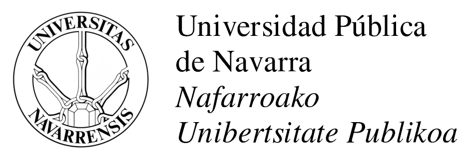


E.T.S. de Ingenieros Industriales y de Telecomunicación
Departamento de Automática y Computación

Tesis Doctoral:

Técnicas Eficientes de Filtrado de Tráfico para Monitorización de Redes de Comunicaciones

Autor:
Eduardo Magaña Lizarrondo

Directores:
Javier Aracil Rico
Jesús Villadangos Alonso

Pamplona, Julio 2001

TESIS DOCTORAL

Técnicas Eficientes de Filtrado de Tráfico para Monitorización de Redes de Comunicaciones

POR

Eduardo Magaña Lizarrondo

Miembros del tribunal

Presidente:	**Dr. Julio Berrocal Colmenarejo**
Secretario:	**Dr. José Ramón González de Mendívil Moreno**
Vocal 1:	**Dr. Arturo Azcorra Saloña**
Vocal 2:	**Dr. Francisco Javier Ruiz Piñar**
Vocal 3:	**Dr. Jorge Martínez Bauset**

Suplentes

1:	**Dr. Juan José Unzilla Galán**
2:	**Dr. Daniel Morató Osés**

Realizado el acto de defensa de la tesis doctoral el día ________________, se acuerda otorgar la CALIFICACION de:

Presidente Secretario Vocal 1 Vocal 2 Vocal 3

TESIS DOCTORAL

Técnicas Eficientes de Filtrado de Tráfico para Monitorización de Redes de Comunicaciones

Eduardo Magaña Lizarrondo
Ingeniero de Telecomunicación

Directores:
Javier Aracil Rico, Doctor Ingeniero de Telecomunicación
Jesús Villadangos Alonso, Doctor Ingeniero de Telecomunicación

Área de Ingeniería Telemática
Departamento de Automática y Computación
Universidad Pública de Navarra

Pamplona. Impreso a fecha 6 de septiembre de 2001.

Agradecimientos

Deseo mostrar mi agradecimiento a toda la gente del Área de Ingeniería Telemática por estar ahí día a día, compartiendo ideas, inquietudes y momentos para recordar. En especial, a mis tutores de investigación Javier Aracil Rico y Jesús Villadangos Alonso por su confianza, atención continuada y absoluta disponibilidad.

La implementación de las herramientas de monitorización ha sido posible gracias a la inestimable colaboración de los siguientes estudiantes de ingeniería de telecomunicación. Para el sistema PROMIS, José Javier Ruíz Cobo e Iñaki Astrain Escola. Para el sistema MONET, Manuel Prieto Miguez.

Además, quisiera agradecer a toda mi familia su apoyo incondicional durante todos estos años.

Este trabajo va dedicado a ti, Edurne. Tu paciencia, comprensión y apoyo me han facilitado todo.

Índice General

1 Filtrado en sistemas de monitorización **1**
1.1 Introducción . . . 1
1.2 Protocolos en redes de datos . . . 2
1.3 Sistemas de monitorización . . . 4
1.3.1 Características de un sistema de monitorización . . . 5
1.3.2 Arquitectura de un sistema de monitorización . . . 7
1.3.3 Paradigmas de monitorización . . . 9
1.3.3.1 Analizadores de protocolos . . . 9
1.3.3.2 Plataformas SNMP . . . 10
1.3.3.3 Monitorización distribuida . . . 11
1.3.3.4 Comparativa . . . 12
1.4 Problemática del filtrado de paquetes . . . 12
1.5 Arquitecturas de filtrado . . . 15
1.5.1 Packet filter . . . 16
1.5.1.1 CSPF . . . 18
1.5.1.2 BPF . . . 18
1.5.1.3 MPF . . . 20
1.5.1.4 PathFinder . . . 21
1.5.1.5 DPF . . . 22
1.5.1.6 BPF+ . . . 22
1.5.1.7 Conclusiones de los packet filter . . . 24
1.5.2 Clasificación de paquetes en routers de alta velocidad . . . 26
1.5.2.1 Arquitectura de routers de alta velocidad . . . 27
1.5.2.2 Basados en árbol . . . 29
1.5.2.3 Basados en comparaciones . . . 34
1.5.2.4 Basados en hashing . . . 34
1.5.2.5 Basados en la combinación de hashing y árbol . . . 34
1.5.2.6 Otros esquemas de enrutamiento: basados en etiquetas . . . 36
1.5.2.7 Conclusiones de clasificación en routers . . . 36
1.5.3 Pila de protocolos en Sistemas Operativos . . . 37
1.5.4 NNstat . . . 37
1.6 Objeto del presente trabajo . . . 38

2 Algoritmo de filtrado PAM-Tree **41**
2.1 Introducción . . . 41
2.2 Conceptos previos . . . 42
2.3 Especificación . . . 46
2.4 PAM-Tree como algoritmo de filtrado . . . 47
2.4.1 Estructura del PAM-Tree . . . 48
2.4.2 Nodos subfiltro . . . 51
2.4.3 Nodos parámetro . . . 53
2.4.4 Concatenación de subfiltros . . . 54
2.4.5 Operaciones en PAM-Tree . . . 55
2.5 Formalización del PAM-Tree . . . 57
2.5.1 Estados del autómata PAM-Tree . . . 57
2.5.2 Signatura de acciones del autómata . . . 60
2.5.3 Transiciones del autómata . . . 60
2.5.3.1 Acciones de entrada . . . 61
2.5.3.2 Acciones de salida . . . 61
2.5.3.3 Acciones internas . . . 61
2.5.4 Descripción de transiciones . . . 63
2.5.5 Propiedades del sistema de filtrado PAM-Tree . . . 67
2.6 Conclusiones . . . 74

3 Análisis de prestaciones **75**
3.1 Introducción . . . 75
3.2 Evaluación analítica del PAM-Tree . . . 75
3.2.1 Modelo de filtrado . . . 76
3.2.2 Análisis de prestaciones para dos niveles de filtrado . . . 79
3.2.2.1 Sistema compuesto . . . 80
3.2.2.2 Sistema fragmentado, PAM-Tree . . . 82
3.2.2.3 Comparativa del tiempo de servicio . . . 89
3.2.2.4 Comparativa del tiempo de espera en cola . . . 91
3.2.2.5 Condiciones de filtrado real . . . 92
3.2.3 Generalización para más de dos niveles de filtrado . . . 94
3.2.4 Conclusiones . . . 99
3.3 Comparativa experimental . . . 101
3.3.1 Introducción . . . 101
3.3.2 Midiendo el consumo de CPU del Kernel . . . 101
3.3.3 Sistemas de monitorización PAM-Tree y LSF . . . 102
3.3.4 Comparativa PAM-Tree / LSF . . . 103
3.3.5 Implementación BPF . . . 105
3.3.6 Conclusiones . . . 107

4 Implantación industrial **109**
4.1 Introducción . . . 109
4.2 Sistema PROMIS . . . 110
4.2.1 Funcionalidades del sistema . . . 110

4.2.2 Sonda 111
4.2.3 Consola 112
4.2.4 Implantación 114
4.3 Sistema MONET 114
4.3.1 Funcionalidades del sistema 115
4.3.2 Sonda 116
4.3.3 Consola 118
4.3.4 Implantación 118

5 Conclusiones y trabajos futuros 127
5.1 Conclusiones 127
5.2 Líneas de trabajo futuras 128

A Algoritmos de búsqueda/clasificación generales 131
A.1 Introducción 131
A.2 Búsqueda secuencial 132
A.3 Búsqueda por comparación de claves 132
A.3.1 Búsqueda binaria 132
A.3.1.1 Búsqueda binaria uniforme 133
A.3.1.2 Búsqueda de Fibonacci 134
A.3.1.3 Búsqueda por interpolación 134
A.3.2 Búsqueda por árbol binario 134
A.3.3 Árboles balanceados 135
A.3.4 Árboles multicamino 135
A.4 Búsqueda digital 136
A.5 Búsqueda por hashing 136
A.6 Resumen 137

B Implementación de PAM-Tree 139
B.1 Introducción 139
B.2 Consideraciones 139
B.3 Inserción, borrado, consulta y actualización 143

C Publicaciones y proyectos 149
C.1 Publicaciones 149
C.2 Trabajos pendientes de revisión 149
C.3 Patente 150
C.4 Proyectos 150
C.5 Informes técnicos 151

Índice de Figuras

1.1 Arquitectura de un sistema de monitorización de redes de datos 7
1.2 Escenario de monitorización de red usando sondas distribuidas 11
1.3 Relación del packet filter con otros elementos del sistema 17
1.4 Ejemplo de funcionamiento de un packet filter como BPF 19
1.5 Grafo acíclico correspondiente al filtro de la izquierda de la figura 1.4 19
1.6 Arquitectura de router centralizado 27
1.7 Arquitectura de router distribuido 28
1.8 Arquitectura de router paralelo 28
1.9 Ejemplo de estructura del árbol Patricia 29
1.10 Ejemplo de tabla de rutas usando el algoritmo Patricia 30
1.11 Árbol binario 31
1.12 Árbol Patricia correspondiente al árbol binario de la figura 1.11 31
1.13 LC-Trie correspondiente al árbol binario de la figura 1.11 32
1.14 Codificación de nodos para la definición del árbol multiresolución 33
1.15 Ejemplo de árbol multiresolución 33
1.16 Codificación de los subárboles T_1 y T_2 33

2.1 Ejemplo de cabeceras y campos 43
2.2 Entorno y subsistema de filtrado 47
2.3 Ejemplo de árbol de filtrado PAM-Tree 49
2.4 PAM-Tree y el sistema de monitorización 50
2.5 Ejemplo de estructura PAM-Tree para un filtrado AND/OR 56
2.6 Variables globales y componentes de la estructura en árbol del autómata . . . 58
2.7 Autómata y sus acciones 60

3.1 Modelo del clasificador 76
3.2 Campos de un paquete a filtrar 77
3.3 Ejemplo de relación entre el árbol de filtrado y el modelo propuesto 77
3.4 Unión de nodos 78
3.5 Modelo para 2 nodos del árbol 79
3.6 Filtros formados entre nodos 79
3.7 Filtros formados entre nodos a) para el caso mejor y b) para el caso peor . . . 80
3.8 Sistema fragmentado y compuesto para 2 niveles de filtrado 80
3.9 Filtros establecidos con el nodo i del primer nivel 84
3.10 Modelo de establecimiento de filtros entre subfiltros a 2 niveles 84

3.11 Tiempo de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 25 filtros 90
3.12 Tiempo de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 100 filtros 90
3.13 Probabilidad de que el sistema fragmentado sea mejor que el compuesto en función del número de filtros simultáneos n 91
3.14 Tiempo de espera en cola para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 25 filtros 92
3.15 Tiempo de espera en cola para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 100 filtros 93
3.16 a) Tiempo medio de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n para un filtrado dirección IP origen - dirección IP destino b) Tiempo medio de espera en cola correspondiente 94
3.17 Modelo para el caso de 3 filtros 95
3.18 Tiempo medio de servicio para a) sistema fragmentado y compuesto en función del número de filtros simultáneos n para un filtrado de 2 y 3 niveles b) sistema de 3 niveles y mayor n 100
3.19 Tiempo medio de espera en cola para 3 niveles $K_1 = K_2 = K_3 = 5, O_1 = O_2 = O_3 = 10$ 100
3.20 Consumo de CPU en función del número de filtros y su naturaleza a) PAM-Tree b) LSF 104
3.21 Consumo de CPU en función de la carga de red a) PAM-Tree b) LSF 105
3.22 Consumo de CPU en función del tamaño de paquete a) PAM-Tree b) LSF . . 106
3.23 Comparativa de consumo de CPU PAM-Tree/BPF en un sistema FreeBSD frente al número de filtros establecidos 106

4.1 Estructura en bloques de la sonda PROMIS 111
4.2 Árbol de filtrado en PROMIS 112
4.3 Consola PROMIS sobre Windows NT 113
4.4 Consola PROMIS sobre WWW 113
4.5 Estructura en bloques de la sonda MONET 116
4.6 Interfaz de consola MONET 119
4.7 Implantación de sondas MONET en la cabecera de la central de RETENA . . 119
4.8 Carga de la red semana 6-12/12/2000 a) Sonda Ethernet b) Sonda NetFlow . . 120
4.9 a) Carga de la red del día 6/12/2000 00:00-2:36h b) Test de varianza frente a nivel de agregación 121
4.10 Bps WWW 122
4.11 Número de usuarios simultáneos 123
4.12 Bps IP destino 10.1.9.53 123
4.13 Bps aplicación dls-monitor 124
4.14 Bps aplicación bootps 124
4.15 Tráfico durante 4 semanas 125

A.1 Ejemplo de árbol binario 133
A.2 Ejemplo de búsqueda binaria uniforme 133

A.3 Árbol de Fibonacci de orden k=5 134
A.4 Ejemplo de búsqueda digital 136

B.1 Campos de los nodos de PAM-Tree 141
B.2 Estructura esquemática del PAM-Tree de la figura B.1 142

Índice de Tablas

1.1 Características de sistemas de monitorización de red 13
1.2 Comparativa de las características de los packet filters 26
1.3 Array que representa el LC-trie anterior 32
1.4 Complejidad de varios algoritmos de búsqueda en tablas de enrutamiento . . . 36

3.1 Comparativa para un caso real de filtrado IP-TCP 93

4.1 Puertos TCP origen 122
4.2 Puertos UDP origen 122
4.3 TOP direcciones IP 124
4.4 TOP protocolos IP 125
4.5 Horas cargadas por día 126

A.1 Comparativa de algoritmos de búsqueda 137

Resumen

Las necesidades de intercambio de datos han crecido de manera espectacular en los últimos años y con ello la infraestructura de telecomunicaciones asociada. Además, la aparición de nuevos servicios ha requerido cada vez un control más estricto de la red. Por todo ello, la gestión de las redes de comunicaciones se ha convertido en un campo de actuación muy importante. Dentro de él se pueden enmarcar los sistemas de monitorización que ofrecen información detallada del tráfico que circula por las redes. En el presente trabajo se presenta un algoritmo de filtrado de paquetes con aplicación a los sistemas de monitorización de redes de datos que poseen una problemática particular. En concreto, el número de filtrados simultáneos que se suele requerir a los sistemas de monitorización es elevado.

En una primera parte se presentan las diferentes alternativas existentes en la actualidad para el filtrado de paquetes. Por un lado están los sistemas packet filter, optimizados para filtrado de un único flujo de paquetes. Por otro lado existen multitud de propuestas para clasificación de paquetes en routers de alta velocidad, que sin embargo se centran únicamente en campos muy concretos del paquete como las direcciones IP destino para realizar la clasificación. También existen sistemas de filtrado o clasificación de paquetes en los propios sistemas operativos, para poder proveer de diferentes puntos de acceso a servicios a las aplicaciones de la misma máquina y para realizar el encaminamiento cuando la máquina posea varios interfaces de red. Existen escasas propuestas específicamente orientadas a sistemas de monitorización.

Posteriormente se realiza una propuesta de técnica de filtrado que se aprovecha de las peculiaridades de los sistemas de monitorización: el algoritmo PAM-Tree. Su característica principal es la reutilización de bloques de subfiltrado con lo que se soporta de manera más adecuada un mayor número de filtros simultáneos. Además se incorporan en la propia estructura de filtrado los parámetros de monitorización por lo que la actualización de parámetros es inmediata a la vez que se realiza el proceso de filtrado de cada paquete. Tras su descripción, se realiza una formalización mediante teoría de autómatas que se aplica a la demostración de las propiedades del algoritmo propuesto.

Una vez presentado el algoritmo, se pasa a su estudio analítico, modelando el coste de filtrado de un paquete como el tiempo de servicio en un sistema de colas M/G/1. Al comparar el modelo del algoritmo PAM-Tree con otro aplicable a los sistemas de tipo packet filter, se comprueba la mejora obtenida con PAM-Tree conforme crece el número de filtros. Tras ello, una comparativa experimental entre PAM-Tree y los packet filter BPF/LSF nos dará una visión más real del comportamiento del algoritmo propuesto sobre implementaciones de sistemas de monitorización. Los resultados experimentales obtenidos validan el modelo analítico considerado.

El algoritmo propuesto se ha implementado en dos sistemas de monitorización implantados sobre las redes de datos de una empresa fabricante de coches y una operadora de cable regional, en dos versiones, una primera con funcionalidades recortadas. El algoritmo de filtrado es el núcleo fundamental de estas herramientas y su funcionamiento determinará la flexibilidad en la definición de parámetros de monitorización de red por parte del gestor.

Abstract

The data exchange needs have grown in a phenomenal way during the last years and, consequently, the associated telecommunications infrastructure has also developed at an extraordinary pace. Besides, a myriad of new services have recently emerged, requiring an even stricter control of the network. Therefore, communication networks management has become a fundamental field in data networking. Network monitoring systems are a part of the network management systems since they offer detailed information about the traffic that is carried by the network. In this work, we present an algorithm for packet filtering with application to network monitoring systems. Such algorithm tackles the specific problems of such systems. More specifically, the number of simultaneous filters that are required is usually high.

In the first part, we present the different alternatives currently available for packet filtering. On one hand, packet filters are optimized for filtering a unique packet flow. On the other hand, a large number of packet classification schemes for high-speed routers have been proposed, but, however, their scope is limited to filtering certain packet fields such as destination IP address. Packet filtering systems or classification systems are also used in operating systems, in order for them to provide different service access points to the applications running in the same machine and to carry out the routing tasks if the machine has several network interfaces. The state of the art reveals that there are scarce proposals oriented to monitoring systems.

Precisely, due to the scarcity of filtering techniques that fulfill the needs of traffic monitoring systems, we propose a filtering technique tailored to the requirements of network monitoring systems: the PAM-Tree algorithm. Its main characteristic is the reuse of subfilter blocks, which make it possible to support a larger number of simultaneous filters in a more efficient way. It also incorporates the monitoring parameters in the same filtering structure so that parameter updates and packet filtering take place jointly in the same data structure. A formalization using Input/Output automata theory is performed. Finally a possible implementation is presented.

Once the algorithm is explained, we perform an analytical study, modelling the cost of packet filtering as the service time of a M/G/1 system. We compare the model of PAM-Tree algorithm with other models which also apply to monitoring systems, such as the packet filter. Both the CPU load and processing time for incoming packets increase in a sublinear fashion, whereas packet filters provide linear increase. Furthermore, an extensive empirical comparative analysis between PAM-Tree and packet filters BPF/LSF is performed. Both PAM-tree and packet filter are implemented in a PC and tested with real traffic input, showing good agreement with theoretical analysis.

Finally, the proposed algorithm has been implemented in two monitoring systems which are currently operative in regional cable operators. Two software releases have been issued, the first one with clipped functionalities. From our experience we can state that the filtering algorithm is the cornerstone of network monitoring tools and, thus, an adequate performance is of fundamental importance to guarantee the flexibility and accuracy. The PAM-tree algorithm presented in this Doctoral Dissertation provides an efficient filtering algorithms that serves as a platform to develop high-speed network monitoring systems.

Capítulo 1

Filtrado en sistemas de monitorización

1.1 Introducción

Las redes de comunicaciones se han convertido en un elemento fundamental de la sociedad actual. En concreto, las redes de datos son importantes no sólo para los abonados finales sino también para las grandes redes corporativas. Además, las tecnologías de red evolucionan rápidamente, haciendo que las pequeñas redes locales aisladas se conviertan en grandes sistemas de información multiservicio [1, 2] con acceso de manera simultánea a servicios de voz (como VoIP, Voice over IP), datos, imagen y vídeo (como VOD, Video On Demand). El número de máquinas interconectadas por redes crece de forma exponencial y además éstas se conectan a Internet por troncales de acceso cuyas velocidades también crecen rápidamente. Los servicios clásicos de Internet como WWW, correo electrónico, FTP o Telnet se completan ahora con otros como telefonía o vídeo que poseen mayores requerimientos de recursos de red y en los que la disciplina de servicio *Best Effort* [3] no es suficiente. Como factor añadido, es de sobra conocido que el tráfico de redes de datos sigue modelos complejos de generación, no tan simples como el tráfico telefónico que sigue un proceso de generación de Poisson y una distribución exponencial de la duración de las llamadas [4]. El tráfico de redes de área local [5], redes de área extensa [6] e Internet [7] se ajustan a modelos estadísticamente autosimilares (fractales) caracterizados por ráfagas en un gran rango de escalas de tiempo. Esto hace que el tráfico de redes de datos no tenga una disminución de la varianza tan rápida al aumentar la escala de observación a diferencia del tráfico telefónico, manteniéndose las ráfagas en todas las escalas. Incluso con un tráfico de incrementos independientes se añade el problema de la no estacionariedad. Por tanto, se complica el dimensionamiento de los recursos de red.

Las redes de datos se convierten de esta forma en un elemento estratégico y como tal se hace necesaria su supervisión, monitorización y gestión. Estas tareas se han dejado tradicionalmente en segundo plano debido a los costes del hardware específico necesario y de la escasez de personal especializado. Sin embargo, cada vez más se da a estas tareas la importancia que merecen: en todo momento se desea llevar cuenta de lo que está sucediendo en la red y comprenderla lo suficiente como para optimizar su funcionamiento. Así, puesto que estas redes de datos constituyen una parte fundamental de cualquier actividad particular o empresarial es preciso asegurar su disponibilidad mediante un sistema de monitorización de red.

Dentro de un sistema de monitorización, la parte que determina la eficiencia de funcionamiento es en gran medida el mecanismo de filtrado utilizado. Para poder obtener parámetros de monitorización es necesario poder aplicar filtros sobre los paquetes de la red de datos que permitan llevar cuenta de la ocurrencia de determinados eventos. Así, por ejemplo, si se quiere medir la carga de tráfico generada por determinada dirección IP origen es necesario establecer un filtro que compruebe que se trata de un paquete IP y que el campo dirección IP origen coincide con el solicitado. Si el número de estos filtros fuese pequeño la eficiencia del filtrado no tendría mucha importancia en el comportamiento del sistema de monitorización. Sin embargo, el número de filtros simultáneos que ha de soportar un sistema de este tipo en una red mediana es de miles de filtros simultáneos. Imaginar por ejemplo que se desea contabilizar el tráfico intercambiado entre todo par de máquinas de la red (matriz de tráfico). Si fuesen 100 máquinas esto supondría 10.000 filtros, por lo que el filtrado va a ser el elemento del sistema de monitorización que limitará las capacidades del sistema.

Los sistemas de monitorización comerciales permiten al usuario definir un pequeño número de filtros, dejando la mayor parte de las funcionalidades en manos de filtros predefinidos y, por tanto, con código optimizado al efecto. Sin embargo, lo que interesa en un sistema de monitorización es flexibilidad total a la hora de definir los parámetros de interés y por tanto en la definición de los filtros. En este trabajo se presenta una técnica de filtrado que permite definir filtros de manera flexible sin que ello suponga una sobrecarga extra. Como se verá más adelante, se han propuesto técnicas de filtrado de paquetes como los packet filters para filtrado de paquetes en sistemas UNIX o algoritmos de clasificación en routers de alta velocidad. Nuestra propuesta estará relacionada con estas técnicas y supone un avance importante respecto a las existentes.

La Tesis Doctoral aborda el tema de la monitorización de todo tipo de redes, desde redes de área local (LAN) a enlaces troncales de Internet pasando por redes corporativas y, en concreto, los algoritmos de filtrado a implementar en las herramientas de monitorización. El trabajo se estructura del siguiente modo. En el presente capítulo se muestran los antecedentes y necesidades de la propuesta, abordando las características de los sistemas de monitorización. También se detallan los problemas en el filtrado de paquetes y el estado del arte en técnicas de filtrado de paquetes. En el capítulo 2 se describe y presenta la formalización de la propuesta de algoritmo. En el capítulo 3 se muestran los resultados del análisis teórico y experimental de la propuesta y en el capítulo 4 se describe su aplicación industrial. Finalmente se presentan las conclusiones.

1.2 Protocolos en redes de datos

Las redes de ordenadores modernas se diseñan basándose en el concepto de jerarquía de protocolos a capas, estandarizado por el modelo OSI (Open Systems Interconnections) que fue propuesto por la ISO (International Standards Oganization) en 1984. Este modelo se basa en los siguientes principios [8, capítulo 3] [9, capítulo 1]:

- Una descomposición lógica de las redes complejas en partes más pequeñas y entendibles (7 capas).

- Interfaces estándares entre funcionalidades de red.

- Simetría en las funciones realizadas en cada nodo. Cada capa realiza las mismas funcionalidades de su capa gemela del otro nodo de red.

- Un lenguaje estándar que facilite la comunicación entre equipos y aplicaciones diferentes.

Dentro de la estructura en niveles, los datos de una capa N, SDU_N (Service Data Unit), junto con la información propia de la capa que es la cabecera, PCI_N (Protocol Control Information), forman una PDU (Protocol Data Unit) que se pasará a la siguiente capa $N-1$ para repetir el proceso. De esta manera, la PDU va acumulando cabeceras correspondientes a cada nivel, que es lo que se denomina *encapsulación de datos*.

Las capas que se añaden en el nodo transmisor se usan para invocar las funcionalidades simétricas de la misma capa en el nodo receptor, y en aquel, estas cabeceras se eliminan para pasar la PDU a niveles superiores.

Las unidades de información reciben diferente nombre según el nivel OSI de que se trate [10, capítulo 1]: trama a nivel físico, paquete a nivel de enlace, datagrama a nivel de red y datos a nivel superior. En este trabajo consideraremos que la unidad básica de información que se transmite por una red se estructura en un contenedor que llamaremos *paquete*, que es el nombre normalmente considerado en la literatura frente a su correcta denominación de trama.

