *ipv4.target* e *ipv4.udp.port.target*, por tanto, y según la definición, el servicio responsable buscará en el esquema aquellos nodos *payload* que hagan que la expresión sea coherente. Será allí únicamente donde se instalarán las copias de la expresión que hace de filtro y determinan un canal. En el caso de no haber ningún protocolo dentro de UDP, la expresión se instalará sólo bajo el nodo *ipv4.udp.payload*.

Este otro ejemplo más general clarificará un poco los conceptos:

```
ipv4.target==192.168.0.1
```

Este es un ejemplo más genérico. Supongamos que hemos instalado además de este último filtro, el anterior coherente con UDP. Al añadir esta expresión el servicio responsable buscará todos los campos *payload* que hagan coherente este nuevo filtro. En este caso los campos son *ipv4.payload* e *ipv4.udp.payload*, ya que dicha expresión puede ser reconocida en potencia tanto al reconocer el protocolo IPv4 como el UDP (estando este último definido dentro del primero).

7.10 Contextos de ejecución

Todo servicio se ejecuta en el interior de un contexto de ejecución. Este es una entidad abstracta que entrega gran parte de la funcionalidad que un servicio puede necesitar a lo largo de su vida.

En nuestra implementación un contexto de ejecución es una clase, y todo servicio debe crear otra clase heredando de este para formar el objeto que identificará unívocamente el servicio.

En las siguientes subsecciones mostraremos el aspecto básico de un servicio y los componentes principales y clases utilitarias que el desarrollador tiene a su disposición.

7.10.1 Estructura básica de un servicio

La distribución del nodo contiene un módulo que implementa un servicio "plantilla" cuya funcionalidad es mínima. Su objetivo es mostrar la forma de crear un servicio y de servir de plantilla para otros servicios. El nombre del módulo es "*dummy.cpp*", si bien el servicio que implementa puede tener cualquier nombre, dado que este le es asignado fuera del módulo, de forma que podamos tener múltiples módulos idénticos (instancias) con diferentes nombres.

El código siguiente ilustra el contenido mínimo de un módulo implementando un servicio:

```
#include <stdlib.h>
#include <unistd.h>

#include "context.h"
#include "message.h"
#include "cl.h"

class dummy:public Context
{
 private:

 public:

   dummy(Service*srv):Context(srv)
   {
   }

  ~dummy()
   {
   }

   virtual
   action_t handler(Message&msg,sid_t from,mid_t mid,mid_t rid,cli cl)
   {
    action_t action;
    if((action=Context::handler(msg,from,mid,rid,cl))==a_none)
      {
       return a_none;
      }
    else
    return action;
   }


   virtual
   void run()
   {
    sys.log("'%s' ha abierto y está ejecutándose",sys.name());

    if(sys.mbx.deps.waitfor())
      {
       while(dispatch()==false);
      }
   }
};

extern "C"
Context*factory(Service*srv)
{
 return new dummy(srv);
}

extern "C"
void destroy(Context*ctx)
{
 delete ctx;
}
```

Este código, una vez compilado, cargado e instanciado, solamente responde a los comandos "*ping*" y "*quit*"; ambos ejecutados dentro de la clase padre *Context*.

Empezamos la explicación de este código por el final:

Cuando el nodo, a través del contexto de usuario, levanta el servicio, busca en la librería de carga dinámica dos entradas denominadas respectivamente *factory* y *destroy*. Debido a que el lenguaje C++ almacena internamente los nombres de los símbolos codificados con sus

parámetros y tipo de retorno necesitamos indicar mediante la instrucción *extern "C"* que deseamos que los símbolos sean almacenados al estilo C, esto es, simplemente su nombre, de forma que podamos recuperarlos desde el exterior en el momento de la carga de la librería[6].

La función *factory* tiene como parámetro un objeto de tipo *Service*. Este objeto representa el lado del nodo del servicio y sólo debe ser usado para crear la clase *dummy*, declarada en la parte superior del código. La semántica de esta función se limita a crear un objeto de dicha clase y devolverlo. Aquí no necesitamos hacer más.

Por el contrario, la función *destroy* hace la operación inversa: debe destruir el objeto que le pasamos como parámetro; que será el mismo que habremos creado con su función gemela. Estas dos funciones serán siempre iguales a las del ejemplo, con la salvedad de la clase del objeto a instanciar.

Una vez creado el objeto instancia del servicio el nodo llama al método virtual *run* y se queda esperando a que este vuelva en el interior de un *thread* de usuario. Por tanto la ejecución de un servicio reside por entero en el método *run*. El servicio inicia su funcionalidad cuando este método es invocado y finaliza cuando este método devuelve el control al nodo.

En el caso que nos ocupa el método *run* introduce una nueva entrada en su fichero de traza, espera a que sus dependencias sean satisfechas y ejecuta el bucle principal de procesamiento de mensajes. La función *sys.log* posee una sintaxis semejante a la conocida *printf*. La función *sys.mbx.deps.waitFor* espera a que las dependencias hayan sido satisfechas, en cuyo caso devuelve *true*. Si devolviese *false* significa que estas no han sido satisfechas y ha recibido la orden de finalizar.

El bucle principal está formado por el código *while(dispatch()==false);* el cual se ejecuta mientras el método de la clase base *Context::dispatch()* devuelve *false*. Cuando devuelve *true* el servicio debe finalizar.

[6] La librería *GLibc* entrega la función *demangle*, la cual a partir del nombre de un símbolo codificado nos devuelve su firma (nombre, tipo y posiblemente sus parámetros si se tratase de una función o método). Desgraciadamente la función *mangle* (la inversa) no existe en dicha librería dado que la codificación de nombres de símbolos es una característica altamente dependiente de la herramienta usada e incluso de la versión. Habría sido posible hacer la función *mangle* para el ABI (*application binary interface*) de G++, sin embargo esta sería dependiente de la herramienta y así los *drivers* de compilación sólo podrían ser usados con G++.

Ya hemos descrito el ciclo de vida principal de un servicio, sólo nos queda mostrar cómo recibe este los mensajes. El método *Context::dispatch()* es el encargado de esperar por un mensaje y a la llegada de este invocar el método virtual *Context::handler()* el cual habremos solapado con nuestra implementación. Así cada vez que haya un mensaje dirigido a nuestro servicio nuestra implementación del método *handler()* es invocado. Analicemos este método:

```
virtual action_t handler(Message&msg,sid_t from,mid_t mid,mid_t rid,cli cl);
```

Es un método virtual, por lo cual sabemos que la clase base implementa un método con la misma firma. Devuelve un valor de la enumeración *action_t* el cual indica a *Context::dispatch()* que queremos hacer (o que hemos hecho), en el caso que nos ocupa devolvemos lo que la llamada al método de la clase base ha devuelto o en su defecto *a_none*, indicando así que no hemos procesado el mensaje. Los parámetros del método son: *Message&msg* es un objeto pasado por referencia conteniendo el mensaje. *sid_t from* es el identificador del servicio origen del mensaje (entero con signo). *mid_t mid* es el identificador del mensaje que recibimos y *mid_t rid* el identificador del mensaje del cual este es respuesta, cero si no es respuesta de ningún mensaje (enteros sin signo). Finalmente *cli cl* es el objeto que nos permitirá iterar a través del comando que el mensaje en potencia contiene. En caso de no contener ningún comando este objeto contiene solamente una raíz sin descendientes.

Para finalizar el recorrido por la codificación de un servicio básico, solo añadir que debemos implementar el constructor y destructor semejantes a los mostrados. Normalmente será aquí donde iniciemos y finalicemos las propiedades privadas del objeto que hereda de la clase Context.

7.10.2 La clase Message

La clase *Message* es el contenedor que nos permite manejar los mensajes que recibamos o enviemos, pero antes de describir los elementos de la clase debemos hacer una introducción sobre los subcomponentes de los mensajes propiamente dichos.

Un mensaje está compuesto básicamente por tres elementos: el tipo del mensaje (*kind*), el mensaje propiamente dicho (*message*) y un valor pasado con el mensaje a disposición del usuario (*userValue*).

El tipo de mensaje es un valor entero sin signo de 16 bits que es interpretado como una lista de valores booleanos. A su vez este valor es dividido en dos partes, la parte del sistema (8

bits) y la parte de usuario (otros 8 bits). El sistema sólo tiene en cuenta los bits del 0 a 7 al tratar los mensajes.

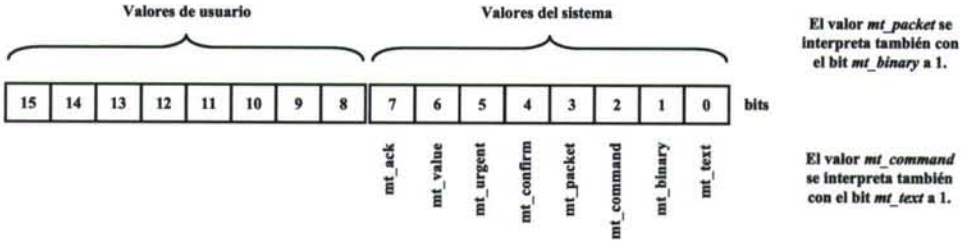

Figura 1: Interpretación de los bits de usuario y de sistema en el campo "tipo de mensaje"

Los ocho primeros bits, los valores del sistema, son banderas booleanas que el sistema interpreta para mandar o manipular el mensaje. *mt_text* indica que el campo *message* es una cadena de caracteres acabada en cero binario (*char**). Por el contrario *mt_binary* especifica que el mensaje es un puntero a una lista de bytes de cualquier tipo (*void**); no es posible conocer por estos medios la longitud de esta información[7]. La bandera *mt_command* implica siempre también *mt_text*; su significado indica que el mensaje es un comando de tipo texto. Similarmente *mt_packet* implica del mismo modo *mt_binary*, dado que señala que el envío es un paquete, por tanto binario, con un formato (la cabecera) específico e independiente de dispositivo. *mt_confirm* simplemente señala al receptor que el emisor espera una confirmación. *mt_urgent* es una bandera que es tomada en cuenta por el gestor de mensajes y después borrada; así pues el receptor nunca recibe esta bandera puesta a *true*. Su significado ayuda al sistema a decidir si el mensaje es lo suficientemente importante como para que el mensaje que lo contiene sea encolado el primero de la lista, y no el último como sería lo normal[8]. El valor *mt_value* indica que el campo *userValue* tiene significado, que contiene un valor útil. Finalmente la bandera *mt_ack*, sirve como indicador de que el mensaje es un asentimiento, típicamente asociado al campo *userValue*. En general el

[7] La librería de gestión de memoria compartida permite, sin embargo, conocer la longitud real del espacio reservado en múltiplos de 128 bytes. Así pues es posible conocer la longitud del segmento, o al menos una cota superior. Al igual que ocurre en el caso de que el contenido del mensaje es una cadena de caracteres acabada en cero, la longitud exacta del mensaje debe poder deducirse de su contenido.

[8] Los mensajes se encolan en una lista – buzón – a la espera de que el receptor esté dispuesto a leerlos. Así pues tenemos dos formas básicas de encolar los mensajes, al principio o al final de la lista. Situando los mensajes al principio de la lista aseguramos que el mensaje es leído antes.

sistema envía un *mt_ack* con *userValue* puesto a cero para indicar la falta de asentimiento, mientras que un valor diferente a cero indica todo correcto.

Los bits de 8 a 15 (de usuario) están a disposición de los servicios para ser empleados indistintamente; el sistema no los comprueba.

La clase *Message* nos permite tratar estas propiedades de una manera coherente. El siguiente esquema UML muestra las operaciones públicas de las que disponemos al manejar un objeto instancia de la clase *Message*:

Esquema UML 1: Clase *Message*

En general los constructores de la clase necesitan al menos un puntero al gestor de memoria compartida que usarán los objetos instanciados para manipular el contenido del mensaje. El constructor copia no necesita esta referencia, dado que la obtiene del objeto que tiene como parámetro.

El método *regcmp* posee como parámetro una cadena de caracteres conteniendo una expresión regular y un valor booleano que indica si el *pattern matching* ha de ser insensible a mayúsculas/minúsculas. Devuelve *true* si el contenido del mensaje, de ser del tipo *mt_text*, es reconocido por la expresión.

El método *test* hace una comparación *exacta* del tipo del mensaje con su parámetro. No es aconsejable usar este método, dado que varios mensajes con la misma semántica podrían tener tipos diferentes (por ejemplo si *mt_confirm* está presente o no). Será mucho más útil usar los operadores booleanos y relacionales sobrecargados que veremos más adelante.

Los métodos *copy*, *cmp* y *casecmp* operan sobre el contenido del mensaje de forma semejante a las funciones *strcpy*, *strcmp* y *strcasecmp* de la librería estándar de C.

Los métodos *kind*, *text*, *pointer* y *userData*, sirven para obtener y establecer los campos correspondientes del mensaje. *text* y *pointer* operan sobre el contenido del mensaje dando por supuesto que se trata de mensajes *mt_text* y *mt_binary* respectivamente.

Los métodos *isText* y similares nos permiten averiguar bit a bit si el correspondiente valor del campo *kind* está o no activado. La excepción es *isNull* que como su nombre indica nos informa de la nulidad del contenido del mensaje.

Los métodos *reset* y *format* nos habilitan para manipular el contenido del mensaje, bien borrándolo o bien introduciendo un nuevo valor. *reset* inicializa el mensaje, eliminando el contenido anterior si su parámetro es *true*. *format* opera de manera idéntica a la conocida *printf*; si el mensaje contiene algo lo elimina y, en cualquier caso, hace que el contenido del mensaje pase a ser el texto formateado y parametrizado indicado por sus parámetros.