Normalmente, en cada nivel existe un campo en la cabecera que indica el tipo de protocolo que se encapsula en el siguiente nivel. Claro está, para el nivel inferior (primer nivel) esto no puede ser: para conocer el protocolo de nivel inferior éste dependerá del interfaz de red utilizado. Pero hay redes en las que existen varios encapsulados posibles de nivel MAC (Medium Access Control), por ejemplo en una Ethernet, por lo que la detección del tipo de encapsulado dependerá de valores de la propia cabecera. El resto de campos de la cabecera de nivel de protocolo serán independientes del resto de protocolos por lo que este hecho facilitará el filtrado necesario en el sistema de monitorización.

Pongamos como ejemplo el encapsulado Ethernet. Existen 4 tipos de paquetes Ethernet (Ethernet II, Ethernet 802.3, Ethernet 802.2 y Ethernet SNAP), que tienen en común los campos preámbulo, dirección MAC origen, dirección MAC destino y el checksum. El criterio para identificar uno u otro tipo de paquete se basa en realizar las siguientes comprobaciones sobre campos de la misma cabecera Ethernet:

- Si el valor del campo Ethertype/Length es mayor que 0x05DC (1500 en decimal) se trata de un paquete Ethernet II. Por ejemplo, el valor de este campo 0x0800 corresponde a IP y 0x8137 a IPX, ambos sobre Ethernet II.

- Si los bytes 0xFFFF siguen al campo Longitud, el paquete es de tipo 802.3 con tráfico IPX/SPX de Netware.

- Si el byte que sigue al campo Longitud (DSAP) es 0xAA se trata de un paquete Ethernet SNAP.

- En el resto de casos se tratará de un paquete Ethernet 802.2.

El sistema de monitorización va a procesar esos paquetes. La selección de paquetes en la red se realiza mediante la definición de filtros. Un filtro es el testeo de una serie de condiciones que han de cumplir los campos de un paquete. Por tanto, el sistema de monitorización se puede aprovechar de la jerarquía en niveles, desencapsulando los datos nivel a nivel gracias a la independencia entre niveles. También será necesario incorporar al sistema de monitorización funcionalidades especiales, como la mostrada para diferenciar el tipo de trama Ethernet o, como se verá posteriormente, para tratar la fragmentación.

1.3 Sistemas de monitorización

La monitorización y el análisis de red pueden ser definidos como la *captura, filtrado, decodificación y organización de las observaciones que se realizan sobre el funcionamiento de los servicios de una red* [11]. Por tanto, los sistemas de monitorización son útiles para conocer el estado de la red y proponer acciones correctivas incluso antes de producirse los problemas. Distinguimos dos modos de monitorización según el grado de control sobre la disponibilidad de la red y son los siguientes:

- Reactivo, una vez se ha producido un problema en la red ser capaz de localizarlo y solucionarlo con rapidez. Será adecuado un sistema de monitorización que ofrezca herramientas potentes de alarmas y filtrado.

- Preventivo, antes de que los usuarios detecten problemas en la red ser capaz de preveerlos y actuar en consecuencia. Un sistema de monitorización continua que ofrezca herramientas de informes periódicos y sistemas expertos de interpretación de datos satisface este modo de acción.

La necesidad de captura de paquetes y su análisis apareció con la llegada de las redes de área local. Conforme se extendió el uso de redes Ethernet, se fue haciendo indispensable la utilización de monitorizadores de red. El CMU/Standford Packet Filter (CSPF) [12], desarrollado en 1980, es el primer packet filter sobre UNIX. Éste evolucionó a soluciones similares en otras plataformas como el Ultrix packet filter en DEC [12], NIT (Network Interface Tap) bajo SunOS [13] y BPF (BSD Packet Filter) en BSD [14].

Aparecieron entonces multitud de programas en UNIX que a nivel de usuario imprimían cabeceras de los paquetes. Sun implementó NIT para capturar paquetes y Etherfind para imprimir sus cabeceras. Los sistemas UNIX proveían ventajas por la gran variedad de herramientas disponibles para la manipulación y análisis de las trazas de paquetes.

Tcpdump [15] es probablemente la herramienta de monitorización más popular en sistemas UNIX. Su primera versión data de 1989 y fue incluida en BSD Net Release2 en 1991. *Tcpdump* está basado en BPF como mecanismo de filtrado originariamente. Para poder ser usado en otros sistemas operativos se desarrolló la librería *pcap* [16] que proporciona un interfaz de captura y filtrado de paquetes independiente de la plataforma. Desde 1999 un grupo de voluntarios mantiene el código de estas herramientas [17].

Posteriormente, los sistemas de monitorización de alta velocidad empiezan a ser explorados por la herramienta OC3MON [18], basada en un hardware PC y exclusiva para redes ATM. CoralReef es un paquete desarrollado por CAIDA [19] para analizar la salida de OC3MON. También se han desarrollado herramientas para el análisis de redes a largo plazo como el Statspy-NNStat [20] o basadas en SNMP como MRTG [21] y RRDtool [22]. Otra herramienta de procesado es el Network Animator [23] que es capaz de trabajar con trazas de *tcpdump* o *ns* [24].

En la actualidad la variedad de equipos de monitorización hardware o software es inmensa. Desde software que en un PC permite capturar paquetes y decodificar campo a campo su contenido para una red de área local a grandes sistemas de gestión que permiten monitorizar el estado de cada segmento y equipo de red, pasando por equipos de monitorización de redes de alta velocidad.

Un sistema de monitorización provee al gestor de una serie de parámetros de red que le son útiles para llevar cuenta de la carga de la red, calidad de servicio ofrecida y problemas que puedan surgir a los usuarios. Básicamente consisten, en primer lugar, en un hardware que recoge todos los paquetes de la red a la que está conectada. Posteriormente se procesan estos paquetes, filtrando aquellos en que se esté interesado. Según los paquetes que superan los filtros se actualizan los parámetros de monitorización y finalmente se presentan las estadísticas obtenidas al usuario a través de un interfaz adecuado.

1.3.1 Características de un sistema de monitorización

La cantidad de información que se desea transmitir a través de las redes de datos es inmensa y crece continuamente. Para poder aprovechar la capacidad de la red al máximo se hace necesario conocer en profundidad el estado de la misma y, para ello, los sistemas de monitorización son una herramienta fundamental. Su utilidad queda demostrada para las siguientes tareas:

- Problemas de red. Un sistema de monitorización ayuda a detectar máquinas que generan tráfico erróneo (por ejemplo, por fallo de la tarjeta de red), segmentos con muchos fallos de CRC (por ejemplo, por interferencias en el cableado) o alto número de colisiones (por ejemplo, por excesiva carga en la red o distancia entre máquinas fuera del estándar).

- Optimización de red. Según el tráfico que se observe se puede, por ejemplo, intentar unificar los protocolos utilizados y minimizar los paquetes de difusión (*broadcast*) para aumentar la eficiencia de la red. Se podrá conocer del mismo modo los mayores usuarios de red para tratarlos de forma especial, o los destinos más habituales para optimizar la conectividad.

- Diseño de red. Ante una futura expansión, los datos de monitorización ayudan a un correcto diseño de los segmentos de red, capacidades, dimensionamiento y conectividad. El conocimiento de la evolución del tráfico en los últimos tiempos de una red ayudará a estimar su crecimiento en el futuro.

- Seguridad. Se podrá tener conocimiento de primera mano de intrusiones en la red o de la existencia de servidores no autorizados, para tomar las medidas oportunas. La detección de ataques o la existencia de sniffers no autorizados en la red (que puedan estar capturando información confidencial) también será posible gracias a la utilización de un sistema de monitorización.

Un sistema de monitorización se caracteriza por sus capacidades internas (precisión, filtrado) por los tipos de información monitorizable (bits, paquetes) y por las funcionalidades de presentación de la información al usuario. A continuación se presentarán aspectos de cada una de estas partes.

En cuanto a las capacidades internas de los sistemas de monitorización será deseable que cuenten con las siguientes características:

- Precisión temporal: el sistema ha de ser capaz de proveer estadísticas del tráfico en varias escalas de tiempo.
- Capacidades de filtrado: el sistema de monitorización debe filtrar tráfico por lo que ha de soportar la decodificación de un gran número de protocolos.
- Adaptabilidad a nuevos protocolos y servicios: debido a la naturaleza cambiante de las topologías y tecnologías de red.
- Visión global de la red [25]: el sistema ha de proveer estadísticas de tráfico de todos los segmentos de red.
- Operación continua: es necesario llevar un control continuo del estado de la red para poseer toda la información referente a su estado de funcionamiento.
- Facilidad de uso: la herramienta ha de ser de fácil manejo para el operador de red, ayudando a éste en la interpretación de problemas o en la localización de fallos.
- Economía: cada vez las redes tienen más segmentos y puede darse la situación paradójica de que sea más costoso económicamente el sistema de monitorización que la propia red.

Un sistema de monitorización debe ser capaz de proveer diferentes estadísticas respecto a diferentes tipos de información monitorizable. Entre estos tipos de información, que llamaremos parámetros de monitorización, son habituales los siguientes:

- Bits o paquetes que verifican un filtro, acumulados en un intervalo de tiempo.
- Valores de campos de los paquetes que verifican un filtro. Por ejemplo, descubrir los valores de dirección IP origen que acceden a determinado puerto TCP destino.
- Los propios paquetes para su decodificación campo a campo.

Los parámetros de monitorización anteriores se podrán trasladar al usuario haciendo uso de diferentes funcionalidades de presentación de la información:

- Monitorización en tiempo real (*on-line*): ofrece información actualizada del parámetro monitorizado en el último intervalo y la evolución temporal del mismo en los últimos intervalos. Por ejemplo, puede mostrar los bits por segundo de carga total de la red, actualizando la información cada segundo. Además, permite aislar tráficos de distintos tipos y ver su comportamiento por separado, todo ello con realimentación inmediata al usuario.

- Capturas: en lugar de mandar al operador de red los parámetros monitorizados conforme se van generando, se pueden almacenar en disco duro local para su posterior envío conjunto [26]. Al solicitar una captura se definen los instantes de comienzo y finalización de la misma.

- Informes periódicos: son capturas que se repiten periódicamente, por ejemplo, todas las semanas [27].

- Alarmas: se pueden definir umbrales para ciertos parámetros del tráfico, que una vez rebasados activen una alarma que se envíe al operador de red para avisar de una situación anormal [28]. Por ejemplo, que salte una alarma cuando la utilización de la red sobrepase el 60% durante 10s.

1.3.2 Arquitectura de un sistema de monitorización

Un sistema de monitorización se compone típicamente de los bloques básicos presentados en la figura 1.1. Por un lado, se distinguen los dos elementos fundamentales, uno de recogida de información, la *sonda*, y otro de presentación de la información al gestor de red, la *consola*. Estos dos elementos podrán implementarse en la misma máquina, caso de un analizador de protocolos, o en máquinas diferentes, caso de los sistemas distribuidos de monitorización en los que nos encontramos una consola que centraliza la información de monitorización proveniente de sondas repartidas en distintos puntos de una red de área extensa.

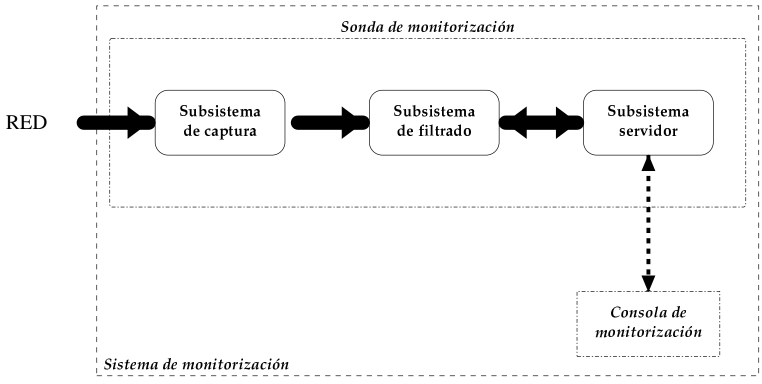


Figura 1.1: Arquitectura de un sistema de monitorización de redes de datos

La recogida de información por parte de la sonda se puede descomponer en varias tareas. Por una parte, la sonda conectada a la red de comunicaciones recoge todos los paquetes que

circulen por ella. Esta tarea corresponde al *subsistema de captura*. Éste recoge los paquetes de la red a través del dispositivo adecuado a esa red (por ejemplo una tarjeta Ethernet 10/100). Normalmente, el dispositivo hace de interfaz físico y enlace por lo que aunque se trate de un medio compartido sólo pasa a la máquina los paquetes a ella dirigidos (en Ethernet, aquellos paquetes que tengan por dirección MAC la de la tarjeta de red). Para poder capturar todos los paquetes de la red será necesario colocar el dispositivo en modo promiscuo (si el dispositivo lo soporta). De esta forma, se podrá tener copia de todos los paquetes que circulen por la red, junto con su tamaño y el instante en que se recibieron (timestamp). Estos paquetes se almacenan en un buffer para pasarlos al siguiente bloque y así minimizar el efecto de ráfagas que puedan aparecer en la red. Las características que tendrá que tener este subsistema de captura son las siguientes:

- No puede perder paquetes: la captura de todos los paquetes que circulen por la red es un factor fundamental que determina la fiabilidad de los resultados ofrecidos por el sistema de monitorización. La incorporación de buffers facilitará este punto.

- Precisión del timestamp: es otro factor importante que tendrá que ir de acuerdo a la velocidad de la red que se esté monitorizando, para tener suficiente granularidad en la medida.

- Detección de errores de red: parámetros como colisiones detectadas en la red, errores de CRC, etc. han de ser proporcionados por el dispositivo de red y transferidos por el subsistema de captura al subsistema de filtrado. Normalmente los dispositivos de red no diseñados específicamente para tareas de monitorización no proveen esta información.

El *subsistema de filtrado* se encarga de llevar cuenta de los parámetros de monitorización solicitados por el subsistema servidor. Para ello, un parámetro de monitorización se asocia con un filtro, por ejemplo, para contabilizar los bits por segundo transmitidos por una dirección IP origen. En tal caso será necesario comprobar dos condiciones: (i) que el paquete leído es IP y (ii) que la dirección IP origen coincide con la solicitada. Sólo en caso de que ambos se verifiquen se podrá actualizar el contador que lleve cuenta de los bits (incrementando este contador con el tamaño del paquete en bits). Este proceso de filtrado de paquetes y actualización de contadores lo realiza el subsistema de filtrado. Por un lado, recibe los paquetes a procesar desde el subsistema de captura. Por otro lado, desde el subsistema servidor recibe petición de qué parámetros monitorizar y de qué parámetros dejar de monitorizar.

Este subsistema de filtrado no lleva cuenta del parámetro temporal, se limita a incrementar los contadores según los parámetros definidos. La potencia del sistema de monitorización dependerá en gran medida de la optimización de este elemento: es el subsistema con más carga computacional porque tendrá que ser capaz de llevar miles de filtros simultáneos en la mayoría de las ocasiones. Nuestro objetivo es analizar los mecanismos de filtrado existentes y proponer uno que mejore sus prestaciones e implementarlo.

Sin embargo, los sistemas de monitorización actuales simplifican la labor de este subsistema de filtrado, limitándolo a filtrar paquetes y pasar al subsistema servidor aquellos que superen los filtros configurados. Así queda en manos del subsistema servidor el recibir un flujo (*stream*) de paquetes por cada filtro e incrementar los contadores del parámetro correspondiente. Esta

arquitectura puede ser válida para un número pequeño de filtros, pero no para un sistema de monitorización de alta capacidad. Incluso podrá ocurrir que un paquete verifique varios filtros y se tenga que copiar a varios flujos (colas de paquetes) para luego leerse en el subsistema servidor, lo cual es altamente costoso cuando lo que interesa normalmente en el parámetro son contadores de bits o paquetes, y no más información del paquete.

Finalmente dentro de la sonda nos encontramos con el *subsistema servidor* que es el que interacciona con el subsistema de filtrado para solicitar cierto parámetro. Este subsistema sí tendrá en cuenta la referencia temporal para poder ofrecer estadísticas del tipo parámetro durante cierto intervalo de tiempo. Para ello muestreará el valor de los contadores del subsistema de filtrado según el intervalo temporal predefinido para cada parámetro. En los casos en los que la sonda de monitorización se encuentre separada de la consola, el subsistema servidor es el que traslada al subsistema de filtrado las peticiones procedentes de la consola y al revés, traslada a la consola el valor de los parámetros de monitorización. Además este subsistema servidor será el encargado de ofrecer las funcionalidades de monitorización en tiempo real, capturas, informes periódicos y alarmas.

El otro elemento fundamental, la consola, será la encargada de ofrecer el interfaz de usuario que facilite su interacción con el sistema de monitorización. La consola tendrá que tener la base de datos de protocolos que permita mapear los filtros de alto nivel (definición de los campos de protocolos, su significado y posibles valores) en filtros adaptados a lo que entiende la consola (offsets, secuencias de bits, etc.) a través de un interfaz gráfico que permita una definición sencilla de los filtros y parámetros de monitorización. Esta base de datos integrada con un interfaz gráfico amigable facilitará la labor al usuario, presentándole las estadísticas de red en formato de tabla, histograma, gráfica de barras y matriz de tráfico. Todo el procesado se realiza de esta forma en las sondas, dejando en la consola el conocimiento de las pilas de protocolos a través de la base de datos de protocolos.

1.3.3 Paradigmas de monitorización

En este apartado vamos a describir los tres tipos de herramientas de monitorización más usados: analizadores de protocolos, herramientas basadas en SNMP y la monitorización distribuida.

1.3.3.1 Analizadores de protocolos

Un analizador de protocolos, también llamado *Sniffer*©[1], es un elemento hardware o software que se conecta a un segmento de red para realizar medidas de parámetros de tráfico. Un analizador de protocolos es, por tanto, un elemento autónomo que integra las funcionalidades de sonda y consola. Proveen muy buena precisión porque son equipos dedicados, y proporcionan variedad de parámetros como factor de utilización del segmento, colisiones, porcentaje de protocolos y servicios, hasta filtrados específicos por máquina. Sin embargo, la flexibilidad

[1] Sniffer es una palabra registrada por Network Associates en referencia a "Sniffer Network Analyzer". Sin embargo, esta misma palabra se utiliza de manera habitual para nombrar analizadores de protocolos o analizadores de red de otras marcas comerciales.

en la definición de filtrados nuevos no predefinidos en el equipo es baja. El objetivo para el que fueron diseñados fue como herramienta reactiva para detección y solución puntual de problemas en la red. Por tanto, tienen las siguientes limitaciones:

- La medida realizada sólo hace referencia al segmento monitorizado, no da idea del estado global de la red.
- Suelen estar limitados por la cantidad de memoria RAM, que suele ser pequeña (en torno a 8MB), por lo que no permiten realizar grandes capturas de trazas de paquetes. Trabajan con memoria y no con disco duro para acelerar el procesado.
- Se trata de una herramienta reactiva. Cuando se detecta un problema en la red el operador coloca el analizador de protocolos para intentar localizar su origen. No realiza un control continuado del estado de la red y, por tanto, su utilidad es restringida.

Existen multitud de analizadores comerciales entre los que destacan [29]: Sniffer de Network General, Domino LAN de Wandel & Goltermann y HP LAN Advisor de Hewlett Packard.

Un caso especial son los sniffers de alta velocidad tipo OC3MON o CoralReef. Se trata de herramientas muy especializadas que ofrecen parámetros de monitorización muy poco flexibles (no se puede definir el filtrado correspondiente al parámetro deseado, sólo limitarse a los existentes en la herramienta) y que tienen restricciones como la ausencia de monitorización en tiempo real [18, 19].

1.3.3.2 Plataformas SNMP

Para monitorizar correctamente una red corporativa o de un proveedor de servicios, caracterizada por la gran variedad de segmentos y tecnologías de red, se hace necesario poseer una consola central de gestión que permita recoger información del estado individual de cada segmento. A partir de la información recogida se tendrá conocimiento del estado global de toda la red. El protocolo SNMP (Simple Network Management Protocol) [30] permite que una consola de gestión (gestor) sondee los agentes de gestión normalmente situados en los propios elementos activos de la red como concentradores, conmutadores, puentes o routers, y que también reciba alarmas de estos agentes. Los parámetros que soporta son los determinados por la MIB (Management Information Base)[31] entre los que se encuentran parámetros de medida de tráfico además de otros de gestión de los sistemas. Los agentes se encargan de actualizar las entradas de la MIB realizando el procesado correspondiente dentro del elemento activo de red.

Las desventajas de este paradigma derivan de que el sistema se comporta como un sistema centralizado que utiliza un protocolo no fiable para transferencia de información con los agentes. Así, para llevar parámetros con dependencia temporal como la utilización de la red, la consola de gestión realiza un sondeo (*polling*) sobre los agentes a intervalos constantes de tiempo. El valor así obtenido sólo puede ser una estimación porque:

- El uso del sondeo implica que existe una incertidumbre en los datos aportados por el agente, porque no se puede determinar con exactitud el intervalo de tiempo real

transcurrido debido a la varianza en los retardos que introduce la red desde que se solicita un parámetro hasta que se recibe su valor.

- Como SNMP se soporta sobre UDP se pueden perder mensajes, especialmente en situaciones de alta carga, que es justamente cuando se hace más necesaria esta información. Las sucesivas retransmisiones que tienen lugar entonces, incrementan la variabilidad en el tiempo de llegada entre mensajes.

Por tanto SNMP no está diseñado específicamente para sistemas de monitorización sino que sólo es útil para acceso a información genérica de gestión de los equipos de una red.

Existen diversas plataformas de gestión comerciales (NMPs, Network Management Platforms) como HP OpenView [32], IBM Netview/AIX [33], Sun Solstice [34], Cabletron Spectrum [35] y Castle Rock [36], que proporcionan un control global de todas los equipos de una red de área extensa. Una herramienta interesante de libre distribución es Tkined/Scotty [37], útil como primera aproximación a los sistemas de gestión.

1.3.3.3 Monitorización distribuida

Los sistemas de monitorización distribuida tienen como objetivo principal evitar los problemas, carencias y desventajas de los sistemas anteriores: monitorización localizada y falta de memoria en analizadores de protocolos, imprecisión en la medida y pérdida de datos en caso de fallo del gestor en plataformas SNMP. Los sistemas de monitorización distribuida solventan estas dificultades mediante el uso de sondas de monitorización, las cuales tienen alta capacidad de procesamiento y almacenamiento. La información se reune en la consola que será un mero interfaz para el acceso sencillo a la información por parte de los usuarios.

Las sondas se distribuyen por todos los segmentos a monitorizar y la consola sirve de punto de acceso centralizado a la información procesada por cada sonda. En la figura 1.2 se puede ver una disposición típica de las sondas para una red determinada.

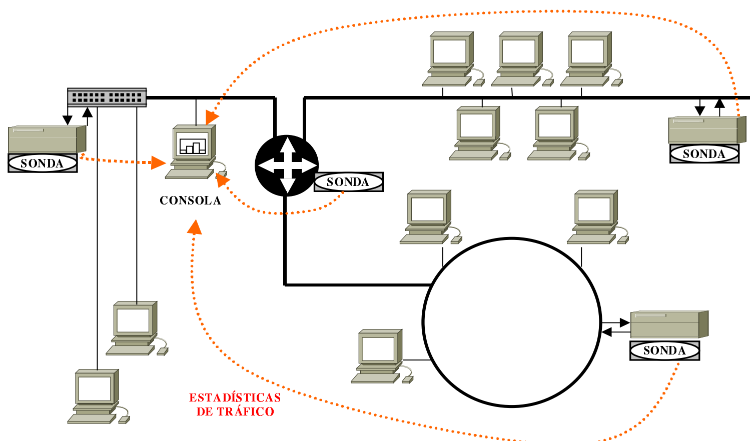


Figura 1.2: Escenario de monitorización de red usando sondas distribuidas

Las sondas son elementos hardware o software que miden el tráfico correspondiente a un determinado segmento de red. Puede tratarse de elementos dedicados exclusivamente a la monitorización o residir en equipos activos de la red que a su vez realizan sus tareas de conmutación, enrutado, etc. En cualquier caso, las sondas proveen estadísticas de tráfico de alta precisión sin necesidad de sondeo periódico por parte de la consola.

La consola proporciona el interfaz gráfico de usuario a través del cual el gestor de la red puede acceder a la información aportada por las sondas. Esta información puede ir desde gráficos de monitorización en tiempo real de determinado parámetro de tráfico hasta la recepción de capturas, pasando por la recepción de alarmas ante problemas detectados en la red. Desde esta consola además se tendrá acceso a la configuración de las sondas.

A la hora de determinar el conjunto de parámetros que se ha de ser capaz de monitorizar se puede seguir el estándar RMON. El estándar Remote MONitoring (RMONv2) [38] define una serie de conjuntos de objetos de información que han de contemplar las sondas de monitorización. Pero RMON no determina los algoritmos a utilizar en el filtrado y análisis de tráfico en redes de datos.

Respecto a las implementaciones comerciales, cada vez más los analizadores de protocolos o las plataformas de gestión están tendiendo a ofrecer posibilidades de monitorización de tráfico distribuida. Algunos ejemplos son los sistemas Distributed Sniffer de Network General [39], NetScout de Acterna [40], Tivoli Distributed Monitoring [41] y Distributed Protocol Inspector de Fluke [42].