Sólo nos queda explicar los operadores sobrecargados, los cuales trabajan sobre los campos del mensaje. Los operadores de asignación relacionan un *r-value* con un *l-value*, siendo este último siempre un objeto tipo *Message* y el primero un valor del tipo indicado por el operador. Su semántica es asignar al campo de *l-value* que corresponda el valor de *r-value*.

El operador *operador == (char*)* es idéntico al método *casecmp*. El resto de los operadores trabajan sobre el campo *kind* del mensaje.

Los operadores relacionales sobrecargados ==, ¡=, >, <, >=, <= llevan a cabo las siguientes comparaciones, devolviendo *true* de ser ciertas, *false* en caso contrario:

== Es equivalente al método *test*, lleva a cabo una comparación exacta del tipo de mensaje con su parámetro.

¡= Devuelve *true* sii el tipo del mensaje es distinto al parámetro indicado.

> Devuelve *true* sii el tipo del mensaje contiene más valores que los indicados por su parámetro.

< Devuelve *true* sii el parámetro indicado contiene más valores que el tipo del mensaje.

>= Devuelve *true* sii al menos todos los valores booleanos indicados en su parámetro están igualmente activos en el tipo del mensaje.

<= Devuelve *true* sii al menos todos lo valores booleanos indicados en el tipo del mensaje están igualmente activos en su parámetro.

Por ejemplo el código:

```
msg.isCommand() && ¡msg.isNull()
```

Si *msg* es un objeto del tipo *Message*, esta expresión se evaluará *true* si el mensaje es de tipo comando (*mt_text* y *mt_command*) y el contenido es diferente a NULL. Otras formas de hacerlo serían:

```
msg>=mt_command && msg.pointer()!=NULL
msg>=mt_command && msg.text()!=NULL
```

Las cuales son absolutamente equivalentes.

7.10.3 La clase CLI

El lenguaje CL, aún poseyendo una sintaxis simple, puede llegar a ser muy complicado de manejar en lenguaje C o C++. Para poder solventar esta dificultad existe la clase *cli*, la cual implementa un interface que pone a disposición del programador de servicio recorrer un texto escrito en CL.

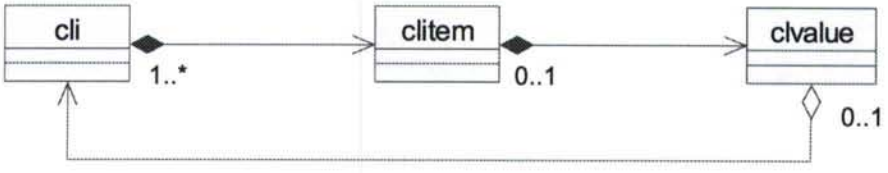

Ilustración 57: Relación entre las clases *cli*, *clitem* y *clvalue*

La clase *cli*, por sí misma no hace nada, es un contenedor de objetos de la clase *clitem*, el cual a su vez puede contener un objeto del tipo *clvalue*. Para habilitar la forma de árbol este último puede contener recursivamente objetos de tipo *cli*.

La clase *cli* contiene los siguientes métodos destacados:

```
cli(const char*);
unsigned count();
clitem operator[] (const int);
clitem operator[] (const char*);

bool match(const char*);
```

El constructor de la clase permite crear un objeto de este tipo a partir de un texto que debe poseer una estructura que cumpla con la sintaxis del lenguaje CL. En caso de error se producirá una excepción.

El método *count* permite conocer el número de objetos *clitem* que contiene y los operadores sobrecargados [ ] permiten recorrer esta lista bien con un índice bien por el nombre del objeto *clitem*. El método match implementa el reconocimiento parcial de un nivel por medio del sub lenguaje CL-*match* (δ7.6.4), devolviendo *true* si acepta este patrón.

Los objetos del tipo *clitem*, permiten acceder al nombre del nodo, el tipo y a su valor:

```
const char*name();
clvalue    value();
cltype_t   type();
bool       test(const char*);
```

Los métodos *name* y *type* devuelven respectivamente el nombre del nodo y su tipo. El método *test* es una ayuda a un proceso que se repite a menudo, la comparación del nombre del nodo con un texto arbitrario. El método *value* devuelve un objeto del tipo *clvalue*, el cual posee las siguientes operaciones:

```
bool    isNone();
bool    isInteger();
bool    isReal();
bool    isCharacter();
bool    isText();
bool    isAddress();
bool    areItems();

long    asInteger();
double  asReal();
int     asCharacter();
char*   asText();
dword_t asAddress();
cli     asItems();
```

Las podemos clasificar en dos categorías: los métodos que informan del tipo de valor que contienen y aquellos que acceden al valor del objeto. Por ejemplo, el método *isNone* indica de ser cierto, que el objeto no contiene ningún valor; el método *areItems* indica si el objeto

contiene a su vez una estructura de tipo *cli*, y por tanto una lista de nodos. Por otro lado los métodos como *asInteger* devuelven el contenido del objeto como un entero largo con signo (*asAddress* lo devuelve sin signo) y asItems devuelve un objeto *cli* susceptible de ser usado recursivamente.

Como ejemplo de uso, el siguiente código forma parte del servicio *deploy*:

```
cl.match("install scheme=<list> ...")
```

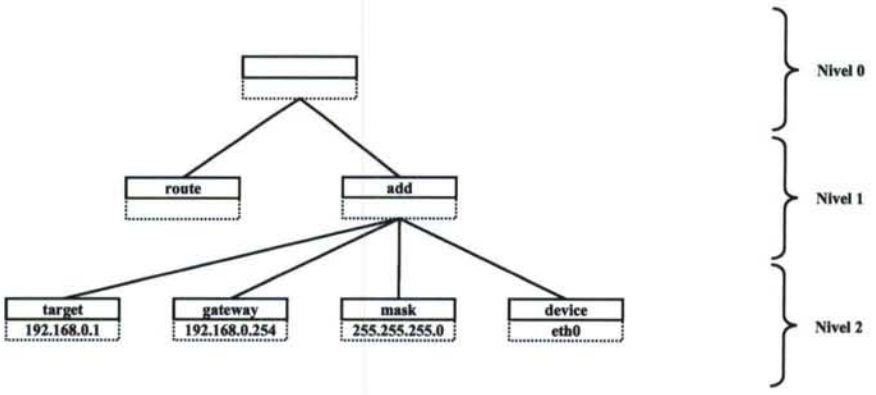

Ilustración 58: Estructura en árbol de un comando CL

Y este otro, para el comando (Ilustración 58):

```
route add=
      {
       target=192.168.0.1,
       gateway=192.168.0.254,
       mask=255.255.255.0,
       device="eth0"
      }
```

La siguiente expresión es cierta:

```
cl.match("route add=<items>")
```

Siendo *cl* un objeto *cli* inicializado con el comando anterior. Así mismo tenemos las siguientes equivalencias:

```
cl["add"].value().asItems()["target"]          192.168.0.1
cl["add"].value().asItems()["gateway"]         192.168.0.254
cl["add"].value().asItems()["mask"]            255.255.255.0
cl["add"].value().asItems()["device"]          "eth0"
```

O más simple, si hacemos:

```
cl2=cl["add"].value().asItems();
```

Podemos:

```
cl2["target"]                    192.168.0.1
cl2["gateway"]                   192.168.0.254
cl2["mask"]                      255.255.255.0
cl2["device"]                    "eth0"
```

7.10.4 La clase *context*

Como hemos indicado previamente (δ7.10.1) todo módulo conteniendo un servicio debe implementar una clase que herede de la clase *context*. El servicio propiamente dicho es la instancia (única) de esta clase y por tanto tiene acceso a los métodos y propiedades públicas y protegidas de *context*.