1.3.3.4 Comparativa

En la tabla 1.1 se muestran las características principales de los 3 tipos de sistemas de monitorización comentados. En ella se observa como el sistema de monitorización distribuida es el que ofrece mejores prestaciones. Por una parte ofrece medidas on-line (en tiempo real) de cualquier parámetro de red que hace que conozcamos en todo momento el estado de la red. La monitorización distribuida también ofrece precisión temporal y fuertes capacidades de filtrado debido a la alta capacidad de procesado disponible en las sondas. También es de destacar la flexibilidad para adaptarse a nuevos protocolos y servicios, con sólo actualizar la base de datos de protocolos de la consola. Finalmente, la monitorización distribuida ofrece cobertura global, es decir, permite hacer seguimiento de diferentes segmentos de la red de forma simultánea, y también ofrece control continuo para un seguimiento de la red en todo momento.

En lo que sigue, cuando hablemos de sistema de monitorización se referirá a sistema de monitorización distribuido, que ofrece las mejores características y sobre el cual tendrá cabida nuestra propuesta de algoritmo de filtrado.

1.4 Problemática del filtrado de paquetes

El núcleo central de un sistema de monitorización es el filtrado de paquetes, ya que para obtener estadísticas de tráfico de la red es necesario decodificar y filtrar todos los paquetes, por lo menos hasta los niveles de protocolos en que estemos interesados. De esta forma la eficiencia

	Analizadores de Protocolos	*Plataformas SNMP*	*Monitorización distribuida*
Medidas On-line	Si, dispositivos independientes	Si, pero sólo por *polling*	Si
Precisión temporal	Si	Imprecisión debido al *polling*	Si
Capacidades de filtrado	Si	No	Si
Adaptabilidad a nuevos protocolos y servicios	No, medidas preconfiguradas	No, limitado a parámetros de la MIB	Si, dependiendo de la técnica de filtrado
Cobertura global	No, medidas limitadas a un único segmento	Si	Si
Control continuo	No, limitado por memoria RAM	Si	Si

Tabla 1.1: Características de sistemas de monitorización de red

del sistema dependerá de la optimización de este filtrado. Por tanto, la problemática de un sistema de monitorización se reparte entre el subsistema de captura y, en mayor grado, el subsistema de filtrado.

De entre los puntos problemáticos a tener en cuenta para el subsistema de captura destacan los siguientes [43]:

- La precisión del *timestamp* (marca temporal con la que se ha recibido el paquete) depende del sistema sobre el que funcione, aunque en los sistemas actuales es del orden de microsegundos para PC, estaciones de trabajo y mayoría del hardware dedicado. Esta resolución será insuficiente cuando se desee monitorizar redes de muy alta velocidad. Sin embargo, las mayores dificultades vendrán dadas por la sincronización entre los diferentes elementos del sistema de monitorización distribuido.

- Las pérdidas de paquetes se pueden producir a dos niveles. En el ámbito de la tarjeta de red, se pueden perder paquetes cuando no se hacen lecturas sobre la tarjeta a suficiente velocidad y ésta ha de sobreescribir los paquetes recibidos en su buffer, normalmente de tamaño pequeño (del orden de 32-64 Kbytes en una tarjeta Ethernet). El otro ámbito en que se puede producir pérdida de paquetes es en el buffer intermedio entre el subsistema capturador de paquetes y el subsistema de filtrado, cuando el proceso de filtrado y análisis no es lo suficientemente rápido.

- La generación de paquetes fantasma se debe a fallos de la implementación del soporte de red. Por ejemplo, se ha detectado como en sistemas Linux de kernel 2.2.12 se capturan paquetes de tamaño menor al mínimo permitido en una Ethernet (64 bytes) [44]. En

realidad, estos paquetes de tamaño pequeño no existen sobre la red, porque son los propios paquetes que envía la sonda y que ella misma recibe antes de ser completados a 64 bytes (*padding*) en la pila de protocolos Ethernet.

En lo que se refiere a la problemática del subsistema de filtrado, existen multitud de factores:

- A diferencia de los interfaces y protocolos de monitorización, los algoritmos de filtrado de paquetes orientados a la monitorización no se encuentran estandarizados. *Los estándares como RMONv2 especifican los parámetros que el sistema de monitorización ha de proveer pero no el cómo obtener esos parámetros.* Por tanto, en la práctica el rendimiento de unos u otros equipos comerciales varía sobremanera, dependiendo de la implementación realizada. Estudios recientes de eficiencia de sondas RMON comerciales muestran pérdidas apreciables de paquetes [45][46, capítulo 10.10] con lo que se obtienen medidas falsas de tráfico. Estos problemas se deben al diseño ineficiente de las sondas de tráfico, que son incapaces de soportar miles de filtros simultáneos con sus parámetros en situaciones de alta carga. Más aún, el protocolo de comunicación con las sondas RMON es SNMP y por tanto aparecen los problemas ya comentados antes para SNMP. De esta forma un algoritmo eficiente de cálculo de parámetros será fundamental en el rendimiento de un sistema de monitorización. En la actualidad no existen propuestas estandarizadas de este tipo de algoritmos.

- Los parámetros de monitorización se actualizarán a partir de la información obtenida de los paquetes que se transmiten a través del segmento de red que se monitoriza. Debido a que se analiza cada uno de los paquetes para saber qué filtros verifica y actualizar los parámetros asociados, la capacidad del sistema vendrá dada por los paquetes que se pueden filtrar por unidad de tiempo para cierto número de filtros o, de igual forma, por el número de parámetros que se es capaz de actualizar con una tasa constante de paquetes. Por tanto, plantear un algoritmo de filtrado que maximice estos valores será objetivo primordial.

- Un aspecto importante del sistema de filtrado ha de ser la flexibilidad para soportar nuevos protocolos y poder definir cualquier tipo de comprobación sobre secuencias de bits de una cabecera de protocolo dada. Por tanto, el subsistema de filtrado tendrá que ser flexible a la hora de definir filtros, de seleccionar los parámetros a monitorizar y en la forma de muestrear la información recogida por los parámetros. Debido a la continua implementación de nuevos protocolos y servicios, tradicionalmente los sistemas de filtrado para monitorización de red han sido por software. Sin embargo, para otro tipo de equipos, como por ejemplo elementos activos de red que tengan que conmutar a cierto nivel de protocolo, esta labor se realiza por hardware, por necesidades de velocidad, y por tanto el sistema de filtrado se diseña específicamente para soportar ciertos protocolos con lo que se pierde esta flexibilidad.

- En una implementación sobre una plataforma hardware de propósito general, la forma más eficiente de realizar el filtrado es a nivel de kernel del sistema operativo, de forma que no haga falta copiar los paquetes a nivel de usuario debido al coste en copias de memoria

que esto supone. Sin embargo, en tal caso, el desarrollo y depuración del algoritmo de filtrado se complica en gran medida.

- Conforme se desee soportar redes de mayor velocidad, las plataformas de procesamiento genéricas tendrán que ser sustituidas por implementaciones hardware, por ejemplo, una tarjeta de red que incluya los subsistemas de captura y filtrado. Por tanto, será necesario reducir al máximo los requisitos de memoria y operaciones complejas en la parte de filtrado.
- Aunque los niveles de protocolos sean independientes entre sí, existen campos que los interrelacionan para saber qué protocolo se encapsula por encima de otro. También, para conocer el protocolo de bajo nivel será preciso estudiar la propia cabecera (por ejemplo, distinción de los 4 tipos de paquetes Ethernet). Por tanto, será necesario añadir funcionalidades especiales adaptadas a los protocolos procesados, que permitan un filtrado correcto.
- Otro aspecto problemático en el filtrado es la fragmentación de paquetes. En caso de fragmentación sólo el primero de los paquetes resultantes contiene la cabecera de alto nivel (por ejemplo, TCP o UDP) que será muchas veces necesaria para realizar el filtrado sobre todos los fragmentos. Por tanto, será necesario que los fragmentos se reensamblen en el subsistema de filtrado cuando se desee testear campos de niveles superiores.

Habrá que tener en cuenta todos estos aspectos a la hora de diseñar un algoritmo de filtrado para sistemas de monitorización. Además, habrá otros aspectos como la reutilización de bloques de filtrado que se aprovechan de las características de los protocolos y permitirán aumentar la eficiencia del proceso. Cuando existen multitud de filtros definidos en el sistema resultará que muchos de los bloques de filtrado serán comunes entre los filtros. Se tratará entonces de aprovechar esta coincidencia para realizar el menor número de operaciones de forma que la comprobación de un bloque de filtrado sea suficiente para todos los filtros que lo tienen en común. Esto será lo que llamaremos reutilización de bloques de filtrado. Por ejemplo, será típico filtrar campos por encima de IP (direcciones, puertos, etc.) y todos estos filtros compartirán un primer bloque de filtrado que es la comprobación de que se trata de un paquete IP. Mediante la reutilización, comprobar que se trata de un paquete IP una sola vez será suficiente para avanzar en la comprobación de todos los filtros que tienen en común este bloque de filtrado. Así, se podrá llegar a soportar un mayor número de filtros simultáneos.

1.5 Arquitecturas de filtrado

A continuación se presentan las alternativas existentes en la actualidad en el filtrado de paquetes y por tanto con posible aplicación a sistemas de monitorización, en concreto a la parte del subsistema de filtrado. Por una parte se encuentran los *packet filter* que permiten filtrado de paquetes a nivel del kernel del sistema operativo para poderse quedar con un flujo de paquetes que verifica determinadas reglas. Por otra parte, se verán los algoritmos clásicos de clasificación de paquetes en routers de alta velocidad, en los cuales el filtrado se refiere a la parte que verifica a qué subred pertenece la dirección IP destino del paquete que llega. También

se citan las técnicas utilizadas en los sistemas operativos para el acceso a la información de red, útiles para demultiplexar la información recibida en una máquina con destino a diferentes servicios.

Prácticamente la totalidad del trabajo existente en la actualidad sobre los temas de filtrado y monitorización se centran en la parte de filtrado (packet filters, clasificación en routers de alta velocidad y soporte de red en sistemas operativos). Sin embargo, apenas se tiene constancia de trabajos que traten el tema de filtrado orientado a la monitorización. Sólo el sistema NNstat de monitorización de redes aporta un algoritmo de filtrado asociado a una estructura de parámetros de monitorización separada [20].

1.5.1 Packet filter

Fuera de lo que son equipos hardware específicamente diseñados para monitorización de red, el filtrado de protocolos en un PC o estación de trabajo de propósito general se puede realizar desde el *kernel* del sistema operativo o desde un proceso de usuario. En el primer caso el código es más difícil de desarrollar y mantener, y además la configuración de filtros se complica. En el segundo caso se produce una sobrecarga de procesado y una bajada del rendimiento, debido al incremento de cambios de contexto y llamadas al sistema que supone las copias de paquetes del kernel al nivel de usuario.

Una solución de equilibrio que se utiliza normalmente en sistemas UNIX es el *packet filter* [12]. Los packet filter permiten una programación de filtros desde el nivel de usuario con una parte del kernel que filtra paquetes según los criterios especificados al nivel de usuario. Así resulta el proceso eficiente porque se evita el coste computacional de copias de paquetes del kernel al nivel de usuario donde también se podría realizar el filtrado. Además, el packet filter aísla al kernel de los detalles de implementación de los protocolos. La interfaz que provee el packet filter otorga facilidades para transmisión y recepción de paquetes, e información y control de estado. Algunos ejemplos de packet filter son el CSPF, que fue la primera tentativa, y el BPF ya comentados.

El objetivo con el que surgió la idea de los packet filter fue el de implementación de protocolos desde el nivel de usuario haciendo uso de funcionalidades del kernel para obtener mayor eficiencia. De esta forma se obtienen todos los beneficios de la programación a nivel de usuario y a nivel de kernel. El packet filter forma parte del kernel del sistema operativo por lo que es capaz de demultiplexar paquetes reduciendo los cambios de contexto y las copias de paquetes, sin perder la posibilidad de definir los criterios de filtrado de forma dinámica por parte del usuario. Su utilidad queda demostrada por su extendido uso durante años.

Un packet filter se coloca por encima del controlador de dispositivo (*device driver*) como se muestra en la figura 1.3. Provee un interfaz de programación que consta de primitivas para transmisión de paquetes, recepción de paquetes y control e información de estado:

- Transmisión de paquetes: dado un buffer con un paquete que incluya el nivel de enlace, una llamada *write* devuelve el control una vez el paquete ha sido encolado para transmisión.
- Recepción de paquetes: se consigue con una llamada *read* y obtiene el paquete completo,

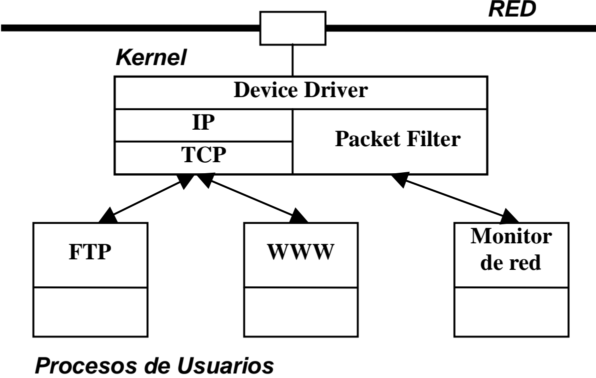


Figura 1.3: Relación del packet filter con otros elementos del sistema

incluida la cabecera de nivel de enlace. En una misma llamada al sistema se pueden recibir varios paquetes simultáneamente, lo que es útil si se quieren procesar los paquetes de una red cargada porque se reduce el número de llamadas al sistema. También se puede configurar un *timeout* para no quedarse bloqueado en el *read*.

- Control y Estado: se trata de un lenguaje simple que hace uso de constantes y referencias al paquete recibido para definir los filtros. Éstos se configuran mediante llamadas al sistema *ioctl* con un coste de proceso comparable al de recibir un paquete. Se pueden configurar también los timeouts de bloqueo de las lecturas, la señal que ha de ser lanzada con la recepción de un paquete y la longitud de la cola de entrada. También se puede solicitar que los paquetes vayan marcados con un timestamp y con un contador que indique el número de paquetes perdidos debido a la saturación del buffer de entrada.

Los filtros se compilan en tiempo de ejecución y el lenguaje de definición opera sobre una máquina virtual de estructura tipo pila o de registros (lenguaje imperativo), lo que da más flexibilidad que un lenguaje declarativo (o de predicados) formado por un array de pares {*offset del campo - tamaño*, *valor esperado*}.

A cada filtro se le asigna una prioridad según la cual se puede hacer que un paquete sea devuelto sólo por el filtro de mayor prioridad que verifica, o que se puedan dar copias del paquete en cada flujo de salida de cada filtro que verifica.

Algunas de las características principales comunes de las arquitecturas de filtrado basadas en packet filter son las siguientes:

- Desacopla la parte de red del kernel y el sistema operativo.

- Los sistemas de monitorización de red normalmente funcionan a nivel de usuario, por lo que es necesario copiar los paquetes entre los espacios de memoria de kernel y de usuario. El packet filter minimiza estas copias realizando el filtrado a nivel de kernel, y copia sólo los paquetes de salida de los packet filter.

- Frente a una implementación en el kernel con código nativo, los packet filter ofrecen peor eficiencia [12] (dos o tres veces peor para implementaciones antiguas como CSPF e

imperceptible para implementaciones más recientes) debido a la emulación del filtrado sobre una máquina virtual que trabaja también a nivel de kernel.

- Reduce el coste en cambios de contexto y en intercomunicación respecto de un proceso demultiplexor implementado a nivel de usuario.

- Cuanto antes se pueda realizar el filtrado en la pila software, mejor será la eficiencia del sistema porque se descartarán los paquetes no necesarios antes de realizar ninguna copia.

1.5.1.1 CSPF

Una de las primeras implementaciones de packet filter fue el CSPF (CMU/Standford Packet Filter) [12] que utiliza un árbol de expresiones booleanas, en el que los nodos representan operaciones booleanas y las hojas campos del paquete a comparar. Los filtros se implementan sobre una máquina virtual con modelo de pila por lo que su eficiencia en las máquinas actuales es reducida. Sus limitaciones son que no puede tratar cabeceras de tamaño variable y la máquina virtual trabaja con palabras de 16 bits.

1.5.1.2 BPF

El más conocido de los packet filter es el BPF (BSD Packet Filter) [14, 47] que realiza el filtrado sobre una máquina virtual de registros, más eficiente en las arquitecturas de ordenadores actuales, compuesta de dos registros (A y X), el paquete accesible como un array de bytes (P[]) y una memoria (M[]). El filtrado se basa en un grafo dirigido acíclico (CFG, *Control Flow Graph*) en el que cada nodo representa un campo del paquete a testear y según el resultado del test se atraviesa una rama o la otra. Sólo hay dos hojas que representan los valores verdadero/falso para todo el filtro. El CFG es local para cada filtro y no global al conjunto de filtros.

En los estudios mostrados en [14] se muestra una eficiencia mayor del BPF frente al CSPF. Sin embargo, sigue quedando por debajo de lo que sería una implementación nativa a nivel de kernel debido al coste de interpretación del código del packet filter que determina su flexibilidad. En la figura 1.4 se presenta un ejemplo de funcionamiento de BPF con tres filtros. Su filtro de la izquierda ($IP_1\, origen, IP_2\, destino, UDP$) tiene un grafo CFG que se muestra en la figura 1.5.

El diseño del BPF supone que la mayoría de aplicaciones rechazan más paquetes que los que acepta e incluye entre ellas las aplicaciones de monitorización. Es el caso de la aplicación *tcpdump* [15] que bajo un interfaz textual nos permite ver los paquetes de la red que cumplan determinado filtro. Si bien es verdad que aplicaciones de monitorización como tcpdump están orientadas a la captura de pocos paquetes o paquetes que cumplan una única serie de reglas, el problema genérico de monitorización requiere una solución más potente que permita no sólo limitarse a un pequeño número de paquetes y ofrezca posibilidades de una monitorización más global, permitiendo simultanear miles de filtros.

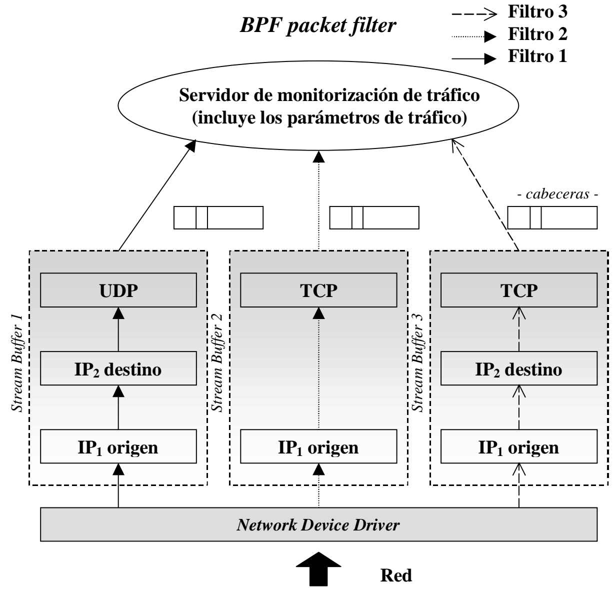


Figura 1.4: Ejemplo de funcionamiento de un packet filter como BPF

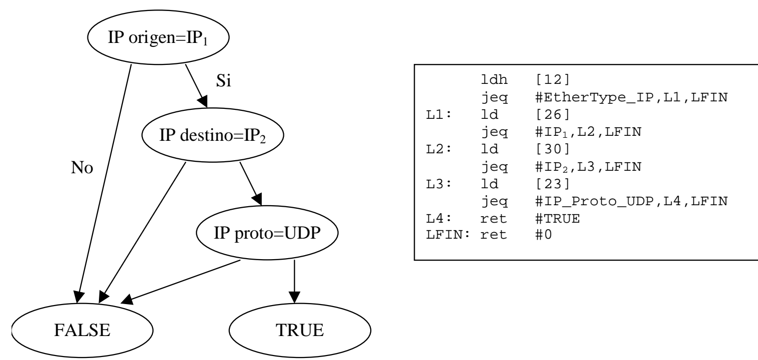


Figura 1.5: Grafo acíclico correspondiente al filtro de la izquierda de la figura 1.4

Otro aspecto importante en BPF es la independencia entre filtros. Cuando se establecen varios filtros simultáneos, no se reutilizan tareas comunes y para cada uno se crea un flujo de salida al que se copiará los paquetes que verifiquen el filtro correspondiente (figura 1.4). Por tanto, con la llegada de un paquete al controlador de dispositivo se testean secuencialmente todos los filtros definidos según una disciplina round-robin.

El lenguaje que utiliza BPF incorpora instrucciones de longitud 64 bits incluyendo los operandos. Se trata de un lenguaje estilo ensamblador (ver ejemplo de la figura 1.5) y las instrucciones de que dispone son de los siguientes tipos:

- Carga de datos al acumulador o al registro de índice, a partir de una posición fija o variable del paquete, de su longitud o de lo almacenado en memoria.
- Almacenar el acumulador o el registro de índice en memoria.
- Aritmético-lógicas, usando como operandos el registro de índice o una constante.
- Salto, cambiar el flujo del programa al comparar una constante o el registro de índice con el acumulador.
- Retorno, para finalizar e indicar la porción de paquete que ha de devolver el filtro en el flujo de salida o 0 si se descarta.
- Misceláneas, por ejemplo, transferencia entre registros.

Al ser alta la flexibilidad en la definición de filtros también es compleja, sobre todo si tenemos en cuenta las cabeceras de protocolos de tamaño variable y la fragmentación de paquetes. La librería *pcap* [16] incorpora un sistema de traslación de esos filtros: los filtros se definen con un lenguaje de más alto nivel y amigable que después se mapeará al requerido por el BPF. Además esta librería ofrece un interfaz común de acceso a la lectura de paquetes y definición de filtros para cualquier plataforma UNIX, utilizando en cada caso los recursos que disponga el kernel del sistema operativo o incluso dejando en manos de la librería la labor de filtrado si el kernel no provee esas facilidades.

1.5.1.3 MPF

El Match Packet Filter (MPF) [48] provee un mecanismo que permite combinar filtros similares en un único filtro y ha sido implementado en el sistema operativo Mach 3.0. Supera las prestaciones del BPF gracias a la mejora de la escalabilidad con el número de filtros y la incorporación de mecanismos de manejo de paquetes fragmentados.

En cuanto a la escalabilidad, el MPF incorpora un sistema que optimiza los filtrados. Normalmente, el filtrado de un protocolo consiste en dos partes: una identifica el protocolo (por ejemplo TCP) y la otra la sesión dentro de ese protocolo (por ejemplo, un puerto destino determinado). Por tanto, la primera parte será común para todas las sesiones dentro de ese protocolo y se podrá minimizar el número de operaciones a realizar si se hace uso de esta coincidencia. Para ello, MPF transforma los filtros de un mismo protocolo en un mismo

programa que luego se diferenciará en la distinción de cada sesión. Además, para distinguir cada sesión hace uso de tablas de hash para optimizar la búsqueda.

Aunque esta optimización es muy potente y sencilla de implementar, es muy restrictiva. Un filtro MPF sólo puede tener una única secuencia común al principio y ha de ser del tipo protocolo-sesión. Además, una vez identificada la sesión no se pueden seguir más instrucciones. Sería interesante poder tener varias secuencias comunes y en cualquier parte del filtro. Por tanto, la estructura de filtrado resultante es más global al compartir secuencias comunes protocolo-sesión, pero se mantiene la independencia para filtrados más complejos y, en todos los casos, la existencia de un flujo por filtro al que se copian los paquetes que superan cada filtro.

Para abordar el problema de la fragmentación, MPF incorpora información de estado en cada filtro que persiste durante la llegada de nuevos paquetes. Ante la llegada del primer fragmento se almacena la información de sesión junto con el identificador, de manera que a los siguientes fragmentos que lleguen con el mismo identificador se les pueda asociar la información de sesión correspondiente y de esta manera saber qué filtro verifican. En el caso de que se reciban los últimos fragmentos en los primeros lugares, se almacenan hasta recibir el primero que es el único que contiene la información de sesión. En todos los casos, los fragmentos y la entrada con la información de sesión se mantienen un tiempo finito tras el cual son eliminados.

De esta forma, MPF mejora el tiempo empleado en el filtrado a cambio de encarecer el coste de creación de filtros. Esto se debe a que ahora cuando se crea un filtro nuevo es necesario compararlo con los filtros existentes para intentar usar la parte común de filtrado de protocolo existente y añadir sólo el filtrado de sesión de forma optimizada. Otro aspecto importante es que la sobrecarga introducida por el soporte de fragmentación es despreciable al tener código nativo que realiza esta tarea [48].

1.5.1.4 PathFinder

El PathFinder [49] es una alternativa optimizada tanto para su aplicación software como para su implementación hardware. La especificación de los filtros se realiza con un lenguaje declarativo, es decir, se define el patrón que se ha de verificar. Básicamente el patrón está compuesto por dos líneas y cada línea por celdas. Cada celda consiste en el testeo de determinada secuencia de bits y es de la forma *(offset, longitud, máscara, valor)*. Con esta celda se testea que los bytes colocados a ese *offset*, tomando *longitud* bytes y aplicándoles la *máscara* coincidan con *valor*. De las dos líneas que componen cada patrón, la segunda es cargada sólo en el caso de que se verifique la primera. Esto se utiliza para la fragmentación, la primera línea la detecta y la segunda se encarga de guardar información de la misma para poder procesar el resto de fragmentos. Si los últimos fragmentos llegan antes que el primero se pospone el procesado de éstos hasta recibir el primero.

Para almacenar todos estos filtros hace uso de un grafo dirigido acíclico, donde cada nodo del grafo corresponde a una celda y existen nodos especiales denominados de cabecera que delimitan los patrones de diferentes protocolos. Estos nodos se unen por ramas que pueden indicar operaciones lógicas *AND* u *OR*. Incorpora una caché que optimiza la búsqueda de paquetes similares a los recibidos últimamente usando las segundas líneas de los patrones

cuando no hay fragmentación.

Como resultado, PathFinder mejora las prestaciones de MPF en los filtrados. Sin embargo, el coste de inserción/borrado es también grande debido a que se tiene que buscar qué parte del patrón y en qué medida coincide con uno ya existente, además del coste de las reservas, copias y liberaciones de memoria.

1.5.1.5 DPF

DPF (Dynamic Packet Filter) supera las prestaciones de todos los anteriores [50]. Consiste en un lenguaje declarativo que se optimiza de forma agresiva usando generación dinámica de código también llamada compilación dinámica. El lenguaje se estructura en sentencias de comparación del tipo *(offset: bits == valor)* y sentencias de desplazamiento de offset del tipo *(SHIFT(offset))*.

Al igual que el PathFinder se basa en una secuencia de comparaciones booleanas denominadas átomos y unidas por conjunciones *AND* u *OR*. Si todos los átomos son ciertos, el filtro acepta el paquete y éste es direccionado a la aplicación asociada.

La estructura de datos es un árbol especial denominado *Trie* (apéndice A.4). La optimización más importante viene al compilar los filtros en código ejecutable usando el sistema de generación dinámica de código VCODE [51] que permite su portabilidad y simplifica el mecanismo de filtrado al eliminar la necesidad de un evaluador de expresiones como los packet filter clásicos. Por tanto, aunque se optimiza el proceso de búsqueda, la inserción queda seriamente penalizada por esta necesidad de compilación además de por necesitar que los elementos se encuentren secuencialmente en memoria. Al realizar este código las constantes de los filtros se incluyen directamente en las instrucciones máquina siendo ésta una de las principales causas de mejora. Por ejemplo, cuando un campo se ha de comparar contra una serie de posibles valores. Por el uso de este proceso de compilación, se hace sencilla la comprobación de la alineación de las palabras y el chequeo de los límites de filtrado.

Como resultado, la eficiencia de una pila TCP/IP implementada con DPF es tan buena como la conseguida en una implementación nativa sobre el kernel.

1.5.1.6 BPF+

Una segunda versión de BPF aparecida recientemente llamada BPF+ [52] minimiza la redundancia de filtros mediante la incorporación de un optimizador. El BPF+ trata de conseguir una especificación de filtros a alto nivel (flexibilidad) con una buena eficiencia.

Las fases que sigue un filtro desde que se especifica hasta que se pone en funcionamiento son las siguientes:

- La entrada es un expresión, en lenguaje de predicados de alto nivel, del filtro deseado. Por ejemplo: “UDP”, “((src host 130.206.160.215) and (TCP port 80))”.
- Se convierte en un lenguaje imperativo con una representación en grafo haciendo uso de SSA (Static Single Assignment) [53].

- La representación en formato SSA es pasada al optimizador que realiza una optimización local (minimizar las instrucciones por sentencia) y otra global (minimizar el camino medio por el grafo). La salida es un *byte code* que entiende la máquina virtual que provee BPF+.

- El *byte code* se pasa del espacio de usuario al entorno de ejecución en el kernel para realizar el filtrado.

- Se chequea la seguridad del código pasado (alineamiento, límites de memoria, ciclos infinitos e instrucciones válidas).

- Finalmente un ensamblador JIT (Just In Time) traduce los *byte codes* a código nativo de la máquina y realiza una última optimización en función de la máquina. También puede ser que el BPF+ esté implementado en kernel y pueda interpretar directamente los *byte codes*, pero esta opción es más lenta.

BPF+ es capaz de reconocer, al igual que DPF, si dos filtros son iguales y reducir las operaciones comunes. A diferencia del DPF, que utiliza una estructura de plantillas, tiene mayor flexibilidad a la hora de realizar operaciones matemáticas sobre campos de la cabecera o comparar un campo de la cabecera frente a otro. La máquina virtual sobre la que se basa consiste en 32 registros y memoria, e incluye operaciones de carga en registro, almacenamiento en memoria, operaciones aritmético-lógicas, de salto y retorno. Permite especificar los campos del paquete que se quieren testear y conectar esos predicados con operaciones booleanas "AND", "OR" y "NOT".

Este packet filter aplica un proceso de optimización sobre el código de la máquina virtual para conseguir mayor eficiencia del sistema. Modelando el packet filter como una función de predicados booleanos, la optimización del filtrado se puede asimilar a la reducción de un árbol de decisión [54]. Así, se pueden distinguir cuatro tipos de optimización:

- Eliminación de predicados redundantes. BPF+ genera árboles de decisión con muchos predicados redundantes. En tiempo de compilación se ha de detectar esta redundancia y evitarla. Para ello se siguen los siguientes pasos:

  - Eliminación de información parcial. Elimina operaciones innecesarias que se realizan dentro de los nodos, normalmente cargas de campos del paquete u operaciones aritmético-lógicas repetidas. Para ello, asigna un identificador a cada una de las operaciones (una carga desde memoria, copia entre registros,...) y busca identificadores comunes dentro del mismo nodo. Se define una relación de nodo dominador que determina con qué orden se han de atravesar los nodos. Si en un nodo se repite una operación ya realizada en un nodo dominante, se limita a copiar el resultado de ésta desde el nodo dominante.

  - Propagación de predicados afirmativos. Análisis que consiste en la propagación de valores de determinados predicados a través del grafo. Al igual que en el caso anterior, si existen operaciones calculadas con anterioridad y se llega a un nodo con las mismas operaciones se puede eliminar ese nodo y hacer que la rama vaya directamente a alguno de sus nodos hijo.

  - Predicción estática de predicados. Usa la información de aseveraciones para identificar de manera estática bifurcaciones y desvíos siempre que sea posible.

- Optimización de predicados. Consiste en la optimización del código dentro de cada bloque, tendiendo a sustituir los registros sobre los que se realizan las operaciones por constantes, cuando su contenido es conocido a priori o procede de un cálculo anterior.

- Encapsulación de búsqueda en tablas. Para realizar búsquedas en tablas de un campo del paquete frente a diferentes valores a filtrar, se puede hacer uso de una búsqueda lineal, binaria o por hash según el número de valores que formen la tabla y su distribución.

- Reserva y asignación de recursos. Los 32 registros, de los que hace uso la máquina virtual de BPF+, necesitan ser mapeados a los registros reales de la máquina de manera dinámica. No es problemático porque la vida de uso de los registros suele ser de unas pocas instrucciones.

BPF+ conjuga el uso de un lenguaje de alto nivel en la definición de los filtros con una fuerte optimización que lo convierte en una implementación eficiente, incluso mejor que la conseguida con otros packet filter que usan un lenguaje más limitado de definición de filtros.

1.5.1.7 Conclusiones de los packet filter

Los packet filter revisados, aunque proporcionan una herramienta útil para el análisis parcial del tráfico que circula por la red, no son de aplicación en el caso de redes de medio y gran tamaño. El caso de análisis de redes de forma global supone miles de filtros simultáneos y procesado de todos los paquetes de la red. En estos entornos, con enlaces de alta velocidad, un packet filter produce una carga excesiva por los posibles filtros duplicados o por el coste computacional que supone trasladar la información de cada paquete al proceso de usuario que lleve las estadísticas de monitorización.

La figura 1.4 muestra un ejemplo de filtrado mediante packet filter, en concreto BPF, aplicado a la monitorización de red. Aparece en la figura un proceso servidor de monitorización de tráfico que actualiza los parámetros a monitorizar según recibe paquetes por los flujos asociados a cada filtro. La arquitectura opera como sigue:

1. En la mayoría de los packet filters los filtros se definen de manera independiente por lo que no se reutilizan bloques de filtrado comunes. En otros más avanzados como BPF+ sí se realiza una optimización global entre los diferentes filtros, pero en todos los casos la optimización se aplica sobre el lenguaje de la máquina virtual implementada haciendo uso de una estructura en grafo intermedia.

2. Se crean diferentes colas para pasar los paquetes que cumplen cada filtro al servidor de monitorización. Además, para el caso del BPF se crea un dispositivo (*device*) $/dev/bpfxxx$ por cada filtro, limitando este número a 256 que es el máximo *minor number* para un dispositivo en un sistema de ficheros UNIX. En otros casos se crea un socket por cada filtro cuyo número (como descriptor de fichero que es) también está limitado en el sistema operativo.

3. Cada paquete se testea secuencialmente contra todos los filtros definidos.

4. Si un paquete verifica determinado filtro se hace una copia del mismo a través de la cola al servidor de monitorización que es un proceso de usuario.

5. Si un paquete verifica más de un filtro, se copia a todas las colas asociadas a esos filtros verificados.

Por tanto, los problemas planteados por esta arquitectura se pueden interpretar de la siguiente manera. En primer lugar, que la verificación de un filtro suponga una copia hace que en sistemas de monitorización donde existen gran cantidad de filtros definidos y donde es probable que un paquete verifique varios filtros, las múltiples copias sean un factor limitante en la eficiencia del sistema. En segundo lugar, la independencia entre filtros hace que no se reutilicen bloques de subfiltrado comunes. Así por ejemplo, en la figura 1.4 el campo de dirección IP_1 origen se testea tres veces, una por filtro. Finalmente, se necesita un dispositivo o descriptor de fichero por cada filtro definido, lo que junto a la reserva del buffer para la cola por cada filtro supone un coste elevado asociado a la inserción de un nuevo filtro.

De esta forma, se puede concluir que los packet filter son adecuados para proveer un filtrado muy concreto a aplicaciones que realicen un análisis detallado sobre un flujo. Sin embargo, proveen una solución ineficiente para aplicaciones de monitorización de red que requieran un alto número de filtros simultáneos.

Un aspecto a destacar en los packet filters es la importancia del lenguaje de definición de filtros. Según el tipo de lenguaje será más sencillo un tipo de optimización u otro. Los lenguajes imperativos son aquellos en los que se especifica la secuencia de pasos para producir el resultado deseado, mientras que los lenguajes declarativos describen relaciones entre variables en términos de funciones o reglas de inferencia, y al ejecutarse se aplica algún tipo de algoritmo fijo a esas relaciones para obtener el resultado. Un lenguaje de definición de filtros de tipo declarativo hace los filtrados más concisos, más fáciles de escribir y sobre todo permite reusar los bloques de subfiltrado más fácilmente: un bloque de subfiltrado se podrá reutilizar en otro filtro si la definición declarativa de ambos coincide [50, 55]. Sin embargo, también es menos potente al tener que limitarse a una estructura fija de definición de filtros.

En la tabla 1.2 se muestra una comparativa de las características principales de los sistemas de packet filter revisados. La columna de flujo por filtro indica si los paquetes se copian a una cola de salida por cada filtro definido. La columna de optimización global indica si se realiza optimización de filtros teniendo en cuenta otros ya existentes, es decir, reaprovechando partes del filtrado. Respecto al lenguaje, determina las peculiaridades del sistema: si es imperativo tendrá más flexibilidad mientras si es declarativo la optimización de código será más sencilla. Finalmente la última columna indica el mecanismo de filtrado, si es por máquina virtual y si además se hacen uso de estructuras de grafo o mecanismos de generación dinámica de código o ensamblador JIT.

	Flujo por filtro	*Optimización global*	*Lenguaje*	*Mecanismo de filtrado*
CSPF	Si	No	Imperativo	Máquina virtual pila
BPF	Si	No	Imperativo	Máquina virtual registros
MPF	Si	Sólo a un nivel	Imperativo	Máquina virtual registros
PathFinder	Si	No	Declarativo	Grafo Dirigido Acíclico
DPF	Si	Si, sobre lenguaje	Declarativo	Generación Dinámica Código
BPF+	Si	Si, sobre lenguaje	Imperativo	Ensamblador JIT

Tabla 1.2: Comparativa de las características de los packet filters

1.5.2 Clasificación de paquetes en routers de alta velocidad

Ante el rápido crecimiento de Internet un problema acuciante en la actualidad es la sobrecarga de los grandes routers de interconexión. Estos routers han de ser capaces de procesar millones de paquetes por segundo y en ellos el cuello de botella principal es la búsqueda en las tablas de rutas. Para dar una idea de los órdenes de magnitud de capacidad de proceso de estos routers, han de ser capaces de soportar alrededor de 2 millones de paquetes por segundo para una velocidad de 1 Gbps [56]. Hasta ahora estas tablas se almacenaban en memorias caché muy rápidas, pero esta técnica ya no es apropiada para grandes redes donde las tablas alcanzan centenares de entradas y cambian bastante rápido. Por tanto, se aplican algoritmos de búsqueda mejorados y de pequeño tamaño para que tengan cabida en la caché primaria del procesador y evitar así accesos innecesarios a memoria.

La búsqueda de la ruta óptima en estos routers se basa en la coincidencia del prefijo más largo (*longest-address prefix match* [57]) en el que para una dirección de una máquina enrutada hacia cierto interfaz tiene que encontrar la secuencia apropiada en unas tablas internas del router, denominadas tablas de encaminamiento o de enrutamiento. Las tablas de encaminamiento contienen secuencias de bits con el prefijo de direcciones que han de cumplir los paquetes que se han de redireccionar por el interfaz del router que también figura en esa tabla asociado a cada prefijo. Esta secuencia coincidente puede ser la propia dirección o el prefijo más largo que indique la subred destino. De esta forma, cada interfaz del router dará acceso a paquetes con direcciones destino en diferentes subredes.

Proponer un nuevo algoritmo exige que su eficiencia en términos de memoria o complejidad del algoritmo sea mejor que comparado a los algoritmos clásicos [56]. Estos estudios de eficiencia varían mucho dependiendo de la tecnología hardware y del escenario de medida. Por tanto, los datos publicados sobre búsquedas por segundo son difícilmente verificables. Por ello se deberá tener en cuenta los siguientes aspectos:

- Considerar el escenario del peor caso para los algoritmos, tomando el número de accesos por búsqueda como la métrica de comparación. Este número de accesos es función del número de entradas y de la longitud de las direcciones. En concreto se puede estudiar la optimización de los algoritmos para las direcciones IPv4.
- Suponer que se verifica en cierta medida el principio de localidad de las direcciones IP

[58, 59] que muestra que gran parte de los destinos de los paquetes corresponden a un pequeño rango de direcciones, sobre todo en periodos de tiempo pequeño.

- Debido a que las tablas de enrutamiento están en continua actualización, es importante determinar el coste de CPU de las inserciones y borrados de rutas en las tablas de enrutamiento.

- El tamaño de la tabla de enrutamiento es también otro factor importante a considerar, sobre todo si su tamaño es tal que quepa en la propia caché del procesador o se desea tener varias copias para acelerar el proceso (paralelismo).

- El manejo de condiciones de desbordamiento u otro tipo de errores es importante. Por ejemplo, el tamaño de las tablas de enrutamiento puede crecer de manera indefinida por el crecimiento de Internet o fallos de configuración de los routers.

Para poder comparar los diferentes sistemas, el proyecto IPMA (Internet Performance Measurement and Analysis [60]) provee trazas con los contenidos de las tablas de enrutamiento de diversos routers de Internet. Por ejemplo, se pueden encontrar datos sobre routers de interconexión de grandes redes con hasta 45.000 entradas.

1.5.2.1 Arquitectura de routers de alta velocidad

Tradicionalmente los routers se han implementado por software corriendo en uno o varios procesadores de propósito general [61]. En ellos, los paquetes que llegan a cualquiera de los Interfaces de Red (IR) se mandan a la Unidad Central de Encaminamiento (UCE) para el cálculo de la ruta y encaminarlo al interfaz de salida adecuado. Se trata de routers centralizados como el mostrado en la figura 1.6, y precisamente esta centralización en la unidad de proceso se convierte en el cuello de botella del sistema (la búsqueda de la ruta es la tarea que más recursos consume). Cada IR tiene una pequeña caché que mejora las prestaciones y cuando un paquete no se encuentra en ella su cabecera se direcciona a la UCE a través de una malla de conmutación. La UCE busca la ruta para esa cabecera y devuelve al IR origen la cabecera con el IR de salida. Ahora el IR de entrada puede redireccionar el paquete al IR de salida. El Procesador de Enrutamiento (PE) es el encargado de comunicarse con los otros routers mediante los protocolos de enrutamiento para intercambiar información sobre las rutas óptimas y poder actualizar así las tablas de rutas.

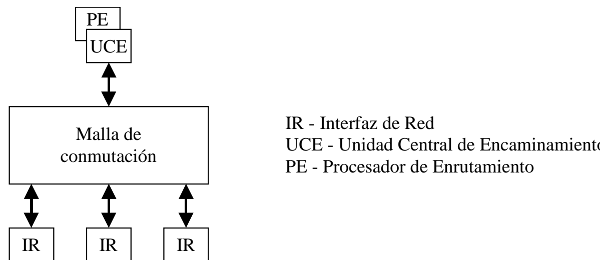


Figura 1.6: Arquitectura de router centralizado

Como mejora de la propuesta anterior existen en la actualidad arquitecturas distribuidas de routers como la presentada en la figura 1.7. En ella existen dos tipos de unidades de encaminamiento, una central (UCE) y varias locales (ULE) asociadas a cada IR. El ULE es capaz de procesar gran parte de los paquetes que provienen del IR evitando el tener que enviar la cabecera al UCE por la malla de conmutación. El UCE periódicamente actualiza las tablas de los ULE, y sólo se redireccionarán peticiones al UCE en el caso de que los ULE no tengan tablas suficientemente actualizadas o sea un caso especial.

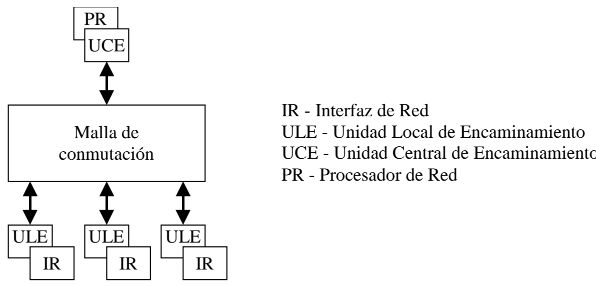


Figura 1.7: Arquitectura de router distribuido

Finalmente otro modelo de arquitectura para un router es la paralela, en la que las Unidades de Encaminamiento (UE) son varias y están compartidas por todos los IR. Al trabajar los UE en paralelo se consiguen mejores resultados si se minimiza el número de cabeceras de paquetes que han de atravesar la malla de conmutación usando cachés en los propios IR a modo de conmutación a nivel IP. Sin embargo, entonces aparece el problema de la sincronización y consistencia en las tablas de rutas distribuidas en los elementos del router, que puede originar pérdidas y desorden de paquetes.

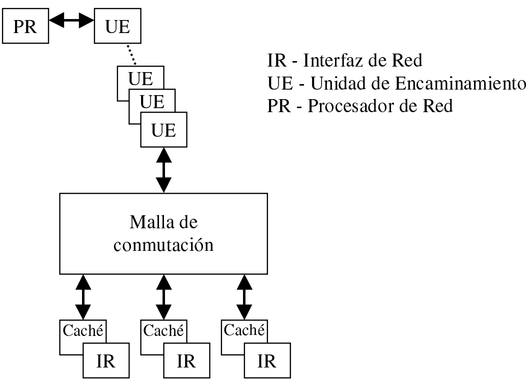


Figura 1.8: Arquitectura de router paralelo

Con todo ello, los algoritmos de búsqueda en tablas de enrutamiento son parte fundamental en el diseño de un router de alta velocidad. Existen diferentes algoritmos propuestos en la literatura para la búsqueda del prefijo más largo que verifica la dirección IP destino de un paquete, y se presentan a continuación. Se clasifican según la estructura de datos que utilicen.

1.5.2.2 Basados en árbol

Tomando como estructura de datos un árbol, cada búsqueda comienza en la raíz y se avanza por los nodos hijos del mismo hasta encontrar el prefijo más largo que coincida con la dirección buscada. La clave (Apéndice A) a buscar no se almacena en cada nodo del árbol sino en el propio camino recorrido en el árbol.

1.5.2.2.1 Árbol Patricia Patricia (*Practical Algorithm To Retrieve Information Coded In Alphanumeric*) [57, 62, 63, 64] es un algoritmo clásico de almacenamiento de claves de longitud variable, dentro del tipo de los algoritmos de clasificación digital. Consiste en un árbol binario en el que se evita la ramificación continua en una sola dirección incluyendo en los nodos el número de bits a saltar antes de hacer el siguiente test como se muestra en la figura 1.9. No busca igualdad exacta entre la clave (dirección IP a buscar) y el argumento (prefijo más largo almacenado en el árbol), pero determina si existe o no una clave que empiece con ese argumento. La búsqueda se implementa examinando un bit de la dirección cada vez mientras dure el recorrido del árbol. Cuando se alcanza una hoja, la dirección se compara con la información de enrutamiento almacenada en la hoja. Si coinciden, se ha obtenido la coincidencia máxima, y en caso contrario, se prosigue la búsqueda volviendo por los nodos padre aplicando las máscaras y repitiendo el proceso hasta encontrar la subred que la dirección satisfaga.

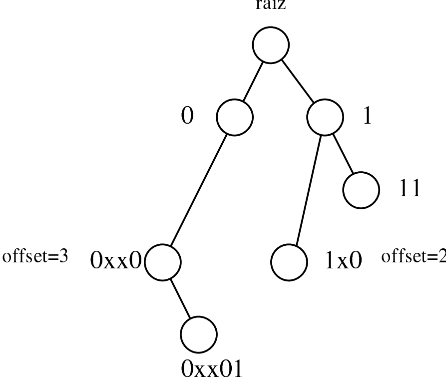


Figura 1.9: Ejemplo de estructura del árbol Patricia

En esta estructura los nodos poseen:

- Un par de punteros a los nodos hijo izquierdo y derecho.
- Offset, el número de bits a avanzar antes de realizar la comparación sobre los bits.

Básicamente el proceso consiste en testear bit a bit la secuencia de búsqueda. Empezando por el nodo raíz, si el primer bit es un 0 se avanza por la rama izquierda y si es un 1 se avanza por la rama derecha. Cuando el *offset* de un nodo es distinto de 0, quiere decir que el bit a testear está *offset* bits hacia adelante respecto a la posición en que se estaba testeando en ese

momento. Cuando se llega al final de una rama y el contenido del último nodo no coincide con la secuencia buscada, se retrocede hasta encontrar un nodo que indique una máscara de red. Entonces se aplica esa máscara y se repite el proceso con el resultante de aplicar la máscara en el subárbol restante. Si se sigue sin verificar se repite con máscaras de nodos padre, alcanzando al final coincidencia o llegando al nodo raíz si le corresponde la ruta por defecto.

De este modo tenemos un árbol como el de la figura 1.9 que representa cuatro secuencias 0xx0, 0xx01, 1x0 y 11. Por ejemplo, si estuviésemos buscando la secuencia 0100 se testearía el primer bit que al ser 0 nos llevaría por la rama de la izquierda, y después el bit cuarto (1+offset) que es 0, clasificándolo como 0xx0. Si se buscara una secuencia inexistente, por ejemplo, 01000, se avanzaría debido al primer bit 0 por la rama izquierda, debido al cuarto por la izquierda también y al testear el quinto no existiría esa rama, por lo que retrocediendo para atrás, llegaríamos al nodo 0xx0 y se repetiría el recorrido sobre el subárbol usando el resultante de aplicar la máscara asociada al nodo 0xx0 sobre la secuencia buscada. Así, se repetiría el proceso sobre nodos padre hasta encontrar alguna coincidencia o llegar a la raíz que significaría aplicar la ruta por defecto.

En la figura 1.10 se muestra una tabla de rutas en las que se puede observar la dirección de loopback (127.0.0.1) con su red, y otras direcciones destino como la 214.130.115.198 o subredes destino como la 214.130.114.0/24.

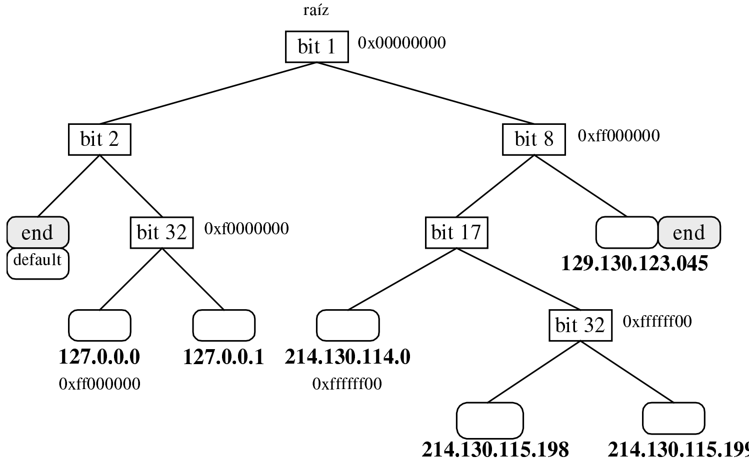


Figura 1.10: Ejemplo de tabla de rutas usando el algoritmo Patricia

El problema que tiene es que puede degenerar en más de 32 accesos a memoria para búsqueda de dirección IPv4 ya que el algoritmo es complejo incluyendo la parte de retroceso por el árbol. Por tanto si W es la máxima altura del árbol, es decir, la máxima longitud de las direcciones, la complejidad es de $O(W^2)$ iteraciones por búsqueda. Si está balanceado, esto equivale a $O(N \ln 2)$ donde N es el número de entradas del árbol y verifica que $N \leq 2^W - 1$. Pero normalmente la búsqueda sin éxito suele ocurrir en nodos externos más cercanos a la raíz, por lo que el coste baja ostensiblemente.