Esta sección describe el interface accesible a los servicios y que forma el API del NodeOS y los métodos virtuales que la clase derivada debe implementar.

7.10.4.1 Métodos virtuales

Los objetos de tipo *context* son creados en interior del módulo por el mismo, a instancias del control de la capa de ejecución. El constructor y destructor de la clase derivada (vamos a llamarla genéricamente *service*) están a disposición del programador y sólo invocar al constructor de la clase padre (*context*) con el parámetro que el control de la capa de ejecución le pasa (δ7.10.1).

Son cuatro los métodos virtuales que la clase *service* debe implementar como mínimo:

```
virtual  void     run      ()=0;
virtual  bool     dispatch ();
virtual  void     idle     ();
virtual  action_t handler  (Message&,sid_t,mid_t,mid_t,cli cl);
```

De ellas la principal es *run*. Este método es invocado por el control de la capa de ejecución y cuando finaliza, el servicio es eliminado del nodo.

```
sys.mbx.deps.waitfor();
while(dispatch()==false);
```

El texto anterior muestra el código mínimo que este método debe contener. La primera línea bloquea la ejecución del servicio hasta que las dependencias hayan sido satisfechas, la segundo se ejecuta hasta que el método *dispatch* (otro método virtual de *context* que *service* no tiene por que implementar) devuelva *false*, momento en el cual el método *run* finaliza y por tanto el servicio es desinstalado del nodo.

El método *dispatch*, implementado en *context*, bloquea el servicio hasta que haya un mensaje en su *mailbox*; en ese momento lo extrae e invoca al método *handler* (que *service* habrá redefinido) y a su finalización comprueba el valor devuelto (la acción). Si la acción

es *a_reply*, devuelve el mensaje al emisor como una respuesta; si es *a_replyAck*, devuelve un mensaje vacío conteniendo un simple ACK; similarmente si es *a_replyNotAck*, transmite un mensaje vacío con un NACK; finalmente si es *a_none* o *a_replyNothing*, no se devuelve nada al emisor original.

El método virtual *idle* no hace nada. Es invocado cuando somos interrumpidos por una señal pero no hay nada en el buzón del servicio. Es útil para implementar alguna acción que ha de ser periódica pero no determinista ni crítica.

El método *handler* es el método más importante. Es invocado por *dispatch* cuando hay que procesar un mensaje y devuelve una serie de acciones que hemos indicando anteriormente. Sus parámetros son el mensaje recibido, el identificador del servicio emisor, el identificador del mensaje y de la respuesta (si forma parte de una) y finalmente un objeto *cli*, representando el contenido del mensaje en caso de ser este un comando en lenguaje CL.

7.10.4.2 API del sistema

La clase context implementa a su vez un API de acceso controlado a los recursos del sistema. Este interface tiene una estructura jerárquica que, aunque obliga a escribir al más de código, hace mucho más fácil la lectura del mismo.

```
char*    sys.name();
char*    sys.user();
void     sys.finish();
void     sys.abort();
void     sys.yield();
long     sys.elapsed();
void     sys.log(const char*,...);
pid_t    sys.pid();

shmem*   sys.shm.get();
bool     sys.shm.valid(void*);
void*    sys.shm.import(void*,size_t,bool);
void*    sys.shm.malloc(size_t);
void*    sys.shm.realloc(void*,size_t);
void*    sys.shm.free(void*);
void*    sys.shm.memdup(void*);
long     sys.shm.sizeOf(void*);

mailbox*sys.mbx.get();
sid_t    sys.mbx.self();
Message  sys.mbx.message();
Message  sys.mbx.message(dword_t,dword_t,const char*,...);
bool     sys.mbx.wait();
bool     sys.mbx.receive(Message&,sid_t&,mid_t&,mid_t&);
mid_t    sys.mbx.send(sid_t,Message,mid_t=0,long,sid_t=0);
bool     sys.mbx.confirm(sid_t,mid_t,long);
bool     sys.mbx.waitForReply(Message&,sid_t&,mid_t&,mid_t,long);
mid_t    sys.mbx.sendAndWait(const sid_t,Message&,const long,const sid_t);
char*    sys.mbx.resolve(sid_t,long);
sid_t    sys.mbx.resolve(char*,long);

void     sys.mbx.deps.append(char*);
void     sys.mbx.deps.release(char*);
bool     sys.mbx.deps.waitfor();
```

```
SOCKET* sys.socket.create(char*,int);
int     sys.socket.accept(SOCKET*,struct sockaddr*,socklen_t);
int     sys.socket.bind(SOCKET*,struct sockaddr*,socklen_t);
int     sys.socket.connect(SOCKET*,struct sockaddr*,socklen_t);
int     sys.socket.listen(SOCKET*,int backlog);
long    sys.socket.read(SOCKET*,void*,size_t);
long    sys.socket.write(SOCKET*,void*,size_t);
int     sys.socket.close(SOCKET*);
```

Las primitivas del interface están clasificadas en cinco módulos: de sistema (*sys*), de acceso a la memoria compartida (*sys.shm*), de gestión del *mailbox* (*sys.mbx*), de gestión de las dependencias (*sys.mbx.deps*) y de *sockets* VAIN (*sys.socket*):

Respecto al módulo de sistema, *sys.name* devuelve el nombre completo del servicio, incluida la trayectoria. *sys.user* devuelve el nombre del propietario que ha instalado el módulo. La primitiva *sys.finish* provoca que el servicio finalice limpiamente su ejecución, mientras que *sys.abort* obliga a que esta sea realizada. *sys.yield* cede tiempo de procesador y *sys.elapsed* devuelve el número de microsegundos transcurridos desde el inicio del servicio. Para incluir algún texto en el fichero de *log*, la primitiva *sys.log* permite introducir una línea de texto usando un formato idéntico al de *printf*. Finalmente *sys.pid* devuelve el identificador del *thread* (no del proceso) en el que el servicio es ejecutado.