Este algoritmo se ha utilizado para representar las tablas de rutas en sistemas operativos como FreeBSD y Linux [64], y también en routers comerciales. El árbol Patricia soporta prefijos de cualquier longitud, y se dice que es de tipo *path-compressed* porque acorta las

ramas del árbol al usar el parámetro *offset* cuando testea cada bit y elimina aquellos nodos que sólo poseen un nodo hijo. En la figura 1.11 se presenta un árbol binario y en la figura 1.12 el árbol Patricia correspondiente.

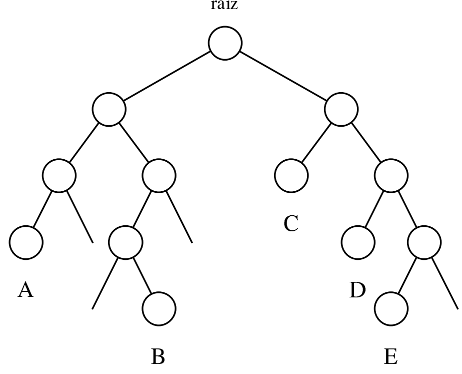


Figura 1.11: Árbol binario

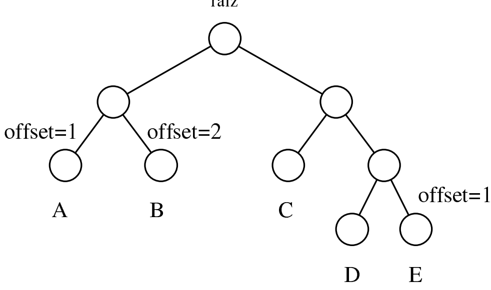


Figura 1.12: Árbol Patricia correspondiente al árbol binario de la figura 1.11

Una variante de esta estructura es el *árbol de prefijos dinámicos* [65] que simplifica el borrado y el recorrido inverso recursivo del árbol, consiguiendo una complejidad de $O(2W)$. Otra alternativa hace uso de una variable que va almacenando el camino seguido en el recorrido descendente del árbol para evitar el recorrido ascendente recursivo consiguiendo de esta forma una complejidad de $O(W)$ [66].

1.5.2.2.2 Árbol con compresión de nivel El árbol con compresión de nivel (*Level-Compressed Trie, LC-Trie*) [67] comprime el árbol anterior eliminando los nodos que sólo tengan un nodo hijo y convierte el nodo resultante en un nodo de mayor grado en el que se va a comparar más de un bit. A esta estructura especial en árbol con varias ramas por nodo se le denomina *Trie*, cuyo nombre viene de 'information re*trie*val'.

Esta técnica muestra menos dependencia de la distribución específica de las direcciones en las tablas de enrutamiento, pero sus estructuras en memoria no facilitan la modificación dinámica del árbol.

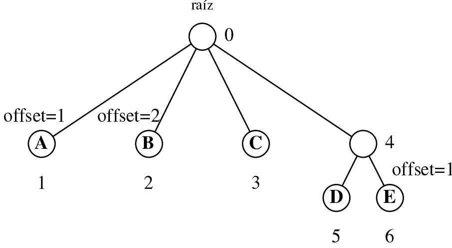


Figura 1.13: LC-Trie correspondiente al árbol binario de la figura 1.11

La técnica de compresión de nivel consiste en la compresión de partes del árbol que se encuentren densamente ocupadas. La idea es reemplazar los i niveles completos más altos del árbol binario por un único nodo de grado 2^i, donde i denota el factor de ramificación. Esta sustitución se realiza recursivamente para cada subárbol. En la figura 1.13 se presenta el árbol con compresión de nivel correspondiente al árbol binario y al árbol Patricia mostrados anteriormente (figuras 1.11 y 1.12 respectivamente). Internamente, el árbol se almacena como una tabla en la que la primera columna indica el número de nodo (numerados en orden "primero el más profundo"). La segunda columna indica el valor k tal que si $k \geq 1$ indica que el nodo interno tiene 2^k nodos hijo y si $k = 0$ indica que es un nodo hoja. La tercera columna indica el valor de *offset* a avanzar en la búsqueda. Finalmente, la última columna es el puntero con dos interpretaciones: para un nodo interno apunta al nodo hijo de más a la izquierda y para un nodo hoja es un puntero a un vector base que contiene las cadenas sobre las que testear la coincidencia de bits. En la tabla 1.3 se muestra el array resultante para el LC-Trie de la figura 1.13.

Nodo	*k*	*Offset*	*Puntero*
0	2	0	*1*
1 (nodo A)	0	1	0
2 (nodo B)	0	2	1
3 (nodo C)	0	0	2
4	1	0	*5*
5 (nodo D)	0	0	3
6 (nodo E)	0	1	4

Tabla 1.3: Array que representa el LC-trie anterior

Por tanto, en cada nodo se testean ahora los k bits que correspondan según el *offset* acumulado hasta ese nodo. Al llegar a un nodo-hoja nos queda acudir al vector base, según la posición indicada por el apuntador del nodo, para verificar si se trata de una verdadera coincidencia. Cuanto mayor sea el factor de ramificación i, más decrecerá la altura del árbol, y el incremento de memoria necesario será moderado.

1.5.2.2.3 Árbol Multiresolución Comprimido Una variante de los árboles de prefijos son los árboles multiresolución comprimidos [56]. Consisten en la codificación del árbol de

prefijos en tablas según lo expuesto en la figura 1.14. Cada nodo se define con 3 bits según sea un nodo interno (representados en blanco) o un nodo externo que haga referencia a una máscara o a una dirección final (representados en gris).

	001	010	011
100	101	110	111

Figura 1.14: Codificación de nodos para la definición del árbol multiresolución

En la figura 1.15 se presenta un ejemplo de árbol multiresolución. En ella, se observa cómo el árbol se subdivide en grupos de a lo sumo 3 nodos para su mejor codificación. La numeración de los nodos se realiza por niveles. Por ejemplo, el subárbol T_1 se codificaría como se muestra en la figura 1.16. El primer número apunta a otra tabla donde se encuentran los prefijos propiamente dichos, el segundo número apunta a la tabla que contiene el siguiente árbol y los otros 9 bits codifican los tipos de los 3 nodos.

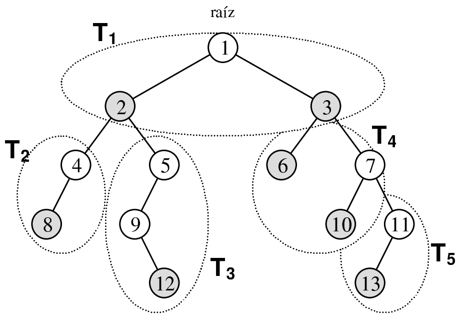


Figura 1.15: Ejemplo de árbol multiresolución

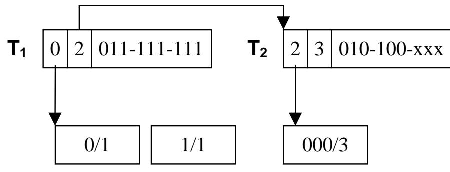


Figura 1.16: Codificación de los subárboles T_1 y T_2

La búsqueda se realiza de forma inmediata sobre las estructuras presentadas, ya que se puede realizar en cada subárbol T_i como si fuera un árbol habitual de prefijos, con las ventajas de un menor espacio de memoria ocupado y una velocidad de búsqueda mayor.

1.5.2.3 Basados en comparaciones

Consiste en una búsqueda binaria sobre un array, con modificaciones para soportar la coincidencia de varias subredes sobre una dirección de entrada [68] y que se quede con aquel prefijo más largo. Normalmente el coste de inserción y eliminación de entradas es muy alto al exigir una reorganización de las estructuras de datos, por lo que su aplicación al almacenamiento de tablas de enrutamiento dinámicas es limitada.

1.5.2.4 Basados en hashing

El resultado de las funciones de hashing se utiliza como índice de memoria para acceder a los datos de enrutado para esa dirección IP [66]. A continuación se muestran algunos tipos.

Content-Addressable Memory (CAM) [69] Se puede considerar como un dispositivo hardware de hashing perfecto, que busca automáticamente la dirección IP y devuelve la ruta de salida. De manera natural, no soportan la búsqueda del prefijo de máxima coincidencia, y los algoritmos sobre CAM que lo consiguen tienen un rendimiento muy bajo.

En los dispositivos CAM se compara, en paralelo, la clave de entrada con el contenido de cada posición de memoria. Incluso algunos dispositivos CAM permiten el uso de máscaras de bits, ya sea fijados a priori o mediante máscaras almacenadas en ciertas posiciones de las CAM. Sin embargo, estas soluciones son muy costosas, ya que se necesita realizar la búsqueda sobre todo el rango de direcciones y no soporta de forma nativa la búsqueda por coincidencia del prefijo más largo ni tampoco un número variable de bits en las direcciones.

Caché Se basan en la localidad de las direcciones destino en Internet (fuerte correlación temporal de las direcciones) [70]. Se intenta minimizar el tamaño de las tablas de forma que una tabla de rutas con 40.000 entradas no ocupe más de 160 KB y de esta forma tenga cabida en la caché del procesador. De nuevo la búsqueda por coincidencia del prefijo más largo no es realizable de forma inmediata.

Arquitecturas de gran memoria Hacen uso de memoria para hacer un hashing perfecto de todas las posibles direcciones IP destino. Se organiza en bancos raíz de 2^{24} y hojas de 2^8 entradas para direcciones IPv4.

1.5.2.5 Basados en la combinación de hashing y árbol

Combinan las ventajas de ambos métodos hashing y árbol, anteriormente explicados. La combinación de ambos métodos conseguirá reducir la memoria necesaria con respecto al hashing y mejorar la velocidad con respecto a las búsquedas en árbol. A continuación se describen diferentes alternativas a la hora de combinar ambos métodos.

1.5.2.5.1 Búsqueda binaria en tabla de hash En la búsqueda binaria en tabla de hash [66] los prefijos de distintas longitudes se sitúan en una tabla de hash, sirviendo la longitud del prefijo como índice de acceso a la tabla. En el caso de que varios prefijos coincidan en su longitud se almacenan en una lista enlazada para la posición de la tabla indexada por esa longitud. Para buscar una dirección de longitud L se empieza por la posición media de esas longitudes (L/2), se recorre la lista si existen varios prefijos con esa misma longitud, y gracias al uso de marcadores se salta a otra posición de la tabla de hash con una búsqueda binaria. El coste es de $O(logW)$ donde W es la máxima profundidad de la estructura.

Para optimizar el algoritmo se usan diversas técnicas:

- El uso de marcadores además de prefijos permite realizar una búsqueda por coincidencia del prefijo más largo. Los marcadores no son más que prefijos asociados con punteros a subárboles de manera que al recorrer una lista de entradas y no encontrar una entrada coincidente con la solicitada, se busca un marcador que sí coincida y me lleve a un subárbol que me permita continuar la búsqueda.
- Cuando se llega al final y no se ha encontrado plena coincidencia, es preciso retornar en sentido contrario hasta encontrar la más próxima. Para evitarlo se puede almacenar el nodo anterior en una variable auxiliar.
- Se puede refinar el esquema básico haciendo un preprocesado de las subredes a almacenar. Una mejora inmediata es limitar la búsqueda a las longitudes de prefijos con alguna entrada. Otra mejora es cambiar la asimetría del árbol de forma que se consulten primero las entradas más probables en la distribución no empezando necesariamente por el medio de la tabla de hash.
- El mecanismo de mutación consiste en que siempre que se consigue una coincidencia parcial y se mueve a un subárbol, sólo es necesario realizar búsqueda binaria en los niveles del nuevo subárbol.

1.5.2.5.2 Árbol con hash En la técnica de árbol con hash los primeros bits de la dirección IP se utilizan como índice de acceso directo a una tabla de hash en la que de cada posición cuelga un árbol para continuar la búsqueda. En concreto, los K primeros bits son el índice de acceso a la tabla donde se encuentra el nodo raíz del árbol. A partir de ahí si el bit $(K+1)$-ésimo es 0 se continua por el nodo hijo izquierdo y si es 1 por el derecho. Se repite el proceso para el resto de bits hasta encontrar una coincidencia perfecta o alcanzar un nodo sin hijos.

Si se encuentra coincidencia perfecta se obtiene de manera inmediata un puntero a la zona de memoria que contiene el interfaz de salida. En el otro caso, mientras se realiza el proceso de búsqueda siempre se encuentran subredes coincidentes y en todo caso se mantiene la mejor (la de prefijo coincidente más largo). Así, al alcanzar un nodo sin hijos, la mejor coincidencia obtenida hasta el momento será la que se tome como resultado.

1.5.2.5.3 Codificación jerárquica de un árbol de prefijos completo La técnica de codificación jerárquica de un árbol de prefijos completo [68] interpreta los prefijos como intervalos dentro del árbol, definiendo tres niveles con los primeros 16, 24 y 32 bits. Realizando la

expansión total del árbol binario, en cada uno de estos niveles tendremos 2^{16}, 2^{24} y 2^{32} nodos que se representan con un bit. Este bit valdrá uno si el nodo tiene nodos hijos (continua el subárbol), si se trata de una hoja o si hay una hoja en un nivel anterior. Valdrá cero si es un nodo cubierto por una hoja en algún subnivel anterior. Haciendo uso de esta estructura y otras tablas, se tiene referencia directa del interfaz asociado si se ha alcanzado una hoja, o un puntero al subárbol hacia el que continuar la búsqueda si todavía no se ha alcanzado el prefijo de coincidencia máxima.

1.5.2.6 Otros esquemas de enrutamiento: basados en etiquetas

Una forma de evitar los problemas de búsqueda de IPs en tablas de enrutamiento es añadir al paquete algo de información extra que permita simplificar esta búsqueda [68]. Este es el caso del *Tag Switching* que consiste en añadir una etiqueta identificativa a cada paquete perteneciente al mismo flujo. Se entiende por flujo aquel conjunto de paquetes con igual dirección IP origen y destino, e igual puerto origen y destino. Esta etiqueta puede ser un simple entero que permite un acceso directo al interfaz de salida del router, y se mantiene en todos los saltos del paquete hasta llegar a destino. Sin embargo estos esquemas no eliminan la necesidad de al menos realizar una búsqueda por los métodos tradicionales en los router frontera que dividan zonas con mapeo tradicional y zonas que entiendan de estas etiquetas.

Una propuesta del IETF [71] es el Multi-Protocol Label Switching (MPLS) [72, 73] que está basado en ideas de la conmutación ATM, es compatible con protocolos de enrutamiento RSVP/IntServ [74], y soporta multicast y agregación de flujos.

1.5.2.7 Conclusiones de clasificación en routers

En la tabla 1.4 se presenta un resumen de la complejidad para los distintos algoritmos descritos. La W denota el número de bits por cada dirección ($W = 32$ en IPv4), N el número de prefijos almacenados y k es una constante propia de cada algoritmo.

Algoritmo	*Inserción*	*Borrado*	*Búsqueda*
Búsqueda binaria	$O(N)$	$O(N)$	$O(\log N + W)$
Patricia-Trie	-	-	$O(W^2)$
Árbol de prefijos dinámicos	-	-	$O(W)$
LC-Tries	-	-	$O(\frac{W}{k})$
Árbol multiresolución comprimido	-	-	$O(\log_k N + 1)$
Arquitecturas de gran memoria	-	-	$O(\frac{W}{k})$
Búsqueda binaria en tabla de hash	-	-	$O(\log W)$
Árbol con hash	-	-	$O(\frac{W}{k})$

Tabla 1.4: Complejidad de varios algoritmos de búsqueda en tablas de enrutamiento

Se observa en la tabla que algunos algoritmos como el árbol con hash alcanzan muy buenos costes de computación en las búsquedas. Sin embargo, estos algoritmos de clasificación en

routers de alta velocidad tienen por objetivo la búsqueda de una dirección IP o una subred que no es inmediatamente aplicable a nuestro problema de filtrado de paquetes en el que es necesario concatenar el testeo de distintos campos. *Se puede hacer el símil de que la clasificación en routers se asemeja a la monitorización de un único parámetro (la dirección IP destino), y por tanto una monitorización real se asemeja a un número muy grande de clasificaciones simultáneas en el router.* Aunque el escenario es diferente al de los sistemas de monitorización, se ha considerado su inclusión porque la búsqueda de una dirección IP en las tablas de un router equivale a buscar patrones de bits y por tanto a un filtro que verifica una sola condición.

De todas formas, como regla general, sí parece ser que las estructuras en árbol junto con funciones de hashing forman una buena alternativa como base de un posible algoritmo de filtrado de paquetes.

1.5.3 Pila de protocolos en Sistemas Operativos

Un sistema operativo tiene la necesidad de realizar el filtrado de los paquetes desde el momento en que proporciona el uso transparente de canales de comunicación e incluso puede realizar funciones de enrutamiento. Dentro del funcionamiento del soporte de red en un sistema operativo de propósito general podemos distinguir dos partes. Una es la decodificación de protocolos, que permite ofrecer al usuario canales de comunicación de manera transparente, y otra es el enrutamiento.

En cuanto a la decodificación de protocolos, esta se realiza normalmente a nivel de kernel del sistema operativo [75], mediante la simple llamada a funciones anidadas encargadas de decodificar cada nivel de protocolos y hacer disponible la información obtenida al siguiente nivel de protocolo que encapsula. Así, se provee a las aplicaciones de puntos de acceso al servicio diversos como pueden ser los de IPX o los sockets TCP/IP. Para ello se utiliza una serie de estructuras a nivel de kernel, a las que se accede normalmente con técnicas hashing, que llevan cuenta de las conexiones activas y unos buffers para el socket (*sk_buff*) que permiten gran flexibilidad para añadir y suprimir cabeceras de protocolos.

En el enrutamiento, se utilizan técnicas similares a las ya vistas en el apartado anterior. Por ejemplo, el árbol Patricia es el utilizado en el Berkeley Unix BSD [64]. Es necesario que un sistema operativo sea capaz de enrutar desde el momento en que puede tener varios interfaces de conexión a diversas redes y ha de saber por cual sacar cada paquete dependiendo de la dirección IP destino.

1.5.4 NNstat

El sistema NNstat [76] de monitorización de redes implementa el árbol FPT (Field Parse Tree) para el filtrado de paquetes [20]. Entre las funcionalidades que el sistema NNstat incluye destacan las siguientes:

- Frecuencia de todos o parte de los valores de un campo.

- Frecuencia de pares de valores (útil por ejemplo para la matriz de tráfico).
- Histograma de un valor.
- Filtro de un valor o entre un rango de valores.
- Localidad temporal de los valores.

El árbol FPT está compuesto por nodos que representan filtros a aplicar sobre determinados campos de la cabecera de un protocolo (nodo filtro) y unas estructuras parámetro separadas encargadas de contabilizar las estadísticas deseadas. El recorrido del árbol es recursivo y hace uso de funciones de hash para optimizar el acceso a estructuras.

Esta propuesta ni reutiliza subfiltros comunes ni contiene los parámetros de monitorización en el propio árbol haciendo uso de una estructura asociada. Además se trata de una estructura estática para la que añadir o eliminar un filtrado supone rehacer todo el árbol, con el consiguiente coste elevado de computación [14]. Otras características del mismo son:

- El analizador está basado en un intérprete de pseudo-máquina.
- El programa de comandos de monitorización se compila.
- La información de protocolos se encuentra compilada en el propio intérprete de pseudo-máquina. Argumenta que los protocolos no cambian a menudo. Si bien puede ser verdad, se pierde la flexibilidad de un sistema de monitorización y se cierra su aplicación por ejemplo a la depuración de protocolos de comunicaciones entre aplicaciones.

Aunque sus limitaciones son importantes, el sistema NNstat ha sido muy utilizado para llevar estadísticas genéricas de monitorización de red a largo plazo (por ejemplo en NSFnet, National Science Foundation NETwork).

1.6 Objeto del presente trabajo

A lo largo de este capítulo se ha visto que el filtrado de paquetes es la parte fundamental en sistemas de monitorización de red. Las limitaciones de un sistema de monitorización dependen en gran medida del subsistema de filtrado. Además, se hace necesario una adaptación de la técnica de filtrado a la problemática concreta de los sistemas de monitorización.

Para ello, una buena referencia parece ser la de los *packet filter*. En ellos se ha visto cómo se pasan los paquetes filtrados al proceso servidor a costa de crear un flujo de transmisión separado por cada secuencia de filtros definida. Leyendo estos flujos el subsistema servidor es capaz de llevar las estadísticas, pagando el coste que supone las copias en memoria de los paquetes que circulan entre el subsistema de filtrado y el servidor, además de la falta de reutilización de filtros comunes en la mayoría de ellos. Por tanto, no parece ésta (packet filter + analizador de flujos) la forma más adecuada de resolver el problema de monitorización.

Por otro lado, los algoritmos de clasificación en routers están pensados para trabajar sobre una información básica del próximo salto que ha de dar cada paquete según la IP destino. Por

tanto, clasifican los paquetes teniendo únicamente en cuenta este campo y teniendo presente la posibilidad de filtrar las múltiples posibilidades. En cambio, para un sistema de monitorización se ha de poder filtrar sobre diferentes campos.

En lo que respecta a la forma de decodificación de protocolos de los sistemas operativos, su aplicación a un sistema de monitorización no es inmediata porque están diseñados para filtrar los paquetes destinados a la misma máquina y no cualquier otra. Además se trata de código en el kernel que soporta el desencapsulado de ciertos protocolos únicamente, sin flexibilidad a la hora de quedarse con ciertos paquetes con alguna característica.

Por tanto, se ha de buscar una forma más eficiente de filtrado y decodificación de paquetes que suponga un aumento de la eficiencia de los sistemas de monitorización. Será necesario por tanto encontrar unas técnicas de filtrado que se adecuen a las características específicas de los sistemas de monitorización.

El objetivo de este trabajo es la propuesta y análisis de una técnica eficiente de filtrado de paquetes de datos orientadas a su aplicación en sistemas de monitorización de tráfico de redes de comunicaciones. Será necesario buscar una técnica que presente las siguientes características principales:

- Decodificación: ha de permitir decodificar paquetes, permitiendo flexibilidad en la incorporación de nuevos protocolos y aplicaciones de red.

- Filtrado: ha de permitir diferenciar los paquetes que cumplan series de filtros definidas por el sistema de monitorización.

- Reutilización de filtrado: la técnica de filtrado a proponer tendrá que minimizar el número de filtros a testear. Para ello, se tendrá que reaprovechar al máximo esos filtros, evitando su duplicidad y aprovechando el carácter jerárquico y en niveles de las pilas de protocolos.

- Almacenamiento: deberá ser capaz de llevar información de monitorización. Por ejemplo, podrá haber contadores que lleven cuenta de los paquetes que cumplen esos filtros o de los bytes que suman esos paquetes. Al sondearlos periódicamente podremos tener las tasas de esos parámetros.

En los siguientes capítulos se presenta una propuesta de algoritmo de filtrado, junto con su formalización, evaluación analítica y evaluación experimental.

Capítulo 2

Algoritmo de filtrado PAM-Tree

2.1 Introducción

Un sistema de monitorización, para ofrecer información de monitorización en tiempo real, capturas, informes periódicos o alarmas, necesita obtener ciertos parámetros de la red con un muestreo temporal determinado. Para ello necesita establecer una serie de filtros por cada parámetro que se desee monitorizar, de manera que todo paquete que llegue de la red se compruebe contra esos filtros. Por tanto, el número de filtros simultáneos puede ser muy alto. Por ejemplo, la matriz de tráfico entre 20 estaciones de la red supondría 400 filtros compuestos de comprobación a nivel Ethernet (para comprobar que encapsule IP) e IP (para comprobar que las direcciones IP origen y destino son unas determinadas). El coste de este filtrado ha de ser el menor posible para que el sistema sea capaz de soportar el mayor número de filtros simultáneos y altas cargas de red.

El presente trabajo tiene en cuenta una serie de peculiaridades del tráfico de red que ayuda a optimizar la realización del filtrado frente a las soluciones existentes hasta el momento:

- La jerarquía de protocolos en niveles de encapsulado sugiere un esquema de filtrado también jerárquico que mantenga la relación entre niveles y se pueda aprovechar de este hecho.

- La repetición de comprobaciones idénticas entre filtros diferentes también sugiere la idea de reutilización de estas comprobaciones. Es decir, que la misma comprobación sirva para los filtros que la comparten.

- La necesidad de llevar contadores del valor de los parámetros de monitorización sugiere su inclusión dentro del propio algoritmo de filtrado.

El subsistema de filtrado es la parte más importante y también más costosa del sistema de monitorización. Los términos de coste se reparten en tres aspectos. Por un lado, el coste de procesado de los filtros será el factor determinante. Por otro lado, la cantidad de memoria utilizada y finalmente el coste del equipo hardware necesario también será importante.

El subsistema de filtrado puede ser consultado para conocer el valor de alguno de los parámetros establecidos, los cuales se actualizan para cada paquete que se transmite a través de la red de comunicaciones que se está monitorizando. Este subsistema de filtrado establece parámetros que se solicitan desde las diferentes consolas, y los elimina cuando sea requerido.

En este capítulo presentaremos una propuesta de algoritmo de filtrado y haremos uso del formalismo de Autómatas de Entrada/Salida [77] porque es una herramienta útil para describir las interacciones entre el entorno y el propio algoritmo de filtrado. Más aún, provee una descripción formal de las propiedades principales del subsistema de filtrado independientemente de la implementación final. La formalización permitirá definir las interacciones como conjunto de acciones. También comprobaremos que el algoritmo propuesto satisface las siguientes propiedades:

- Los filtros siguen una jerarquía en árbol.
- Los nodos subfiltro se reutilizan.

2.2 Conceptos previos

Los paquetes son los elementos de información que circulan por las redes de comunicaciones. La transferencia de la información se basa en la conmutación de paquetes. El conjunto de todos los posibles paquetes que pueden circular por dichas redes vamos a denotarlo por $\mathcal{P}$. Estos paquetes están formados por un conjunto de bits que se pueden agrupar en unidades mayores que vamos a denominar campos. A continuación se definirá un paquete como la composición de un conjunto finito de campos.

Un paquete $P \in \mathcal{P}$ es una cadena finita de elementos c_i, los cuales pueden tomar el valor 1 o 0 (bits). Por tanto, un paquete $P = c_1c_2\ldots c_{length(P)}$ se define del siguiente modo $P = \bullet c_k : k = 1..length(P)$, donde $\bullet$ denota la operación de concatenación de cadenas, siendo *first(P)* y *length(P)* funciones que devuelven el primer elemento de la lista (c_1) y el número de bits del paquete, respectivamente.

Los paquetes se pueden dividir en grupos de bits consecutivos que denominamos campos, E_{jk}. Denominaremos $\mathcal{E}$ al conjunto de todos los campos posibles.

Definición 1 *: Denominaremos campo $E_{jk}(P) \in \mathcal{E}$ a parte de la secuencia de bits de un paquete, $E_{jk}(P) = \bullet c_i : i = j..k$, donde j y k indican las posiciones del primer y último bit del campo.*

En la figura 2.1 aparece un ejemplo de cabeceras de protocolos que están compuestas por una serie de campos con determinado significado. Los campos a los que nos estamos refiriendo aquí no tienen por qué coincidir con un campo de una cabecera de protocolos aunque normalmente así será.

Cada campo se define por el primer y último bit que forman parte del campo con respecto

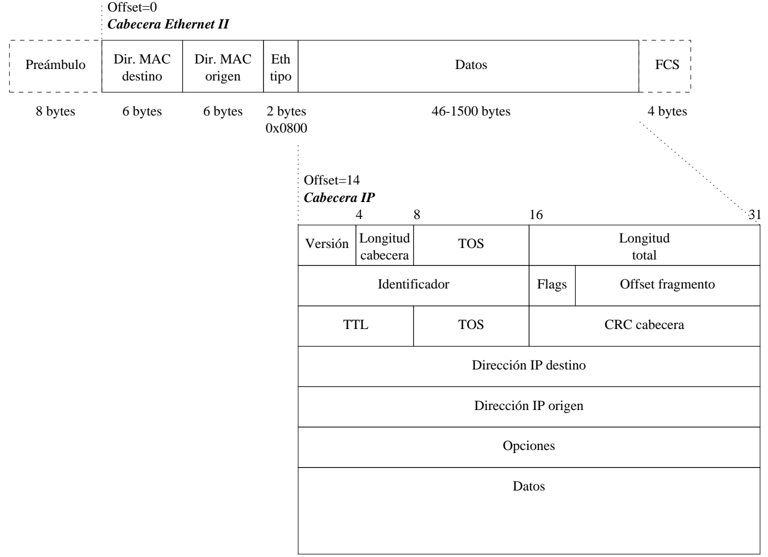


Figura 2.1: Ejemplo de cabeceras y campos

al comienzo del paquete por lo que se obtendrán como:

$$\begin{cases} j = (Offset * 8) + StartBit \\ k = (Offset * 8) + StopBit \end{cases} \tag{2.1}$$

donde *Offset* indica la posición en bytes del comienzo de la cabecera de protocolo que se está procesando y *StartBit*/*StopBit* son las posiciones relativas en bits de los bits de ese campo medidos desde el comienzo de esa cabecera. De esta manera, se consigue la definición de condiciones en determinados bits de la cabecera independientemente del nivel de protocolo en que se encuentren. Esto es útil, por ejemplo en protocolos como IP que pueden aparecer sobre Ethernet o encapsulados a su vez dentro de IP. En ambos casos tener una única definición relativa de los campos de la cabecera IP será suficiente.

Cada campo tiene una interpretación como valor decimal que se calcula mediante la siguiente función: $Val(E_{jk}(P))) = c_j 2^{k-j} + c_{j+1} 2^{k-j-1} + \ldots + c_k 2^0$. A partir de la definición de campo se puede caracterizar un paquete como la composición de un conjunto finito de campos, $P = E_{0i}(P) \bullet E_{(i+1)j}(P) \bullet \ldots \bullet E_{(k+1)n}(P)$, con $i, j, k, n \in \mathbb{Z}^+, 0 < i < j < k < n$.

Los sistemas de monitorización tienen por cometido principal la actualización de un conjunto de parámetros que se han definido previamente. Estos parámetros se actualizan para cada paquete que se transmite por la red de comunicación. Cada parámetro se actualiza sólo en caso de que se cumplan un conjunto de condiciones. Al conjunto de condiciones se denomina filtro. Estas condiciones estarán relacionadas entre sí mediante operaciones lógicas "AND" y/u "OR".

Una condición es la sentencia que sirve para determinar si un campo de un paquete es el que se desea. Normalmente consiste en comprobar si el campo coincide con cierta secuencia de bits definida en la regla, pero también puede ser la comprobación de que sea mayor, menor, distinto, etc.

Definición 2 : *Una condición viene definida por la siguiente tupla:*

$$F_i = (E_{jk}, Value, Function, ComparisonType) \tag{2.2}$$

donde $E_{jk} \in \mathcal{E}$ identifica el campo sobre el que se evalúa la condición, $Value \in \mathbb{Z}$ es el valor con que se compara la salida de la función $Function$ que se aplica al campo E_{jk} para obtener una salida intermedia. Para comprobar si se verifica el filtro se comprobará si la salida intermedia y $Value$ se relacionan según el $ComparisonType \in \{=, <, >, ! =\}$ que determina la relación lógica. La denominaremos indistintamente condición o subfiltro por ser un bloque básico del filtro.

Definición 3 : *Denominaremos filtro a la composición de condiciones mediante operadores lógicos AND/OR que se verificará si y sólo si se verifican las relaciones lógicas definidas entre ellos. En nuestro caso, cada filtro tendrá asociado al menos un parámetro de monitorización.*

Las condiciones se pueden combinar según las operaciones lógicas AND u OR para formar un filtro que es a lo que se asocia un parámetro de monitorización. Para la formalización se puede establecer que cualquier filtro, compuesto por cualquier combinación de condiciones AND/OR, se puede convertir en un OR de secuencias de términos AND. Es decir, las condiciones F_i se asocian entre ellas sólo con operaciones lógicas AND, y las asociaciones resultantes F_{OR_j}, que llamaremos componentes OR, se asocian entre sí mediante operaciones lógicas OR. Toda combinación de condiciones AND/OR se puede convertir a este formato aplicando las propiedades asociativa y la distributiva de AND sobre OR [78].

Tengamos por ejemplo el siguiente filtro G, donde "$\wedge$" representa una operación lógica AND y "$\vee$" una operación lógica OR:

$$G = (G_1 \wedge (G_2 \vee G_3)) \vee (G_4 \wedge G_5)$$

se puede convertir de manera inmediata en:

$$G = (G_1 \wedge G_2) \vee (G_1 \wedge G_3) \vee (G_4 \wedge G_5) = G_{OR_1} \vee G_{OR_2} \vee G_{OR_3}$$

De forma genérica:

$$F = F_{OR_1} \vee F_{OR_2} \vee \ldots \vee F_{OR_j} \tag{2.3}$$

$$F_{OR_j} = F_1 \wedge F_2 \wedge \ldots \wedge F_{n_j} \tag{2.4}$$

Esta descomposición nos será útil posteriormente para que nuestro algoritmo soporte relaciones lógicas OR entre condiciones. También permitirá descubrir de manera sencilla secuencias de condiciones comunes entre filtros y por tanto poder reutilizar esas secuencias. En el caso de que existan sólo relaciones AND de subfiltros, se tendrá un único componente OR (F_{OR_1}).

Un aspecto interesante en el filtrado es la reutilización de bloques de los que está compuesto el filtro, ya que se reduce el coste computacional de análisis de los paquetes. Ante diferentes

parámetros asociados a filtros con condiciones comunes, se tratará de realizar una sola vez la comprobación de estas condiciones comunes.

En lo que sigue se va a ilustrar la idea de condiciones comunes. Las condiciones comunes se presentan entre componentes OR de un filtro o asociadas a diferentes filtros. Vamos a considerar dos secuencias de condiciones f y g, con $a, b \in \mathbb{Z}^+, b > a$:

$$\begin{aligned} f &= F_1 \wedge F_2 \wedge \ldots \wedge F_a \\ g &= F_{a+1} \wedge F_{a+2} \wedge \ldots \wedge F_b \end{aligned}$$

La composición AND de estas secuencias resulta en:

$$f \wedge g = F_1 \wedge F_2 \ldots \wedge F_a \wedge F_{a+1} \wedge F_{a+2} \wedge \ldots \wedge F_b$$

De esta forma, en general, si se tiene dos componentes OR del mismo o diferentes filtros, con f, g, h secuencias de condiciones:

$$\begin{aligned} F_{OR_1} &= f \wedge g \\ F_{OR_2} &= f \wedge h \end{aligned}$$

se podrá decir que ambos componentes OR *comparten una misma secuencia de condiciones* f. Uno de los objetivos del sistema de filtrado será tratar de comprobar las secuencias de condiciones comunes a varios parámetros tan sólo una vez por paquete.

Definición 4 *: Se dirá que dos filtros F y G comparten una misma secuencia de condiciones f, que denotaremos por $F|_f G$, si contienen algún par de componentes OR, F_{OR_i} y G_{OR_j}, tal que $F_{OR_i} = f \wedge g$ y $G_{OR_j} = f \wedge h$.*

Finalmente, el sistema de monitorización se encarga de actualizar parámetros. Éstos se actualizan en función del filtro que cumple el paquete. Es decir, cada parámetro está asociado a un filtro, el cual determina cuando se debe actualizar el parámetro. Por tanto, cada parámetro se define a partir del filtro que deben cumplir los paquetes y con la función que realiza su actualización. Esta función puede ser un incremento o una función más compleja.

Definición 5 *: Un parámetro Q se define como:*

$$Q = (F, UpdateFunction) \tag{2.5}$$

donde F es un filtro y $UpdateFunction$ es la función de actualización que se aplica al parámetro e indica en qué forma se tiene que incrementar el contador asociado. El conjunto de todos los parámetros se denota por $\mathcal{Q}$.

Un ejemplo de parámetro sería “bits de paquetes IP” que consistiría en un filtro compuesto por una única condición que testeará el campo de protocolo del paquete Ethernet para que fuese IP, y un *UpdateFunction* que devolviera el tamaño del paquete recibido para así contabilizar los bits totales recibidos de paquetes de ese tipo.

A todo parámetro insertado en el sistema de monitorización se le asignará un identificador único ID que facilitará las tareas de borrado y consulta del parámetro. El conjunto de todos los identificadores posibles vendrá denotado por $\mathcal{ID}$.

2.3 Especificación

Un sistema de monitorización se puede considerar, de una manera simplificada, formado por un conjunto de procesos de aplicación (consolas) que establecen, solicitan y eliminan parámetros de monitorización a procesos de filtrado de tráfico (sondas), según una arquitectura cliente/servidor. Los procesos de filtrado trabajan en función de dos condiciones: la presencia de paquetes en la interfaz de red que monitorizan y siempre que algún proceso de aplicación haya establecido algún parámetro para monitorizar. El sistema de monitorización analiza los paquetes para actualizar los parámetros configurados. Finalmente, envía la respuesta a la consulta de un parámetro siempre que dicho parámetro esté configurado previamente.

El entorno del sistema modela las operaciones que afectan al sistema de filtrado. Por un lado, el establecimiento, consulta y borrado de los parámetros de monitorización. Y, por otra parte, la recepción de los paquetes que circulan por los enlaces de comunicación que se monitorizan. No nos vamos a preocupar del modelado del entorno ya que lo que nos interesa únicamente son sus interacciones con el sistema de filtrado.

El sistema se considera formado por un conjunto de lugares donde se tienen procesos de aplicación y de filtrado. Estos lugares están dispersos y entre ellos se comunican mediante paso de mensajes. Cuando se emplea la expresión de entorno del sistema se está hablando del comportamiento de la unión de todos estos lugares. Cada lugar controla en cada instante un único proceso de aplicación o de filtrado. Esta simplificación permite una exposición más cómoda e inteligible de este trabajo.

La descripción del entorno se realiza desde el punto de vista del sistema de filtrado, es decir, sólo se considera la parte del entorno que tiene que ver con la creación, consulta y borrado de parámetros de monitorización, así como la captura de paquetes a través del canal de comunicación. Este hecho permitirá desarrollar el algoritmo del subsistema de filtrado evitando la complejidad que introducen todas las operaciones de la consola. De esta forma, el entorno va a estar formado por la consola, el subsistema de captura y el subsistema servidor. El entorno hará peticiones al subsistema de filtrado y recibirá las respuestas correspondientes (ver figura 2.2).

Por una parte, el entorno interacciona con el subsistema de filtrado mediante las siguientes acciones:

- **SetReq(Q)**: solicitud de comienzo de monitorización del parámetro definido por $Q \in \mathcal{Q}$ (expresión 2.5).

- **DelReq(ID)**: solicitud de finalización de monitorización del parámetro identificado por ID. Este ID es el identificador asignado al parámetro una vez que ya se ha activado su monitorización en el subsistema de filtrado. Ese identificador será único para cada parámetro monitorizado en el subsistema de filtrado y será útil para solicitar el valor del parámetro o solicitar el que deje de ser monitorizado (eliminarlo).

- **PollReq(ID)**: solicitud del valor actual del parámetro de monitorización identificado por ID.

- **Packet(P)**: los paquetes P que aparecen sobre la red monitorizada se capturan y pasan al subsistema de filtrado a través de este mensaje. El subsistema de filtrado actualizará los parámetros de monitorización en función de los filtros verificados por este paquete.

Por otra parte, el subsistema de filtrado responde al entorno mediante las siguientes acciones:

- **SetResult(ID)**: tras una solicitud $SetReq(Q)$, el subsistema de filtrado empieza a monitorizar este nuevo parámetro y devuelve el identificador ID asignado a este parámetro. Este identificador servirá para futuras referencias, en consultas del valor del parámetro y en peticiones para eliminar el parámetro.

- **DelResult(ID)**: tras una solicitud $DelReq(ID)$, el subsistema de filtrado elimina el parámetro con ese identificador y devuelve ese identificador ID.

- **PollResult(ID,Value)**: tras una solicitud $PollReq(ID)$, el subsistema de filtrado devuelve el valor $Value$ del parámetro correspondiente a ese identificador ID.

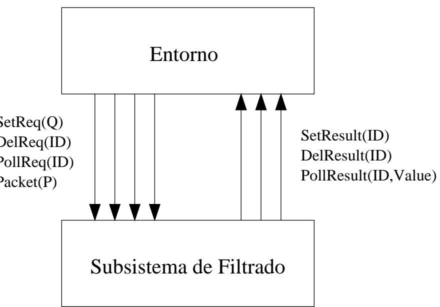


Figura 2.2: Entorno y subsistema de filtrado

En el sistema puede haber varias sondas funcionando de manera simultánea accesibles desde la misma consola. Es decir, podrá haber varios subsistemas de filtrado funcionando simultáneamente. Sin embargo, como los subsistemas de filtrado no interaccionan entre ellos, no es necesario incorporar a la notación subíndices que identifiquen los distintos subsistemas de filtrado. Así, sin pérdida de generalidad en el análisis, consideraremos un único subsistema de filtrado.

2.4 PAM-Tree como algoritmo de filtrado

El Protocol-Adaptive Monitoring Tree, PAM-Tree, es una propuesta novedosa en el campo de filtrado de protocolos con aplicación directa a la monitorización de redes de comunicaciones. Entre sus características fundamentales destacan las siguientes:

1. En primer lugar se basa en una estructura en árbol en la que cada nodo del mismo representa una condición. Cada uno de estos nodos que testea una condición se denominará *nodo subfiltro*. Las ramas del árbol se corresponderán con los diferentes filtrados establecidos. De esta manera se aprovecha la jerarquía mencionada de los protocolos de red.

2. La reutilización de condiciones se consigue compartiendo nodos subfiltro por parte de los filtros que comparten secuencias de condiciones comunes desde el nivel de protocolo inferior.

3. Se incorporan al árbol unas estructuras denominadas *nodos parámetro* que llevarán cuenta de las variables de monitorización asociadas a cada filtro. Es decir, se incorpora la información de monitorización en la propia estructura de filtrado, evitando tener que crear un flujo de salida por filtro (como en los packet filter).

4. El algoritmo se autoadapta a la arquitectura de filtrado de protocolos solicitada. Cada parámetro a monitorizar se extiende en una serie de nodos subfiltros que testean campos del paquete. Esta expansión se realiza de acuerdo a una base de datos de protocolos que facilita la tarea.

5. Se permite gran flexibilidad en la definición de las condiciones a testear sobre cualquier secuencia de bits del paquete, en la concatenación mediante operaciones lógicas AND/OR de estas condiciones y también en el modo de contabilizar el parámetro de monitorización deseado.

Veremos como el coste de procesado de los filtros mejora a las alternativas presentadas en el capítulo anterior. En cuanto a la memoria, no es un factor determinante si se implementa sobre una plataforma de propósito general como es nuestro caso. En cuanto al coste de hardware, su implementación sobre plataforma PC con sistema operativo Linux lo minimizan.

En los siguientes apartados se presentarán los detalles de la estructura de datos y algoritmo PAM-Tree.

2.4.1 Estructura del PAM-Tree

La estructura de datos del algoritmo PAM-Tree es un árbol (ver figura 2.3) en el que cada nodo se corresponde con un subfiltro y los parámetros de monitorización forman parte de la propia estructura de filtrado.

Un árbol es un tipo especial de grafo [79, 80] que consiste en un conjunto de puntos (nodos) en el espacio que están interconectados por un conjunto de líneas (arcos) sin lazos. Un arco se define mediante los 2 nodos que une. Por tanto, un árbol es un grafo conectado sin circuitos:

- conectado: si todos sus nodos están conectados y se puede ir siempre de uno a otro por algún camino (combinación de nodos y arcos asociados).

- sin circuitos: si en cada camino posible cada nodo aparece a lo sumo una vez.

Grado de un nodo es el número de arcos que pasan por ese nodo. Aquellos nodos de grado uno que no sean el raíz se denominan hojas. Si se toman dos nodos u, v unidos por un arco tal que u esté en el camino de la raíz a v, entonces se dice que u es el nodo padre de v y v es un nodo hijo de u.

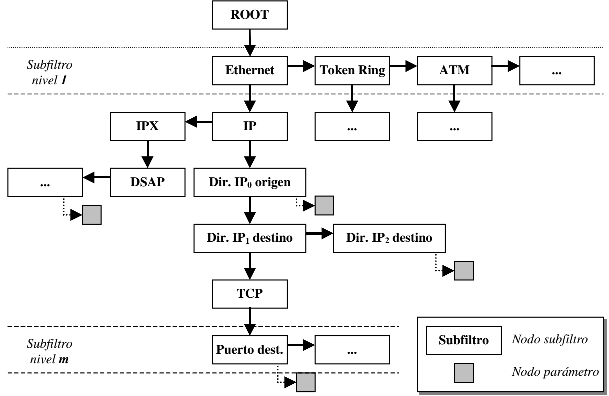


Figura 2.3: Ejemplo de árbol de filtrado PAM-Tree

Como se ha mostrado en la figura 2.3 dentro de la estructura en árbol de PAM-Tree existen dos tipos de nodos fundamentales: los nodos subfiltro y los nodos parámetro.

Los nodos subfiltro contienen información para analizar una condición sobre cada paquete. Si éste se verifica se actualizarán los nodos parámetro asociados y se avanzará por los nodos subfiltro hijo. Si no se verifica no se seguirá avanzando por esa rama del árbol.

Los nodos parámetro contienen un contador de determinado parámetro (bits, paquetes,...) que lleva cuenta de aquellos paquetes que han verificado todos los subfiltros anteriores. Serán nodos hoja del árbol e hijos del último subfiltro perteneciente al filtro al que se asocia el parámetro.

El proceso de filtrado en el PAM-Tree se convierte en encontrar el camino en la estructura de árbol que el paquete verifica. Para ello, los nodos poseen una relación de dependencia según las ramas del árbol que representan los filtros en sí. Como los subfiltros se aplican sobre campos de protocolos y estos se organizan de manera jerárquica la estructura en árbol aprovecha este hecho para disponer los nodos correspondientes a cada subfiltro.

Definición 6 *: Diremos que un nodo subfiltro A restringe a otro nodo subfiltro B si para analizar el nodo subfiltro B es necesario que el paquete satisfaga el nodo subfiltro A.*

Los subfiltros se colocan en orden descendente de manera que los subfiltros más restrictivos se chequeen en primer lugar. Estos subfiltros a su vez restringirán el avance a otros subfiltros inferiores en la jerarquía o harán que el paquete se descarte. Por tanto, el algoritmo genera una estructura en árbol en la que subfiltros colocados al mismo nivel no restringen a nodos vecinos y sí restringen a los nodos inferiores. Debido a la jerarquía natural de las pilas de cabeceras de protocolos (por ejemplo, un selector de protocolo a nivel de enlace restringe a cierto protocolo en el nivel de red) las ramas horizontales coinciden con las capas de protocolos conforme el

algoritmo evoluciona. Esta evolución tiene lugar conforme se definen nuevos filtros en el árbol de manera dinámica, siendo la evolución más natural hacia una estructura del árbol en capas que siga fielmente la estructura en capas de los protocolos que se encuentren en la red bajo análisis.

Así, un parámetro de monitorización viene representado en el árbol por una secuencia de nodos subfiltro y un nodo parámetro final. Empezando de la raíz, el paquete se testea frente a los nodos subfiltro del nivel. Si alguno se verifica, el proceso sigue con los nodos subfiltro hijos de la jerarquía del árbol. En caso de que no se verifiquen subfiltros, el paquete se descarta. El proceso sigue hasta que se han visitado todos los nodos subfiltro hijo de los subfiltros padre que se han verificado, actualizando a la vez los nodos parámetro asociados a cada nodo subfiltro.

En la figura 2.4 se presenta una sonda de monitorización en el que PAM-Tree se encarga de la parte más importante de procesado de los paquetes y actualización de los parámetros monitorizados, mientras que el servidor de monitorización se limita a muestrear periódicamente estos parámetros con un intervalo de muestreo determinado.

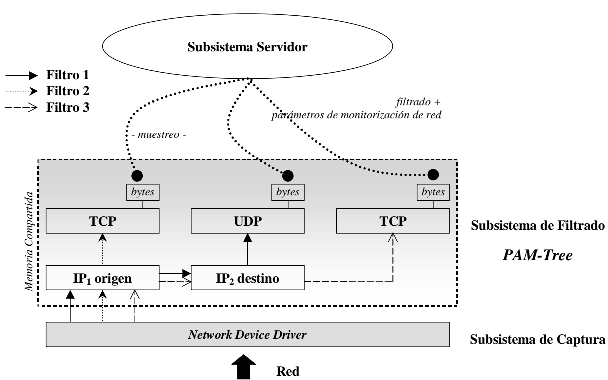


Figura 2.4: PAM-Tree y el sistema de monitorización

En esta figura 2.4 también se muestra la reutilización de subfiltros: en ella hay definidos 3 filtros, de los que nos interesan los bytes/sg:

- Filtro 1: dirección IP_1 origen - TCP
- Filtro 2: dirección IP_1 origen - dirección IP_2 destino - UDP
- Filtro 3: dirección IP_1 origen - dirección IP_2 destino - TCP

Se observa la reutilización de bloques de filtrado: el subfiltro "dirección IP_1 origen" que es común a los 3 filtros sólo se testea una única vez para los 3. Lo mismo ocurre con el subfiltro "dirección IP_2 destino" que también se testea una sola vez para los dos filtros de los que forma parte.

2.4.2 Nodos subfiltro

El nodo subfiltro es aquel que forma la jerarquía básica del árbol de filtrado. Como su propio nombre indica, representa y realiza la tarea de comprobación de una condición dentro de un filtro completo, es decir, se limita a testear la verificación de una condición sobre determinada secuencia de bits de un paquete.

La información principal que tendrá que contener un nodo subfiltro será la definición de la condición a verificar (F_i), y los nodos subfiltro hijos relacionados jerárquicamente para conformar un filtro. Además, pueden tener asociados nodos parámetro si se trata del último subfiltro de cierto filtro definido. La definición de la condición F_i (expresión 2.2) vendrá fijada por la secuencia de bits del paquete sobre la que se realiza el test E_{jk} (determinado por los *StartBit* y *StopBit* de la expresión 2.1), una función genérica *Function* que nos dé un valor intermedio (normalmente se limitará a sacar el valor decimal de los bits E_{jk}), un valor *Value* contra el que comparar el valor intermedio y otra función que nos diga el tipo de comparación *ComparisonType* a realizar entre el valor y el valor intermedio (igualdad, desigualdad, mayor que, menor que, etc.).

Además, si vamos a permitir que un mismo nodo subfiltro pueda ser compartido por diferentes filtrados y, por tanto, diferentes nodos parámetro finales, será necesario incorporar un contador que indique el número de filtros que lo están utilizando. A la hora de borrar un filtro, este contador servirá para eliminar sólo ese nodo subfiltro si no existen más filtros que lo utilicen. De este modo cuando en una inserción se reutilice el nodo subfiltro será necesario incrementar este contador y cuando en un borrado se deje de utilizar será necesario decrementar este contador.

El nodo subfiltro raíz es un nodo especial que no realiza ningún tipo de filtrado y que servirá para contabilizar todo el tráfico visto en la red y el total de filtros configurados.