El módulo *sys.shm* permite acceder a los recursos de memoria compartida. *sys.shm.get* devuelve el objeto que controla la memoria compartida con el propósito de tener acceso a las funciones avanzadas del mismo. *sys.shm.valid* indica si un puntero a memoria es un puntero válido en el contexto de la memoria compartida y si representa a un segmento de la misma. *sys.shm.import* importa del gestor de memoria privado un segmento y copia su contenido en la memoria compartida. Las primitivas *sys.shm.malloc*, *sys.shm.realloc* y *sys.shm.free* poseen la misma semántica que sus equivalentes en el estándar POSIX. Finalmente *sys.shm.memdup* y *sys.shm.sizeOf* duplican un segmento de memoria y devuelven su tamaño respectivamente.

De entre las primitivas del módulo *sys.mbx*, la función posiblemente más utilizada es *sys.mbx.send*, encargada de enviar un mensaje, quizá como respuesta a un mensaje anterior. Esta función devuelve el identificador del mensaje enviado, el cual puede ser usado para confirmar que el destinatario ha leído el mensaje por medio del método *sys.mbx.confirm*.

El método *sys.mbx.receive* comprueba si hay algún mensaje al principio del *mailbox* del servicio, de haberlo lo retorna junto con el identificador del emisor, el identificador del mensaje y el identificador, de serlo, del mensaje del que es respuesta. Si no hay mensajes la primitiva retorna inmediatamente devolviendo *false*.

Una forma más refinada de recibir mensajes es esperar por una respuesta a un mensaje enviado previamente. La primitiva *sys.mbx.waitForReply* hace precisamente esto, espera

por la respuesta de un mensaje y entrega el mensaje, el servicio emisor y el identificador del mensaje respuesta.

Finalmente el método *sys.mbx.sendAndWait* unifica las primitivas *sys.mbx.send*, *sys.mbx.wait* y *sys.mbx.waitForReply* en una sola función. Esta envía un mensaje y espera un tiempo determinado a que haya una respuesta explícita de este mensaje.

Finalmente el módulo *sys.socket*, simula el interface *Socket* BSD. En general su semántica es muy parecida, con la salvedad de que el método *sys.socket.create* requiere el nombre del servicio que entregue la funcionalidad requerida y un número que indique el protocolo (en caso de que el servicio implemente más de uno). La funcionalidad de este módulo es adaptar y simplificar el diálogo entre el cliente y el servidor del protocolo.

7.11 Servicios básicos

Nuestra implementación de un nodo VAIN incorpora un conjunto de servicios que podemos considerar básicos, si bien no estrictamente necesarios. Cualquier usuario del nodo con los permisos adecuados puede añadir, cambiar o sustituir algunos de estos servicios con el objeto de adaptar la finalidad del nodo a sus fines. Para nuestro trabajo hemos creado dos módulos, uno de entrada/salida de paquetes IPv4 y otro encargado de distribuir los paquetes hacia los servicios que así lo soliciten siguiendo la técnica mostrada en δ2.3.2, δ4.3 y en δ7.7.

En las siguientes subsecciones detallaremos estos módulos, desde el punto de vista de su uso, no de su implementación. Todos los servicios descritos responden nativamente a los comandos "*quit*" y "*ping*", ambos devolviendo un ACK como respuesta, el primero si se pide una confirmación explícita y el segundo siempre.

7.11.1 Servicio *deploy*

```
Comando: open channel=<string>
```

Este comando abre un canal, cuyo identificador devuelve como valor en el mensaje de confirmación. El valor de *channel* es la expresión de red que debe de cumplir los paquetes que serán desviados hacia el servicio emisor. La expresión es inmediatamente instalada en el esquema y el servicio recibirá el tráfico tan pronto llegue el siguiente paquete.

```
Comando: close channel=<integer>
```

Este comando invoca el cierre del canal, marcándolo como anulado. El demultiplexor no es generado de nuevo. El servicio aún puede seguir recibiendo el tráfico que esté situado en su buzón, pero no desde este momento por causa del demultiplexor.

```
Comando: does exist=<string>
```

Los servicios usan este comando, que devuelve ACK o NACK, para saber si ya está instalado un esquema con esta trayectoria. El parámetro, por ejemplo "*ipv4.tcp*", indica el nombre completo del protocolo.

```
Comando: install scheme=<list> …
```

Los servicios interesados en instalar esquemas de red usan este comando para incorporar nuevos protocolos al esquema de red global. El valor de *scheme* es en lenguaje CL la descripción de la estructura del nuevo protocolo. Los puntos suspensivos indican la posible presencia de un nodo *anchor*, indicando el protocolo bajo el cual el nuevo debe ser instalado. Devuelve ACK o NACK bajo demanda. El nuevo demultiplexor es generado inmediatamente.

```
Comando: uninstall scheme=<string>
```

Este comando realiza la función contraria al anterior, desinstala del esquema global el protocolo indicado por la ruta indicada como valor del nodo *scheme*. El nuevo demultiplexor es instalado inmediatamente.

```
Comando: show channels
```

Este comando tiene un objetivo de soporte, devolviendo en lenguaje CL la lista de los canales actualmente abiertos y su estado.

```
Comando: show scheme
```

Igualmente este comando devuelve la estructura actual del esquema que el demultiplexor soporta.

```
Paquete: <datos binarios>
```

Este servicio recibe un paquete bajo dos circunstancias: desde los servicios de entrada de tráfico (*ip*/*ethernet* por ejemplo) y desde los servicios que lo han procesado. Según sea el origen este servicio obrará de dos maneras diferentes.

Si recibe el paquete de un servicio de entrada de tráfico, este servicio lo somete al demultiplexor y crea la lista de canales/servicios que procesarán en *pipeline* el paquete, incorporando dicha lista al paquete y reenviándolo al primer servicio de la lista.

Si recibe este paquete de algún otro servicio (lo sabe por que el paquete ya posee una lista calculada en el párrafo anterior) lo reenvía al siguiente de la lista.

En caso de que la lista esté vacía (el demultiplexor no ha encontrado nadie interesado en este paquete) o se alcance el final de la misma, el servicio reenvía el paquete a su creador, esto es, al servicio de entrada/salida.

7.11.2 Servicio ip/ethernet

`Comando: open device=<text>`

Abre el dispositivo indicado por *text* y lo habilita para leer y escribir paquetes IPv4 y ARP. Devuelve ACK o NACK en caso de haber sido requerida una confirmación.

`Comando: close device=<text>`

Similarmente al anterior este comando cierra el dispositivo indicando por *text* y elimina de la tabla de enrutamiento sus entradas relacionadas. Devuelve ACK o NACK en caso de haber sido requerida una confirmación.

`Comando: show devices`

Devuelve un texto en lenguaje CL mostrando los dispositivos abiertos y ciertas estadísticas de los mismos.

`Comando: route add=<list>`

Añade a la tabla de enrutamiento una entrada con las características indicadas por *list*. Esta lista se corresponde con "`target=<address> gateway=<address> mask=<address> device=<text>`" en cualquier orden.

`Comando: route del=<items>`

Elimina de la tabla de enrutamiento las entradas que coincidan con la lista la cual es un subconjunto del mostrado en el comando anterior.

`Comando: show routes`

Este comando devuelve en formato CL la tabla de enrutamiento actual.

`Comando: show arp`

De la misma manera podemos ver con este comando la tabla de resolución ARP del nodo.

```
Comando: arp resolve=<address>
Comando: arp request=<address>
Comando: arp resolve=<address> device=<text>
Comando: arp request=<address> device=<text>
```

Todos estos comandos están relacionados con la resolución ARP de la dirección indicada, pudiendo señalar un dispositivo concreto. Con estos comandos se invoca la resolución ARP, si bien no se espera a que esta sea satisfecha. Se supone que en caso afirmativo esta resolución estará realizada al poco tiempo de haber enviado este comando.

```
Comando: allow self device=<text>
```

Indica pata el dispositivo indicado que es posible que este reenvíe tráfico que él mismo ha recibido.

```
Comando: allow self device=<text> no
```

Este comando indica que el dispositivo dado no puede reenviar tráfico que previamente haya recibido.

Este servicio lleva a cabo dos tareas además de las indicadas anteriormente por los comandos que soporta, por un lado lee el tráfico de los dispositivos abiertos (IPv4 y ARP) y los procesa según muestra la Ilustración 59.

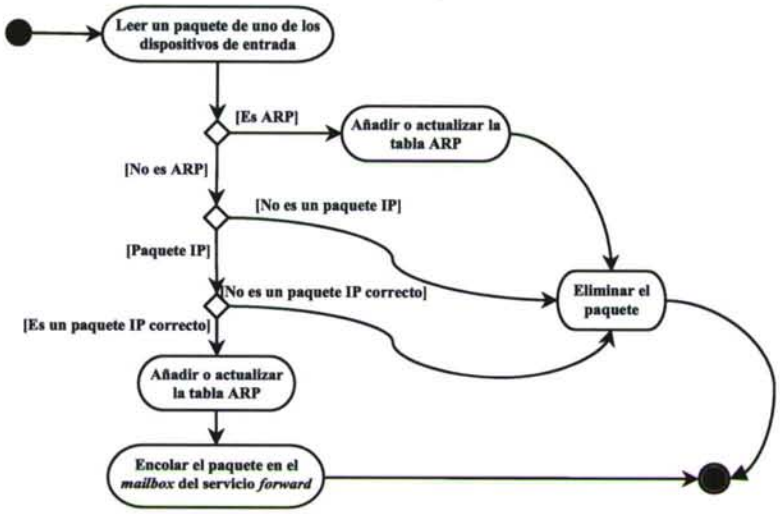

Ilustración 59: Algoritmo de recepción de tráfico desde los dispositivos (*dispatch*)

El proceso de recepción comprueba si es un paquete del tipo ARP, en tal caso añade o actualiza la tabla de resolución ARP. De no serlo y no ser tampoco un paquete IPv4 correcto (*checksum* y campo *version*) lo elimina. En el caso de ser un paquete IPv4 y que su *checksum* es correcto, añade o actualiza la tabla ARP y envía el paquete hacia el servicio *deploy*.

Al recibir un paquete en el buzón del servicio, este lo encamina hacia su destino según al algoritmo mostrado en la Ilustración 60: En primer paso es calcular el siguiente salto

basándose en la tabla de enrutamiento, en caso de no encontrar una ruta, el servicio envía al origen un mensaje ICMP indicando tal situación. De encontrarla decrementa el campo TTL del paquete y si es mayor que cero calcula de nuevo el *checksum* de la cabecera IPv4, para después llevar a cabo la resolución ARP de la dirección de destino o *gateway* (según indique la ruta). Finalmente envía el paquete a través del dispositivo adecuado.

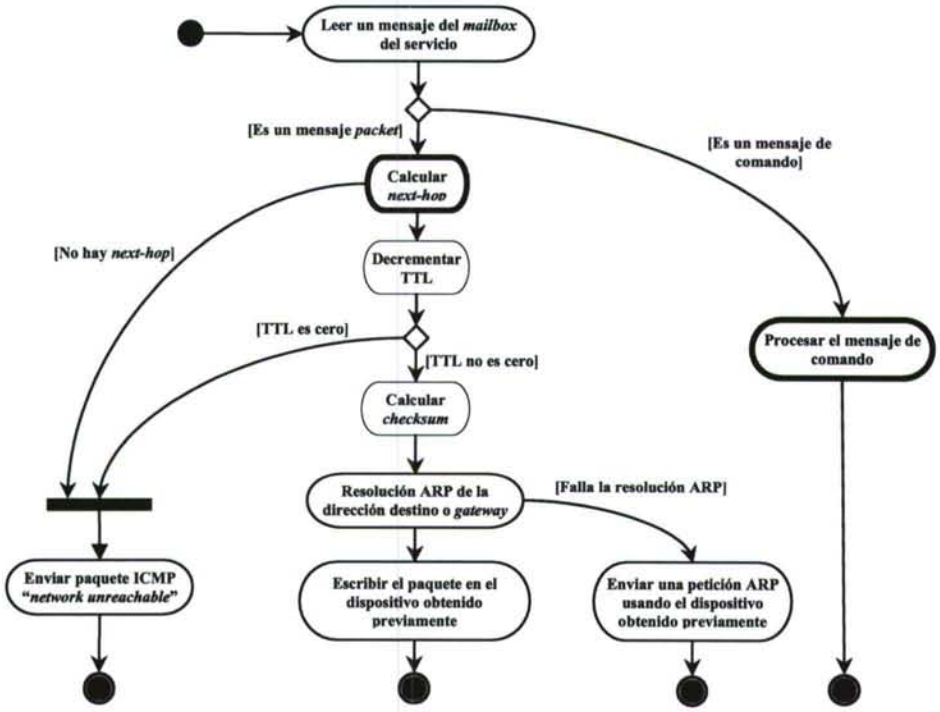

Ilustración 60: Algoritmo de *forwarding* de tráfico hacia los dispositivos

Bibliografía

[Shannon, 48] "*A mathematical Theory of Communication*"; Shannon, Claude; The Bell System Technical Journal; Vol. 27, pp. 379-423, 623-656; July, October 1948.

[Chomsky, 59] "*On certain formal properties of grammars*"; Chomsky, Noam; Information and control; 1959.

[Kleinrock, 61] "*Information Flow in Large Communication Nets*"; Kleinrock, Leonard; RLE Quarterly Progress Report; Massachusetts Institute of Technology; July 1961.

[Hoa, 62] "*Quicksort*", Hoare, C. A. R.; Computer Journal, Vol. 5, 1, 10-15. 1962.