Por tanto, un nodo subfiltro n vendrá determinado por:

$$n = (F_i, Used, Sons, Parameters) \tag{2.6}$$

Dentro de los nodos subfiltros del árbol se hace necesario implementar funciones especiales para adecuarse al comportamiento peculiar de los protocolos de red. La definición del subfiltro otorga gran flexibilidad para la realización de tareas más complejas en el nodo subfiltro como puede ser el soporte de niveles de cabeceras de protocolos, el reensamblado de paquetes o las tareas de autodescubrimiento de valores como se verá a continuación.

La variable global *Offset* de PAM-Tree se utiliza para saber en cada momento en qué posición del paquete comienza la cabecera de protocolo que se está procesando. De esta forma, los campos *StartBit* y *StopBit* de los nodos subfiltro se pueden definir con respecto al comienzo de esa cabecera de protocolo independientemente de dónde se encuentre esta cabecera en el paquete final leído de la red. Así se independizan los niveles de protocolos y se facilita la decodificación o desencapsulado de los mismos. Por tanto, para realizar el filtrado se tomarán los bits del paquete en la posición dada por la combinación del *Offset* y *StartBit-StopBit*.

Esta variable *Offset* (en bytes) comenzará con valor 0 cuando se inicie el recorrido por el árbol, y cuando se pase de un nivel de protocolo a otro (nueva cabecera) será necesario insertar

un nodo que se limite a incrementar esta variable *Offset* haciendo uso de una función *Function* modificada del nodo. De este modo ya estaremos en condiciones de continuar con el filtrado del siguiente nivel de protocolo. Si la cabecera es de tamaño fijo se incrementará en una constante. Por ejemplo, la cabecera Ethernet II tiene 14 bytes, y entonces habrá que incrementar el offset en 14 para pasar al siguiente nivel de protocolo. En cambio, si la cabecera es de tamaño variable, la función deberá ser capaz de deducir su longitud de la información de la cabecera actual. Por ejemplo, la cabecera IP puede tener diferentes tamaños según aparezcan campos de opciones y habrá que acudir al campo *Header Length* que dice el tamaño de la cabecera en palabras de 4 bytes (mínimo 20 bytes). De esta forma, mientras no se incremente la variable *Offset* los nodos que encontremos en el árbol se referirán a la misma cabecera de nivel de protocolo.

Un caso especial de gran utilidad al usar esta variable *Offset* es cuando estando en una cabecera de protocolo nos interese volver para atrás, y volver a filtrar desde el principio. Bastará con resetear la variable *Offset* a 0. Esto es aplicable por ejemplo al caso en que nos interese por ejemplo descubrir las direcciones IP de todas las máquinas que acceden al puerto de web: el filtrado tendría que ver primero que se trata de un paquete de web (comprobando que el campo puerto destino de la cabecera TCP sea 80), después hacer *Offset* 0 para volver al nivel inferior y seguir hasta la cabecera de protocolo IP donde se podría ya seleccionar el autodescubrimiento del campo dirección IP origen. Notar que TCP se encapsula por encima de IP y por tanto se hace necesario este reseteo de la variable *Offset*.

La fragmentación es un fenómeno habitual en redes de comunicaciones y aparece cuando una capa de protocolo de nivel inferior no puede transferir la PDU completa de una capa de protocolo de nivel superior. En estos casos, el nivel inferior tiene que dividir la información a transmitir en unidades más pequeñas que luego ya se recuperarán en destino. El fenómeno de dividir la información se denomina fragmentación y el de recuperarla se denomina reensamblado. Un caso típico es el de Ethernet y TCP/IP [81]. Cuando se quiere transmitir un paquete IP+TCP mayor que la MTU (Maximum Transmission Unit) de la Ethernet (que es de 1500 bytes sin contar la cabecera Ethernet que serían otros 14 bytes [82]), éste es fragmentado en origen y reensamblado en destino por el nivel IP para tener cabida en esa MTU. Los fragmentos tienen en su cabecera IP un identificador común que es único para cada trama TCP. Este campo, junto con los de offset de fragmento y longitud total, permitirán reensamblar en destino todos los paquetes con igual identificador [81, 10].

Un hecho importante es que sólo el primer fragmento contendrá la cabecera TCP. Por tanto, para que el sistema de filtrado funcione correctamente se hace necesario realizar el reensamblado y de esta forma poder asignar la misma cabecera TCP a todos los fragmentos. Sin embargo, este proceso tiene muchos problemas: los fragmentos pueden llegar desordenados o incluso no llegar. Para soportar la fragmentación se hace entonces necesario almacenar información en los nodos subfiltro. Para ello, cuando se quiera realizar un filtrado por encima de IP, el nodo que usando el campo *Protocol* de la cabecera IP identifica el nivel de transporte que encapsula tendrá que tratar a la vez esta fragmentación. Esto se consigue con una adecuación de la función *Function* del nodo, de modo que sea capaz de almacenar los fragmentos recibidos, para ordenarlos si es el caso y formar con ellos un sólo paquete que se pasará a los niveles superiores. Cada vez que se recibe un fragmento, se almacena y no se continua con el procesado del fragmento por esa rama del árbol. En el momento que se reciba el último

fragmento, se reensamblan y entonces sí se continua el procesado por esa rama del árbol. A partir de ahí el filtrado será normal, pero con el paquete resultante del reensamblado.

Se podrán llevar en un mismo nodo varios reensamblados en paralelo. Para indexarlos se hará uso de los campos de la cabecera IP que son la dirección IP origen y el identificador. Puede ocurrir que un fragmento no se reciba, e incluso puede ser que el fragmento perdido sea el primero. En tales casos será necesario descartar los fragmentos. Para ello, si se pasa por el nodo y queda algún reensamblado que no ha recibido fragmentos desde hace más de 15 s (según el estándar [81]) se descartan. En principio la fragmentación no suele ser muy elevada en una red por lo que se puede dejar este proceso de limpieza para la siguiente vez que se pase por el nodo sin desperdiciar muchos recursos. Si la fragmentación fuese elevada sería necesario implementar un recogedor de basuras para realizar esta labor periódicamente por todo el árbol.

En algunas ocasiones se está interesado en realizar el filtrado no sobre un campo del paquete directamente sino sobre otros parámetros como el tamaño del mismo, por ejemplo para obtener la distribución del tamaño de paquetes, o su timestamp, por ejemplo para obtener la distribución del tiempo entre paquetes. Para ello será suficiente con habilitar las funciones *Function* particulares que devuelvan en el primer caso el tamaño del paquete y en el segundo caso su timestamp.

Finalmente, la existencia del campo *ComparisonType* en cada nodo subfiltro deja abierta la posibilidad de definir comparaciones lógicas entre el valor intermedio dado por *Function* y el valor *Value* para implementar nuevas funcionalidades. Así, además de las típicas comparaciones de igualdad, desigualdad, etc. se pueden definir otras como el *autodescubrimiento* que consiste en que el nodo subfiltro genere un nuevo nodo subfiltro con el campo *Value* el valor del campo visto en el paquete cada vez que se vea un valor nuevo de ese campo. Esto es muy útil por ejemplo para la realización de matrices de tráfico, descubrir qué direcciones IP acceden a determinado puerto, etc. Esta funcionalidad, con cada nuevo valor del campo visto en un paquete crea un nuevo nodo subfiltro y el nodo parámetro asociado según el parámetro en que se esté interesado, por ejemplo, los bits por segundo de todas las máquinas que acceden a determinada dirección IP destino.

2.4.3 Nodos parámetro

El nodo parámetro vendrá asociado al último nodo subfiltro de un filtro o a cada uno de los nodo subfiltros finales de los componentes F_{OR_j} si el filtrado contiene concatenaciones OR de filtros. De este modo se llevará cuenta del valor total del parámetro o, si se trata de un OR de filtros, del valor parcial del que se podrá obtener el valor total.

El componente principal de este nodo será el contador que llevará cuenta del parámetro. Para ello, otro campo del nodo, *UpdateFunction*, indicará la forma en que se deba incrementar este contador cuando se verifique el nodo subfiltro al que se encuentra asociado. Normalmente se llevará cuenta del número de paquetes o de la suma de tamaños de paquetes que verifican el nodo subfiltro al que se encuentra asociado y, por tanto, que también verifican todos los nodos subfiltro en el camino desde la raíz del árbol.

De nuevo, si se permite que un mismo nodo parámetro se pueda utilizar para llevar cuenta de dos peticiones del mismo parámetro, será necesario incluir un contador que indique el número de veces que se esté utilizando el nodo, y se incremente y decremente en las operaciones de inserción y borrado de parámetros según corresponda.

Una estructura asociada a los nodos parámetro pero separada del árbol de filtrado es la que llamaremos estructura de acceso rápido *Params*. Se utiliza únicamente en el proceso de consulta de los parámetros para acceder de manera eficiente al contador del nodo parámetro correspondiente o si se trata de un filtro OR a los contadores de los nodos parámetro con los resultados parciales cuya suma dará el total requerido. La estructura se encuentra indexada por el identificador de parámetro que es único para cada parámetro insertado en el árbol y que será el que se utilice para la consulta o borrado del parámetro. Cada entrada de la estructura *Params* representa un parámetro insertado en el árbol y, además de los campos identificador y apuntadores a los contadores de los nodos parámetro correspondientes, almacenarán la definición del filtro al que se asocia el parámetro. Otro campo necesario es el valor de los contadores en la consulta anterior que se almacenará en *LastValues* para cada uno de los contadores apuntados. De este modo, el valor del parámetro será la suma de las diferencias de los contadores actuales con los de la consulta anterior. Así, el acceso al valor del parámetro de monitorización en el último intervalo es inmediato.

2.4.4 Concatenación de subfiltros

Los subfiltros se pueden unir entre sí mediante operaciones lógicas AND u OR para formar filtros asociados a parámetros. Para simplificar su implantación sobre el árbol hacemos uso de la descomposición vista en las expresiones 2.3 y 2.4 que transforman toda combinación de filtros AND y OR, en un OR de componentes OR, F_{OR_j}, que a su vez constan de un AND de condiciones F_i. Gracias a esta transformación se consigue facilitar la separación de las secuencias de condiciones comunes entre filtros. De esta forma, será posible reutilizar subfiltros entre componentes F_{OR} del mismo o diferentes filtros.

También se consigue incorporar el operador lógico OR entre condiciones sin perder la reutilización. Por una parte, las secuencias AND de subfiltros para formar un F_{OR_j} forman un camino en el árbol al que se asociará un nodo parámetro.

La asociación de componentes OR significa que un parámetro se debe actualizar si se verifica cualquiera de las componentes. Por tanto, el árbol de filtrado trata las condiciones por separado a la hora de incluirlas en la estructura del árbol. Al consultar el parámetro se debe tener en cuenta el valor acumulado para los componentes teniendo cuidado de no contabilizar el mismo paquete dos veces si resulta que verifica varios componentes OR de un mismo filtro. Por tanto, se deben sumar sus contadores respectivos teniendo alguna consideración previa.

Para ello se hará uso de una estructura intermedia, *Params*, que para cada identificador de parámetro tendrá referencias a todos los nodos parámetro a sumar para obtener el valor deseado. La función $UpdateFunction()$ de cada nodo parámetro será la encargada de incrementar los contadores de todos los nodos parámetro de los componentes verificados. Además esta función será la encargada de tener en cuenta si el paquete que se está filtrando ya ha verificado otro componente OR del mismo filtro y en tal caso incrementar el $LastValues$ aso-

ciado en la estructura $Params$ en el mismo valor, para que al final se contabilice el paquete una sola vez. Se podría definir una funcionalidad especial, aprovechándonos de este comportamiento, para contabilizar el paquete tantas veces como filtros OR verifica, pero éste no será el comportamiento habitual.

Supongamos por ejemplo que tenemos el filtro:

$$F = (F_1 \wedge (F_2 \vee F_3)) \vee (F_4 \wedge F_5) = (F_1 \wedge F_2) \vee (F_1 \wedge F_3) \vee (F_4 \wedge F_5) = F_{OR_1} \vee F_{OR_2} \vee F_{OR_3}$$

se tratarán los 3 componentes F_{OR_j} resultantes de manera independiente en la estructura, asociando a cada uno un nodo parámetro en el árbol. A la hora de actualizar los contadores, se incrementarán todos con el paquete recibido pero si el mismo paquete verifica más de un componente OR se incrementarán en igual medida los $LastValues$ asociados para contabilizar el paquete una sola vez. Cuando se solicite el resultado del parámetro habrá que sumar los valores de los tres nodos parámetro y restarles el valor de $LastValues$ de cada uno. Para ello, la variable $Params$ almacena para un identificador de parámetro referencias a los tres nodos parámetro con lo que es capaz de realizar la suma. A la vez, en cada nodo parámetro existe un campo $RParam$ que da acceso a la entrada $Params$ correspondiente, lo que permitirá llevar información en esta entrada (en el campo U) para saber si el paquete actual es el primero que verifica el filtro y asegurar así el correcto funcionamiento de $UpdateFunction$. Esta información podría ser el timestamp o un número de secuencia de los paquetes. En la figura 2.5 se puede observar la estructura resultante.

Así se facilita en gran medida la inserción y eliminación de filtros, se soporta la operación lógica OR de gran utilidad a la hora de definir filtros y se consigue mantener las secuencias AND de subfiltros lo que facilitará la reutilización de subfiltros comentada anteriormente, incluso cuando existan operaciones OR intermedias.

2.4.5 Operaciones en PAM-Tree

En una implementación de un sistema de monitorización, el subsistema de filtrado interacciona con el subsistema de captura filtrando los paquetes que recibe de éste. También recibe peticiones de inserción, borrado y consulta de parámetros desde el subsistema servidor. En nuestro caso, PAM-Tree implementa el subsistema de filtrado por lo que tendrá que soportar estas interacciones.

La inserción consiste en, recibida la definición de un parámetro Q formada por el filtro (secuencia de condiciones) y el *UpdateFunction* (qué contabilizar), insertar los nodos subfiltro y nodo parámetro correspondientes, creándolos si no existiesen o reutilizándolos en caso contrario. El proceso de inserción comienza por la raíz, y subfiltro a subfiltro se van pasando los nodos hasta llegar al último nodo subfiltro, al cual se le asocia el nodo parámetro final, repitiéndose el proceso en caso de varios filtros OR. A la vez se hace la entrada correspondiente en $Params$ que apunte al contador o contadores de los nodos parámetro creados. A partir de ahí el filtrado de ese parámetro es plenamente funcional, y los siguientes paquetes que se filtren actualizarán ese parámetro en caso de que verifiquen su filtro. El PAM-Tree devuelve un identificador del parámetro para futuras referencias. El proceso de inserción garantiza una de las características principales del algoritmo que es la reutilización de los bloques de

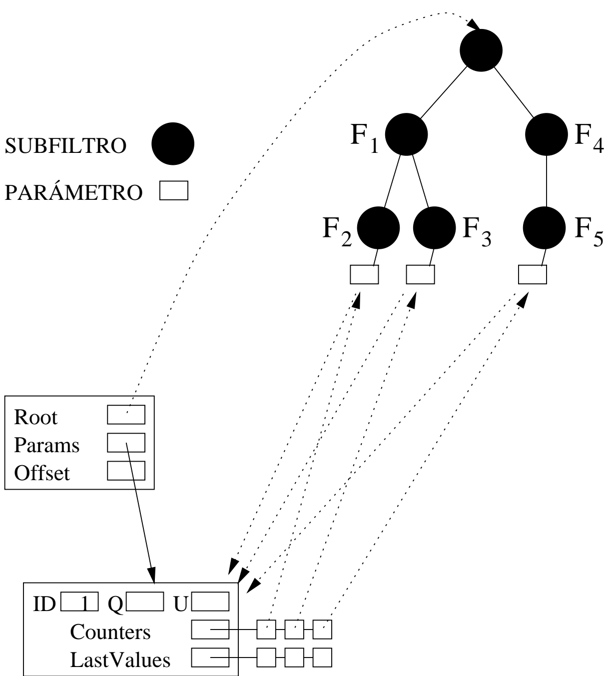


Figura 2.5: Ejemplo de estructura PAM-Tree para un filtrado AND/OR

subfiltrado: si dos filtros comparten subfiltros inferiores comunes, supondrá nodos subfiltro comunes también en el árbol.

El borrado se realiza a partir del identificador de parámetro. Con él se obtiene la definición del filtro Q de la estructura *Params* y de esta definición se obtiene de forma inmediata los nodos subfiltro a borrar del árbol. Así, siguiendo el camino por el árbol se realiza el borrado de nodos subfiltro o el decremento del contador de uso de los mismos avanzando nodo a nodo hasta el nodo parámetro. Finalmente se elimina la entrada de la estructura *Params*. Se devuelve el identificador del parámetro eliminado del árbol.

La consulta del valor de un parámetro es también inmediata a partir del identificador de parámetro. Con este identificador se tiene en *Params* un acceso directo al contador o contadores asociados al parámetro. El valor consultado será la suma de todos los contadores apuntados menos los valores de los mismos en la consulta anterior almacenados en *LastValue* para cada contador, teniendo en cuenta si dan la vuelta los contadores. La consulta devolverá el identificador de parámetro y su valor.

La parte central del algoritmo es el filtrado de paquetes. Partiendo de la raíz del árbol consiste en una inundación de los nodos subfiltro hijos de aquellos nodos subfiltro padre que el paquete verifica. Es decir, para cada paquete se realiza el siguiente proceso:

- **Primero.** Se parte del nodo raíz.

- **Segundo.** Se actualizan los nodos parámetro asociados (el nodo raíz no hace subfiltro, siempre se verifica).

- **Tercero.** Se testean los nodos subfiltro hijos y con cada uno de los nodos subfiltro hijos verificados se repite desde el paso segundo.

El proceso terminará cuando se hayan testeado todos los posibles nodos que el paquete pueda verificar, haciendo uso de la jerarquía en la definición de subfiltros que hará que se avance por las ramas de aquellos nodos subfiltro padre que se vayan verificando. A la vez se irán actualizando los nodos parámetro asociados a cada nodo subfiltro verificado. De nuevo aparece aquí otra de las características fundamentales de PAM-Tree que es la inclusión de los contadores de los parámetros de monitorización en la propia estructura de filtrado que simplifica el proceso de actualización de los mismos ya que se realiza a la vez que el filtrado.

PAM-Tree con estas operaciones ofrecerá las funcionalidades básicas de un subsistema de filtrado: inserción, borrado, actualización y consulta de parámetros. En la formalización que sigue, las cuatro operaciones se podrán llevar a cabo en paralelo (multihilo) al poderse alternar acciones de cada una de las operaciones, considerando atómicas la ejecución de estas acciones. Sin embargo, para poderse ejecutar una operación de un tipo deberá haber finalizado la anterior del mismo tipo.

2.5 Formalización del PAM-Tree

La descripción formal del algoritmo sigue la filosofía de la herramienta denominada Autómatas de Entrada/Salida [77]. Esta herramienta formal se usa ya que facilita la descripción del algoritmo, según un modelo de acciones con sus precondiciones y efectos, y de sus propiedades.

2.5.1 Estados del autómata PAM-Tree

En la figura 2.6 se muestran las variables y las estructuras del árbol de filtrado PAM-Tree para su modelización mediante el autómata PAM. La estructura jerárquica se compone de dos tipos de nodos, subfiltro y parámetro, cuyos campos se detallarán a continuación.

El nodo subfiltro (*Subfilter*) está compuesto por una condición F_i definida como $F_i = (E_{jk}, Value, Function, ComparisonType)$, el campo *Used* que contabiliza el número de parámetros que están usando este nodo (de este modo, un nodo subfiltro sólo se eliminará cuando no haya ningún parámetro que lo utilice), el campo *Sons* que es el conjunto de nodos subfiltro hijos en el árbol y el campo *Parameter* que es el conjunto de nodos parámetro asociados en el árbol.

El nodo parámetro (*Parameter*) está compuesto por el campo *Counter* que lleva cuenta del parámetro monitorizado, y el campo *UpdateFunction()* función que devuelve el valor a incrementar el contador según sea de bits, paquetes, autodescubrimiento, etc. o alguna función especial. También está compuesto por el campo *Used* que lleva cuenta del número de parámetros que están usando este nodo parámetro (o estructuras *Param* que están usando este nodo) y, finalmente, por el campo *RParam* que es un conjunto de referencias a entradas de *Params* que usan este nodo parámetro y permitirá dotar a la función *UpdateFunction()* de mayores funcionalidades.

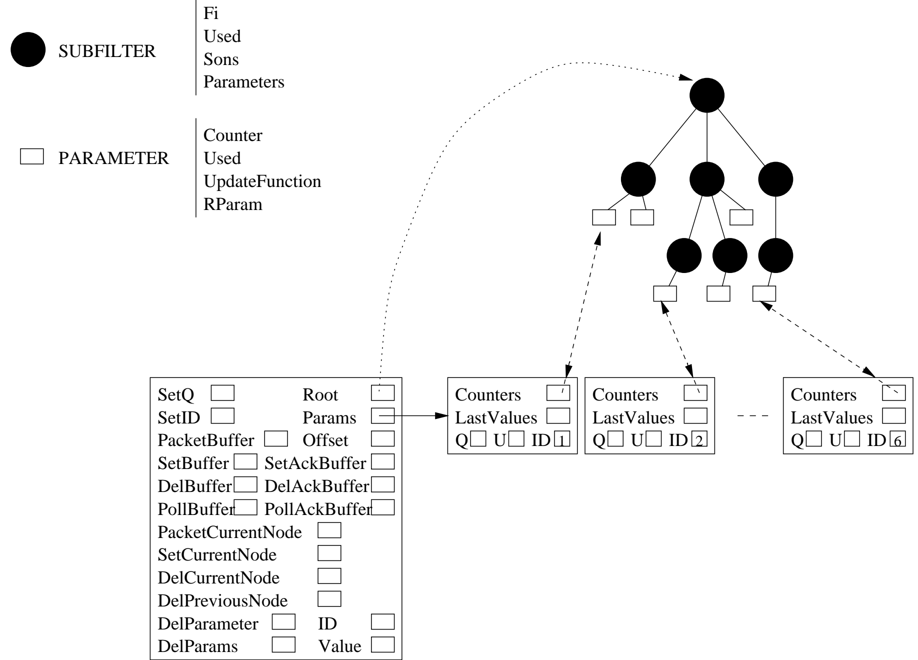


Figura 2.6: Variables globales y componentes de la estructura en árbol del autómata

Estos dos nodos subfiltro y parámetro serán los elementos que conformen el árbol que es la estructura de datos del algoritmo PAM-Tree.

La estructura *Params* permite un acceso rápido al valor de los parámetros indexados por el identificador de parámetro. A la vez permite implementar la operación lógica OR de subfiltros y lleva el valor de los parámetros en la última petición para poder así mandar la diferencia. Por cada parámetro establecido en el árbol, se tiene una entrada compuesta por *ID* identificador del parámetro, *Q* definición completa del filtro asociado, *U* campo de estado usado por *UpdateFunction()* para el control de los primeros paquetes, *Counters* acceso directo a los contadores de los nodos parámetro correspondientes (varios si tiene operaciones OR) y finalmente el campo *LastValues* que almacena el valor de los contadores en la última consulta para en la siguiente consulta devolver la diferencia.

El estado del autómata viene determinado por las variables que se describen a continuación:

- **Root**: nodo subfiltro raíz del árbol. Estado inicial: $Root.Sons = \emptyset$ (conjunto vacío), $Root.Parameters = \emptyset$, $Root.Used = 0$.

- **Params**: conjunto de elementos *Param* que apunta a los parámetros establecidos en el PAM-Tree, siendo una estructura de acceso rápido a los contadores asociados a cada nodo parámetro a partir de su identificador. Estado inicial: $Params = \emptyset$.

- **Offset**: para el recorrido del árbol en la acción interna *PacketSubfilter()*, guarda el offset del nivel de protocolo que se está procesando en la actualidad. Será un valor en bytes que permitirá que los valores de *StartBit* y *StopBit* del subfiltro se puedan definir con

relación a este offset, es decir, desde el comienzo de la cabecera del nivel de protocolo correspondiente. Estado inicial: $Offset = 0$.

- **PacketBuffer, SetBuffer, DelBuffer, PollBuffer**: buffers donde se almacenan los paquetes leídos de la red, las órdenes *SET* para definir un nuevo parámetro en el árbol, las órdenes *DEL* para eliminar un parámetro del árbol y órdenes *POLL* para consultar un parámetro del árbol, respectivamente. Inicialmente, todos estarán vacíos.

- **SetAckBuffer, DelAckBuffer, PollAckBuffer**: buffers para el envío del resultado de una petición *SET* que es el identificador del parámetro insertado, de una petición *DEL* que es el identificador del parámetro eliminado y de una petición *POLL* que es el identificador del parámetro consultado y su valor. Inicialmente no contienen mensajes.

- **PacketCurrentNode**: conjunto de nodos pendientes de revisar en acciones de procesado de paquetes *PacketSubfilter()*. En concreto, contiene los nodos subfiltro que se procesarán en las siguientes iteraciones para actualizar parámetros. Le corresponderá un estado inicial: $PacketCurrentNode = \emptyset$.

- **SetCurrentNode, DelCurrentNode**: nodo subfiltro actual en acciones *SetParam()* mientras se establece un parámetro y en acciones *DelParam()* mientras se borra un parámetro, respectivamente. Estado inicial para todos: $\emptyset$

- **SetQ**: copia de la definición del parámetro (filtro+*UpdateFunction*) que se está insertando en este momento. Se utiliza para saber en las acciones de inserción qué subfiltro toca insertar, ya que se van eliminando los ya insertados.