[Mogul, 87] "*The packet filter: An efficient mechanism for user-level network code*"; Mogul, Jeffrey C.; Rashid, Richard F.; Accetta, Michael J.; Proceedings of the 11º ACM symposium on Operating Systems Principles, 39-51, November 1987; ACM Press.

[Jacobson, 88] "*Congestion Avoidance and Control*"; Jacobson, Van; In ACM's SIGCOMM '88; pags. 314-329. August 1988.

[Burkowski, 90] "*Retrieval Performance of a Distributed database Utilising a Parallel Process Document Server*"; Burkowski, F. J.; In Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems; New York, EE.UU; ACM Press.

[Keppel, 91] "*A Case for Runtime Code Generation*"; Keppel, D; Eggers, S. J.; Henry, R. R.; UWCSE; 1991.

[Catlett, 92] "*Metacomputing*"; Catlett, C.; Smarr, L.; Communications of the ACM, Vol. 35, nº 6; June 1992.

[McCanne, 93] "*The BSD packet filter: a new architecture for user level packet capture*"; McCanne, Steven; Jacobson, Van; Proceedings Winter 1993, USENIX conference, 259-269, January 1993; USENIX association.

[Tomasic, 93] "*Performance of Inverted Indices in Shared-nothing Distributed Text Document Information Retrieval Systems*"; Tomasic, A.; García-Molina, H.; In Proceedings of the 2nd International Conference on Parallel and Distributed Informatin Systems; San Diego, California, EEUU; IEEE Computer Society.

[Yahura, 94] "*Efficient packet demultiplexing for multiples endpoints and large messages*"; Yahura, M.; Bershad, B. N.; Maeda, C.; Moss, J. E. B.; Proceedings Winter 1994, USENIX conference, November 1994; USENIX association.

[Coevreur, 94] "*An Analysis of Performance and Cost Factors in Searching Large Text databases using parallel Search Systems*"; Coevreur, T. R.; Benzel, R. N.; Miller, S. F.; Zeitler, D. N.; Lee, D. L.; Singhal, M.; Shivaratri, N.; Wong, W. Y. P.; Journal of the America Society for Information Science.

[Bailey, 94] "*Pathfinder: A pattern based packet classifier*"; Bailey, M. L.; Sarkar, P.; Pagels, M. A.; Peterson, L. L.; Proceedings 1st symposium Operating Systems Design and Implementation, November 1994; USENIX association.

[Engler, 96a] "*VCODE: A portable, very fast dynamic code generation system*"; Engler, R. D.; SIGPLAN Conference on Programming Language design and Implementation (PLDI'96); Philadelphia-USA; 1996.

[Engler, 96b] "*DPF: fast, flexible message demultiplexing using dynamic code generation*"; Engler, R. D.; Kaashoek, M. F.; Proceedings ACM SIGCOMM'96 conference in Computer Communications review, pp 53-59, Stanford University, California; ACM Press.

[Tennenhouse, 96] "*Towards an active network architecture*"; Tennenhouse, D.; Wetherall, D.; Multimedia Computing and Networking 96; January 1996.

[Chan, 96] "*On Realizing a Broadband Kernel for Multimedia Networks*"; Chan, M. C..; Huard, J. F.; Lazar, A. A.; Lim, K. S.; 3rd COST 237 Workshop on Multimedia Telecommunications and Applications; Barcelona, Spain; November 25-27, 1996.

[Cahoon, 96] "*Performance Evaluation of a Distributed Architecture for Information Retrieval*"; Cahoon, B.; McKinley, K. S.; In Proceedings of the 19th ACM-SIGIR International Conference on Research and Development in Information Retrieval; New York, EEUU; ACM Press.

[Yemini, 96] "*Towards Programmable Networks*"; IFIP/IEEE International Workshop on Distributed Systems: Operations and Management; L'Aquila, Italy, October 1996.

[Tennenhouse, 97] "*A Survey of Active Netowork Research*"; Tennenhouse, D. L.; Smith, J. M.; Sincoskie, W. D.; Wetherall, D. J.; Minden, G. J.; IEEE Communications Magazine; January 1997.

[Bhattacharjee, 97] "*An architecture for Active Networking*"; Bhattacharjee, S.; Calvert, K.; Zegura, E.; Proceedings of ICNP'97; Atlanta, GA; October 1987.

[Pfister, 98] "*In search of clusters*"; Pfister, G. F.; 2ª ed.; ISBN-0138997098; Prentice-Hall; 1998.

[Ribeiro-Neto, 98] "*Query Performance for Tightly Coupled Distributed Digital Libraries*"; Ribeiro-Neto, B.; Barbosa, R.; In Proceedings of the 3rd ACM Conference on Digital Libraries; New York, EEUU; ACM Press.

[Campbell, 98] "*A survey of programmable networks*"; Campbell, A. T.; De Meer, H. G.; Kounavis, M. E.; Miki, K.; Vicente, J. B.; Villela, D.; ACM SIGCOM Computer Communications Review, Vol. 12, nº 2, April, 1998.

[Kulkarni, 98] "*Implementation of a Prototype Active Network*"; Kulkarni, A. B.; Minden, G. J., Hill, R.; Wijata, Y.; Gopinath, A.; Sheth, S.; Wahhab, F.; Pindi, H.; Nagarajan, A.; First International Conference on Open Architectures and Network Programming (OPENARCH); San Francisco; 1998.

[Alexander, 98] "*The Switchware Active Network Architecture*"; IEEE Network Special Issue on Active and Controllable Networks; Vol. 12, nº 3; 1998.

[Whetherall, 98] "*ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*"; Wetherall, D.; Guttag, J.; Tennenhouse, D.; Proceedings IEEE OPENARCH'98; San Francisco, CA; April, 1998.

[Angin, 98] "*The mobiware toolkit: Programmable support for adaptativo mobile networking*"; Angin, O.; Campbell, A. T.; Kounavis, M. E.; Liao, R. R. F.; IEEE Personal Communications Magazine, Special Issue on Adaptative Mobile Systems; August 1998.

[Czajkowski, 98] "*A Resource Interface Control for java*"; Proceedings of ACM OOPSA'98; Vancouver-Canada; October, 1998.

[Hawking, 99] "*Methods for Information Server Selection*"; Hawking, D.; Thistlewaite, P.; ACM Transactions on Information Systems.

[Estrin, 99] "*Next Century Challenges: Scalable Coordination in Sensor Networks*"; Deborah Estrin; Ramesh Govindan; John Heidemann; Satish Kumar. Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99), August 1999, Seattle, Washington.

[Schwartz, 99] "*Smart packets for active networks*"; Schwartz, B.; Jackson, W. A.; Strayer, W. T.; Zhou, W.; Rockwell, R. D.; Partridge, C.; 2nd International Conference on Open Architectures and Network Programming (OPENARCH); New York; 1999.

[Hawking, 99] "*Methods for Information Server Selection*"; Hawking, D.; Thistlewaite, P.; ACM Transactions on Information Systems, 17(1), 40-76; 1999.

[Schmid, 00] "*LARA++ design specification*"; Schmid, S.; Lancaster University DMRG Internal report; January 2000.

[Lu, 00] "*Partial Collection Replication versus Caching for Information Retrieval Systems*"; In Proceedings of the 25th ACM-SIGIR Conference on Research and Development in Information Retrieval. New York, EEUU; ACM Press.

[Galis, 00] "*A Flexible IP Active Networks Architecture*"; Galis, A.; Plattner, B.; Smith, J. M.; Denazis, S.; Guo, H.; Klein, C.; Serrat, J.; Karetsos, G. T.; Todd, C.; Proceedings 2nd International working conference, IWAN2000, Japan, November 2000; Springer.

[Merugu, 00] "*Bowman: A Node OS for Active Networks*"; Merugu, Shashidhar; Bhattacharjee, S.; Zegura, E.; Calvert, K.; Proceedings IEEE Infocom 2000, March 2000, 28-46.

[Back, 00] "*Processes in KaffeOS: Isolation, Resource Management and Sharing in java*"; Back, G.; Hsieh, W.; Lepreau, J.; Proceedings 4th OSDI; San Diego-USA; October, 2000.

[Peterson, 01] "*An OS Interface for Active Routers*"; Peterson, L.; Gottlieb, Y.; Hibler, M.; Tullmann, P.; Lepreau, J.; Schwab, S.; Dandekar, A.; Purtell, A.; Hartman, J.; IEEE Journal on Selected Ares in Communications; 2001.

[Tullmann, 01] "*Janos: a Java-oriented OS for Active Network Nodes*"; Tullmann, P.; Hibler, M.; Lepreau, J.; IEEE Journal on Selected Areas of Communication; march 2001.

[Gong Su, 01] "*Virtual active networks: Towars multi-edged network computing*"; Gong Su; Yechiam Yemini. Computing Networks 36, 153-168. 2001.

[RAE, 01] "*Diccionario de la Lengua Española*" 22ª edición. Real Academia Española. 2001.

[Binder, 01] "*Portable Resource Control in Java: The J-SEAL2 Approach*"; Binder, W.; Hulaas, J.; Villazón, A.; ACM OOPSLA'01; Tampa-FL-USA; October, 2001.

[Larrabeiti, 02] "*A practical approach to network based processing*"; Larrabeiti, D.; Calderón, M.; Azcorra, A.; Urueña, M.; IEEE 4th International Workshop on Active Middleware Services; IEEE Computer Society; Edinburgh-Scoland; July, 2002.

[Craswell, 02] "Overview of the TREC 2002 Web Track"; Craswell, N.; Hawking, D.; Voorhees, E. M. & Buckland, L. P. (Ed.); Proceedings of the Eleventh Text Retrieval Conference; Gaithersburg, Maryland (EEUU); NIST Special Publication; 500-251.

[Cacheda, 02] "*Web Directory Advanced Data Architecture, with Optimisation of Restricted Searches to an Area of the Category Graph*"; Cacheda, Fidel; Thesis.

[Puentes, 03a] "*Active Node Implementation in the Context of a Virtual Active Network Orientated to Collaborative Environments*"; The Vision for the Future Generation in research and Applications (CE, 2003); Vol. I, pp 1199-1204; July, 26-30, 2003; Madeira-Portugal; ISBN 9058096254.

[Puentes, 03b] "*3LVAN: Red Activa Virtual de Tres Capas* "; Puentes, F.; XVIII Simposium Nacional Unión Científica Internacional de Radio (URSI, 2003); September, 10-12, 2003; A Coruña-Spain; ISBN 84-9749-081-9.

[Puentes, 03c] "*Architecture for Virtual Active Networks over Internet*"; Puentes, F.; 2nd IASTED International Conference on Communications, Internet & Information Technology (CIIT, 2003); Scottsdale/Arizona-USA; ISBN 0-88986-402-0.

[Puentes, 03d] "*Virtual Active IP Node for collaborative environments*"; Puentes, F.; IASTED International Conference on Communication, Network & Information Security (CNIS, 2003); New York-USA; ISBN 0-88986-402-0.

[Clarke, 04] "Overview of the TREC 2004 Terabyte Track"; Clarke, C.; Craswell, N.; Soboroff, I.; Voorhees, E.M. & Buckland, L. P. (Ed.); Proceedings of the Thirteenth Text Retrieval Conference; Gaithersburg, Maryland (EEUU); NIST Special Publication; 500-261.

[Puentes, 04a] "*Virtual Active IP Node for collaborative environments*"; Puentes, F.; IADIS International Conference Web Based Communities (WBC, 2004); Lisbon-Portugal; March 24-26, 2004

[Puentes, 04b] "*Virtual Active IP Node for collaborative environments*"; Puentes, F.; 6th International Conference on Enterprise Information System (ICEIS, 2004); pp 49-54; Porto-Portugal; April 14-17, 2004; ISBN 972-8865-00-7.

[Puentes, 04c] "*Improving traffic classifiers for active devices*"; Puentes, F.; 2nd IEEE International Conference on Industrial Informatics; Berlin-Germany; June 24-26, 2004.

[Puentes, 04d] "*A new performance evaluation technique for web information retrieval systems*"; Puentes, F.; IADIS International Conference WWW/Internet; Madrid-Spain; October 6-9, 2004.

[Cacheda, 04] "*Performance Analysis of Distributed Architectures to Index One Terabyte of text*"; Cacheda, F.; Plachouras, V.; Ounis, I.; Proceedings of 26th European Conference on Information Retrieval Research (ECIR'04). Lecture Notes on Computer science (2997); pp. 394-408.

[Varghese, 05] "*Network Algorithmics: an Interdisciplinary Approach to Designing fast Networked Devices*"; Varghese, G.; 2005; Elsevier.

[Puentes, 05] "*Virtual Active IP Node*"; Puentes, F.; Carneiro, V.; Cacheda, F.; International Journal of Web Based Communities; Vol. 1, nº 3; ISBN 1477-8394.

[Cacheda, 05] "*A Case Study of Distributed Information Retrieval Architectures to Index One Terabyte of Text*"; Information Processing and Management Journal, 41(5), pp. 1141-1161, 2005.

[Varghese, 05] "*Network Algorithmics*"; Varghese, G.; Morgan-Kaufmann, Elsevier; 2005. ISBN: 0-12-088477-1

[Bison, URL] URL: http://dinosaur.compilertools.net/bison/index.html

[Intel, URL] URL: http://www.intel.com/design/pentium4/manuals

[NASM, URL] URL: http://nasm.sourceforge.net

[Lemon, URL] URL: http://www.hwaci.com/sw/lemon

[YACC, URL] URL: http://dinosaur.compilertools.net/yacc/index.html