- **SetID**: se encarga de almacenar el identificador de parámetro que se está insertando en el árbol, mientras se insertan los diferentes componentes OR y sus subfiltros correspondientes en diferentes acciones, durante la operación de inserción. Así se es capaz de asociar todos los nodos parámetro insertados, correspondientes a cada componente OR, con la misma entrada en la estructura *Params*.

- **DelPreviousNode**: variable auxiliar utilizada únicamente en la acción *SetSubfilter()* que apunta al nodo subfiltro anterior en el recorrido por el árbol facilitando el proceso de borrado.

- **DelParameter**: variable auxiliar para borrado de un nodo parámetro en la acción *DelParameter()*.

- **DelParams**: variable auxiliar para borrado de una entrada en la estructura *Params* en la acción *DelParameter()*.

- **ID**: variable auxiliar que contiene el identificador único asignado al parámetro durante la acción *SetParameter()*.

- **Value**: variable auxiliar donde se acumulan los valores de los contadores que pertenecen al mismo parámetro en la acción *PollParameter()* para devolver el valor solicitado.

2.5.2 Signatura de acciones del autómata

Para describir el comportamiento del algoritmo propuesto en términos de formalismo de autómatas de Entrada/Salida, debemos proveer el conjunto de acciones del sistema. Dichas acciones describen las interacciones con el entorno y la función interna de filtrado del sistema.

En la figura 2.7 se muestran las acciones de entrada, de salida e internas del autómata PAM que se detallan a continuación.

in(PAM) = {**SetReq(Q)**, **DelReq(ID)**, **PollReq(ID)**, **Packet(P)**}
out(PAM) = {**SetResult(ID)**, **DelResult(ID)**, **PollResult(ID,Value)**}
int(PAM) = {**SetStart()**, **SetSubfilter()**, **SetParameter()**, **DelStart()**, **DelSubfilter()**, **DelParameter()**, **PollParameter()**, **PacketStart()**, **PacketSubfilter()**}

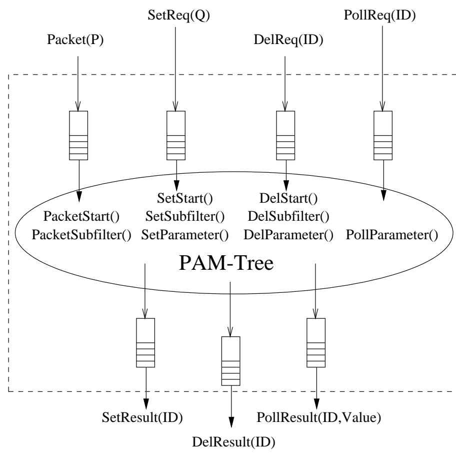


Figura 2.7: Autómata y sus acciones

2.5.3 Transiciones del autómata

Las siguientes acciones se describen usando la estructura precondición/efecto. En cada estado una acción se encuentra habilitada cuando se verifica su precondición. Con la ejecución de una acción las variables de estado se modifican de acuerdo a los efectos de la acción ejecutada.

De forma genérica se denota con μ a listas de elementos y el tipo de los elementos se indica en el subíndice. Por ejemplo, μ_{ID} denota una lista de IDs. La cadena vacía se representará con ε.

Las acciones se ejecutan de manera indivisible, pero será posible que acciones de distintos procesos de inserción, borrado, consulta y procesado del paquete se puedan mezclar, sin ser necesario que se ejecute cada proceso de manera indivisible. A continuación se presentan las acciones y en un apartado posterior se explican detalladamente.

2.5.3.1 Acciones de entrada

SetReq(Q)
effects: $SetBuffer \leftarrow SetBuffer \bullet Q;$

DelReq(ID)
effects: $DelBuffer \leftarrow DelBuffer \bullet ID;$

PollReq(ID)
effects: $PollBuffer \leftarrow PollBuffer \bullet ID;$

Packet(P)
effects: $PacketBuffer \leftarrow PacketBuffer \bullet P;$

2.5.3.2 Acciones de salida

SetResult(ID)
preconditions: $SetAckBuffer = ID \bullet \mu_{ID}$
effects: $SetAckBuffer \leftarrow \mu_{ID};$

DelResult(ID)
preconditions: $DelAckBuffer = ID \bullet \mu_{ID}$
effects: $DelAckBuffer \leftarrow \mu_{ID};$

PollResult(ID,Value)
preconditions: $PollAckBuffer = (ID, Value) \bullet \mu_{(ID,Value)}$
effects: $PollAckBuffer \leftarrow \mu_{(ID,Value)};$

2.5.3.3 Acciones internas

SetStart()
preconditions: $SetBuffer = Q \bullet \mu_Q \wedge SetCurrentNode = \emptyset$
effects: $SetCurrentNode \leftarrow Root;$
$SetQ \leftarrow Q;$
$SetID \leftarrow 0;$

SetSubfilter()
preconditions: $SetQ.F = F_{OR_j} \bullet \mu_{F_{OR_j}} \wedge F_{OR_j} = Fi \bullet \mu_{Fi} \wedge SetCurrentNode \neq \emptyset \wedge SetBuffer = Q \bullet \mu_Q$
effects: $IF\ (Fi \notin SetCurrentNode.Sons)\ THEN$
$SetCurrentNode.Sons \leftarrow SetCurrentNode.Sons \cup NewSubfilter(Fi);$
$ENDIF$
$SetCurrentNode \leftarrow SetCurrentNode.Sons(Fi);$
$SetCurrentNode.Used \leftarrow SetCurrentNode.Used + 1;$
$F_{OR_j} \leftarrow \mu_{Fi};$

SetParameter()
preconditions: $SetQ.F = F_{OR_j} \bullet \mu_{F_{OR_j}} \wedge F_{OR_j} = \varepsilon \wedge SetCurrentNode \neq \emptyset \wedge SetBuffer = Q \bullet \mu_Q$
effects: $IF\ (SetQ.UpdateFunction \notin SetCurrentNode.Parameters)\ THEN$

$SetCurrentNode.Parameters \leftarrow SetCurrentNode.Parameters \cup NewParameter(SetQ.UpdateFunction)$;
$ELSE$
$SetCurrentNode.Parameters(SetQ.UpdateFunction).Used \leftarrow \ldots$
$\ldots SetCurrentNode.Parameters(SetQ.UpdateFunction).Used + 1$;
$ENDIF$
$IF\ (SetID = 0)\ THEN$
$SetID \leftarrow NewID()$;
$Params \leftarrow Params \cup \ldots$
$\ldots NewParam(SetID, Q, SetCurrentNode.Parameter(SetQ.UpdateFunction).Counter)$;
$ELSE$ /* Es un OR de filtros, se añade */
$Params(SetID).Counters \leftarrow Params(SetID).Counters \bullet \ldots$
$\ldots SetCurrentNodeParameter(SetQ.UpdateFunction).Counter$;
$SetCurrentNodeParameter(SetQ.UpdateFunction).RParam \leftarrow \ldots$
$\ldots SetCurrentNodeParameter(SetQ.UpdateFunction).RParam \cup Params(SetID)$;
$Params(SetID).LastValues \leftarrow Params(SetID).LastValues \bullet 0$;
$ENDIF$
$SetQ.F = \mu_{F_{OR_j}}$;
/* Si hay más filtros OR */
$IF\ (SetQ.F \neq \varepsilon)\ THEN$
$SetCurrentNode \leftarrow Root$;
$ELSE$
$SetBuffer \leftarrow \mu_Q$; /* Terminada inserción, reinicializa variables */
$SetCurrentNode \leftarrow \emptyset$;
$Root.Used \leftarrow Root.Used + 1$; /* Llevará cuenta del número de parámetros insertados en el árbol */
$SetAckBuffer \leftarrow SetAckBuffer \bullet SetID$;
$ENDIF$

DelStart()
preconditions: $DelCurrentNode = \emptyset \wedge DelBuffer = ID \bullet \mu_{ID}$
effects: $IF\ (ID \notin Params)\ THEN$ /* No existe tal ID en el árbol, no hace nada */
$DelAckBuffer \leftarrow DelAckBuffer \bullet ID$;
$DelBuffer \leftarrow \mu_{ID}$;
$ELSE$
$DelCurrentNode \leftarrow Root$;
$Root.Used \leftarrow Root.Used - 1$;
$Params(ID).Counters = \emptyset$;
$ENDIF$

DelSubfilter()
preconditions: $DelCurrentNode \neq \emptyset \wedge DelBuffer = ID \bullet \mu_{ID} \wedge Params(ID).Q.F = F_{OR_j} \bullet \mu_{F_{OR_j}} \wedge F_{OR_j} = Fi \bullet \mu_{Fi}$
effects: $F_{OR_j} \leftarrow \mu_{Fi}$;
$DelPreviousNode \leftarrow DelCurrentNode.Sons(Fi)$;
$IF\ (DelPreviousNode.Used = 1)\ THEN$
$DelCurrentNode.Sons \leftarrow DelCurrentNode.Sons - \{DelPreviousNode\}$;
$DelCurrentNode \leftarrow DelPreviousNode$;
$FreeSubfilter(DelPreviousNode)$;
$ELSE$
$DelPreviousNode.Used \leftarrow DelPreviousNode.Used - 1$;
$DelCurrentNode \leftarrow DelPreviousNode$;
$ENDIF$

DelParameter()
preconditions: $DelCurrentNode \neq \emptyset \wedge DelBuffer = ID \bullet \mu_{ID} \wedge Params(ID).Q.F = F_{OR_j} \bullet \mu_{F_{OR_j}} \wedge F_{OR_j} = \varepsilon$
effects: $DelParameter \leftarrow DelCurrentNode.Parameters(Params(ID).Q.UpdateFunction)$;
$IF\ (DelParameter.Used = 1)\ THEN$
$DelCurrentNode.Parameters \leftarrow DelCurrentNode.Parameters - \{DelParameter\}$;
$FreeParameter(DelParameter)$;
$ELSE$
$DelParameter.Used \leftarrow DelParameter.Used - 1$;
$DelParameter.RParam \leftarrow DelParameter.RParam - \{Params(ID)\}$;
$ENDIF$
/* Si hay más filtros OR a borrar */

$IF\ (Params(ID).Q.F \neq \emptyset)\ THEN$
$\quad DelCurrentNode \leftarrow Root;$
$ELSE$
$\quad DelParams \leftarrow Params(ID);$
$\quad Params \leftarrow Params - \{DelParams\};$
$\quad FreeParam(DelParams);$
$\quad DelBuffer \leftarrow \mu_{ID};$ /* Terminado borrado, reinicializa variables */
$\quad DelCurrentNode \leftarrow \emptyset;$
$\quad DelAckBuffer \leftarrow DelAckBuffer \bullet ID;$
$ENDIF$

PollParameter()
preconditions: $PollBuffer = ID \bullet \mu_{ID}$
effects: $PollBuffer \leftarrow \mu_{ID};$
$IF\ \ ((ID \notin Params) \vee (Params(ID).Counters = \emptyset))\ \ THEN$
$\quad PollAckBuffer \leftarrow PollAckBuffer \bullet (ID, -1);$
$ELSE$
$\quad Value \leftarrow 0;$
$\quad FOR\ \ i = 1 : length(Params(ID).Counters),$
$\quad\quad Value \leftarrow Value + (Params(ID).Counters[i] - Params(ID).LastValues[i]);$
$\quad\quad Params(ID).LastValues[i] \leftarrow Params(ID).Counters[i];$
$\quad ENDFOR$
$ENDIF$
$\quad PollAckBuffer \leftarrow PollAckBuffer \bullet (ID, Value);$

PacketStart()
preconditions: $PacketBuffer = P \bullet \mu_P\ \wedge\ PacketCurrentNode = \emptyset$
effects: $PacketCurrentNode \leftarrow Root;$
$Offset \leftarrow 0;$

PacketSubfilter()
preconditions: $PacketBuffer = P \bullet \mu_P\ \wedge\ current \in PacketCurrentNode$
effects: /* Actualizar parámetros asociados al nodo */
$FOR\ all\ (parameter \in current.Parameters)$
$\quad parameter.Counter \leftarrow parameter.Counter + parameter.UpdateFunction(Offset, P, parameter.RParam);$
$ENDFOR$
/* Se recorren todos los nodos hijos almacenando en $PacketCurrentNode$ los nodos que se verifican */
$FOR\ all\ (node \in current.Sons)$
$\quad IF\ Test(node.Fi, Offset, packet)\ THEN$
$\quad\quad PacketCurrentNode \leftarrow PacketCurrentNode \cup \{node\};$
$\quad ENDIF$
$ENDFOR$
$PacketCurrentNode \leftarrow PacketCurrentNode - \{current\};$
/* Después del proceso si no hay nodos en PacketCurrentNode es que se ha terminado de procesar el paquete */
$IF\ (PacketCurrentNode = \emptyset)\ THEN$
$\quad PacketBuffer \leftarrow \mu_{packet};$
$ENDIF$

2.5.4 Descripción de transiciones

Las acciones de entrada se encuentran siempre habilitadas. Es decir, se pueden ejecutar en cualquiera de los estados de las ejecuciones. Esto significa que los procesos externos pueden interaccionar con el sistema de filtrado en cualquier momento mandando mensajes. De la misma forma, se puede recibir un paquete en cualquier estado de la ejecución. Esto se modela con las acciones **SetReq(Q)**, **DelReq(ID)**, **PollReq(ID)** y **Packet(P)**. Los efectos de estas acciones son las de almacenar la definición del parámetro, el identificador de parámetro o el

paquete en el correspondiente buffer. Estos buffers se consideran de tamaño infinito. Dichas acciones modelan el interfaz que recoge todos los mensajes de petición mandados por los usuarios y todos los paquetes capturados del enlace de comunicaciones.

- **SetReq(Q)**:
  Esta acción modela la petición de inserción de un parámetro en el PAM-Tree. Con su ejecución se almacena en el buffer *SetBuffer* la definición del parámetro Q, siendo $Q = (F, UpdateFunction)$ donde $F = F_{OR_1} \vee F_{OR_2} \vee \ldots \vee F_{OR_j}$, $F_{OR_j} = F_1^j \wedge F_2^j \wedge \ldots \wedge F_{n_j}^j$ y $F_i^j = (E_{kl}, Value, Function, ComparisonType)$, a insertar en el PAM-Tree.

- **DelReq(ID)**:
  Esta acción indica la solicitud de petición de eliminación de un parámetro en el PAM-Tree. Su ejecución almacena en el buffer *DelBuffer* el identificador del parámetro *ID* a eliminar .

- **PollReq(ID)**:
  Con esta acción se modela la consulta de un parámetro en el PAM-Tree. Al ejecutarse almacena en el buffer *PollBuffer* el identificador del parámetro *ID* del que se quiere consultar el valor.

- **Packet(P)**:
  Ante la existencia de un nuevo paquete P en la red, esta acción almacena en el buffer *PacketBuffer* el paquete procedente de la red monitorizada y que se desea filtrar en el PAM-Tree.

Las acciones de salida se habilitan cuando una de las funciones del sistema de monitorización haya sido completada. **SetResult(ID)** indica que un nuevo parámetro se ha añadido a la estructura. Cuando un parámetro se elimina de la estructura se habilita la acción **DelResult(ID)** y el resultado de la consulta de un parámetro de monitorización se modela con la ejecución de la acción **PollResult(ID,Value)**.

- **SetResult(ID)**:
  Una vez procesada una petición de inserción de parámetro, se habilita esta acción al haber en el *SetAckBuffer* un identificador *ID* del nuevo parámetro en el árbol. Esto habilita la acción, cuya ejecución elimina el *ID* del *SetAckBuffer*.

- **DelResult(ID)**:
  Una vez procesada una petición de eliminación de parámetro, se habilita esta acción al haber en el *DelAckBuffer* un identificador *ID* del parámetro eliminado. Esto hará que se habilite una acción en el entorno **DelResult(ID)** y se eliminará el *ID* del *DelAckBuffer*.

- **PollResult(ID,Value)**:
  Una vez procesada una petición de consulta de parámetro, se habilita esta acción al haber en el *PollAckBuffer* una tupla formada por el identificador *ID* del parámetro consultado y su valor *Value*. Esto hará que se habilite una acción en el entorno **PollResult(ID,Value)** y se eliminará la tupla *(ID, Value)* del *PollAckBuffer*.

Las funcionalidades del algoritmo PAM-Tree vienen descritas por las acciones internas, que se dividen en cuatro grupos: inserción, borrado, consulta y procesado de paquete.

La inserción de parámetros en el árbol se realiza mediante las acciones **SetStart()**, **SetSubfilter()** y **SetParameter()**. La acción **SetStart()** inicializa la inserción desde la raíz. La acción **SetSubfilter()** inserta los nodos subfiltro necesarios o reutiliza los ya existentes. La acción **SetParameter()** inserta el nodo parámetro final y la entrada en la lista *Params*.

- **SetStart()**:
  Esta acción se habilita cuando existe un parámetro a insertar Q en *SetBuffer* y todavía no se ha comenzado la inserción. Inicializa el recorrido a la raíz del árbol e inicializa las variables *SetQ* y *SetID* a la definición del parámetro Q y 0 respectivamente. El *SetID=0* representa la ausencia de identificador asignado al parámetro a establecer Q.

- **SetSubfilter()**:
  Esta acción se habilita cuando existen nodos subfiltro por insertar, asociados a cada condición F_i de cada componente OR (F_{OR}) del filtro. Si ese nodo subfiltro no existe en el árbol, lo crea (mediante la función **NewSubfilter(F_i)**) y lo inserta como nodo hijo del nodo anterior. En cambio, si ya existía, incrementa su contador de reutilización. Esta acción se habilita tantas veces como subfiltros haya por insertar para cada F_{OR} de la lista.

- **SetParameter()**:
  Se habilita tras insertarse el último nodo subfiltro de cada componente OR de un filtro, y se encarga de la inserción del nodo parámetro. Si el nodo parámetro no existe en el árbol, lo crea (mediante la función **NewParameter(UpdateFunction)**) y lo inserta como parámetro del nodo subfiltro actual. En cambio, si ya existiera, incrementa su contador de reutilización. La variable *SetID* dirá si es el primer nodo parámetro insertado para el parámetro de monitorización Q (cuando valga 0) en cuyo caso se creará un nuevo identificador único para el parámetro insertado mediante la función **NewID()**. Por otra parte se crea una nueva entrada en la estructura *Params*, mediante la función **NewParam(ID,Q,Counter)**, que relaciona todos los nodos parámetro asociados al parámetro de monitorización insertado. Esta acción se repetirá tantas veces como componentes OR tenga el parámetro insertado. Todos los nodos parámetro creados comparten el mismo *ID* y la misma entrada en la estructura *Params*. Tras insertar el último subfiltro del último componente OR se añade el *ID* en el *SetAckBuffer* para habilitar la acción de respuesta.

La eliminación de un parámetro en el árbol se realiza mediante las acciones **DelStart()**, **DelSubfilter()** y **DelParameter()**. Estas acciones realizan las tareas opuestas a las descritas para la inserción, eliminando los nodos subfiltro, nodos parámetro y entradas de la lista *Params* innecesarios.

- **DelStart()**:
  Esta acción se habilita cuando existe un identificador a eliminar *ID* en *DelBuffer* y todavía no se ha comenzado el borrado. Si tal *ID* no existe en *Params* se elimina de

DelBuffer y se coloca el *ID* en *DelAckBuffer* para habilitar el mensaje de respuesta. Si existe, se inicializa el recorrido a la raíz del árbol y se elimina el campo *Counters* de su entrada en Params para no tener acceso a la consulta del parámetro de monitorización a pesar de que existan sus subfiltros asociados mientras dure el proceso de borrado.

- **DelSubfilter()**:
  Esta acción se habilita cuando existen nodos subfiltro por eliminar (F_i) en los componentes OR del primer filtro de *SetQ.F* y ya se ha inicializado el recorrido. Si el nodo subfiltro correspondiente no está reutilizado se elimina mediante la función **FreeSubfilter(subfilternode)** y si lo estuviera se decrementa su contador de reutilización.

- **DelParameter()**:
  Se habilita tras eliminarse el último nodo subfiltro de cada componente OR de un filtro, y se encarga de la eliminación del nodo parámetro. Si el nodo parámetro correspondiente no está reutilizado se elimina mediante la función **FreeParameter(parameternode)** y si lo estuviera se decrementa su contador de reutilización. Cuando no queden más filtros OR por eliminar, suprime la entrada correspondiente a este *ID* en la estructura *Params* mediante la función **FreeParam(param)** y añade el *ID* al *DelAckBuffer* para habilitar la acción de respuesta.

La consulta del valor de un parámetro la realiza la acción **PollParameter()**. Con la ejecución de esta acción, se accede de manera directa a los contadores del árbol asociados al identificador de parámetro cuyo valor se ha solicitado.

- **PollParameter()**:
  Esta acción se habilita cuando existe un identificador a consultar *ID* en *PollBuffer*. Si tal *ID* no existe, añade (*ID,-1*) al *PollAckBuffer* para indicar que el parámetro no existe. En caso contrario, accede a los contadores asociados gracias al campo *Counters* de la estructura *Params* y restándoles los valores de la consulta anterior almacenados en *LastValues*, calcula el valor total del parámetro que junto con el *ID* forma una tupla que se añade a *PollAckBuffer* para habilitar la acción de respuesta.

Finalmente, el filtrado de los paquetes recibidos por la red se realiza mediante las acciones **PacketStart()** y **PacketSubfilter()**.

- **PacketStart()**:
  Esta acción se habilita cuando existe un paquete en *PacketBuffer* y todavía no se ha comenzado el filtrado. Inicializa el recorrido a la raíz del árbol, colocando *Root* como siguiente nodo a revisar. También inicializa *Offset* a 0 para comenzar por el filtrado de cabeceras de protocolos de nivel más bajo.

- **PacketSubfilter()**:
  Se habilita con la existencia de nodos subfiltro por verificar en *PacketCurrentNode*. La acción realiza la función de filtrado del algoritmo. Para cada paquete, esta acción comprueba si se verifican los nodos subfiltro comenzando desde el nodo raíz. Si se

ejecuta sobre un nodo quiere decir que se verifica, por lo que en primer lugar actualiza los parámetros asociados mediante *UpdateFunction()*. Entonces evalúa todos los nodos hijo y añade a *PacketCurrentNode* aquellos que se verifican para su procesado en las siguientes iteraciones. Se comprueba si se verifican los nodos subfiltro usando la función **Test(F_i,Offset,packet)** que comprueba el subfiltro pasado F_i con respecto al offset. Si al final de la acción no existen nodos en *PacketCurrentNode* por revisar, indica que se ha terminado el filtrado por lo que se elimina el paquete de *PacketBuffer*. En esta acción se observa que si existen parámetros que comparten subfiltros, se actualizan a la vez sin tener que repetir subfiltrados ya realizados.

2.5.5 Propiedades del sistema de filtrado PAM-Tree

En el presente apartado se van a presentar una serie de propiedades del algoritmo de filtrado PAM-Tree. Entre estas propiedades destacan la de estructura jerárquica del árbol y la propiedad de reutilización de bloques de filtrado, dos de las principales propiedades del algoritmo de filtrado presentado en este trabajo.

Para ello, se van a presentar una serie de propiedades intermedias sobre la inserción de parámetros, de la existencia de parámetros, de la unicidad de identificadores para cada parámetro y de la estructura en árbol desde el nodo raíz. Todo ello ayudará a una mejor comprensión del funcionamiento del algoritmo de filtrado PAM-Tree.

El funcionamiento del sistema de filtrado se modela por sus ejecuciones, las cuales corresponden a una secuencia de estados a los que se llega por la ejecución de las acciones del sistema. Con s_k se representa el estado k-ésimo y con π_k se representa la acción que lleva al sistema del estado s_k al s_{k+1}.

A la hora de probar las características del sistema se parte de la presencia de los parámetros en el árbol. El parámetro se considera en el árbol siempre que pueda consultarse su valor. Las consultas se pueden realizar siempre que desde la estructura *Params* el parámetro tenga referencia a los nodos parámetro donde se almacenan los valores del parámetro de monitorización. La operación de borrado utiliza la variable $Counters$ para indicar que se comienza el proceso de borrado y que ese parámetro no está disponible para operaciones de consulta. Por tanto, consideraremos que ese parámetro ya no existe sobre el árbol en el momento que comienza su proceso de borrado.

Propiedad 1 *: Sea s_k un estado alcanzable del sistema, $G \in \mathcal{Q}$ un parámetro del árbol. Si $s_k.Params(ID_G).Counters = \emptyset$ el parámetro no existe en el árbol.*

Demostración: La variable $Params(ID_G).Counters$ toma el valor $\emptyset$ sólo en la acción $DelStart()$ que es la acción de comienzo de borrado del parámetro y por tanto no puede consultarse el valor de sus nodos parámetro nunca más. ∎

Para que exista un parámetro sobre el árbol de filtrado se han tenido que ejecutar una serie de acciones. La siguiente propiedad determina estas acciones que, aunque se puedan intercalar

con otras acciones de otros procesos que no sean de inserción, se ejecutan en un orden concreto. Es decir, se muestra que la presencia de un parámetro en el sistema supone un orden en la ejecución de las acciones para llevar a cabo determinada operación. En concreto, para cada condición de un filtro se inserta un nodo subfiltro (o se reutiliza si se diera el caso) y cada componente OR se traduce en la inserción de un nodo parámetro tras el último nodo subfiltro asociado a ese componente.

Por tanto, una inserción comenzará con un **SetReq()** y le seguirá un **SetStart()**. Por cada condición de un componente OR se ejecutará la acción **SetSubfilter()** que insertará un nodo subfiltro y finalmente, tras insertarse los nodos subfiltro correspondientes a ese componente OR, se ejecutará una acción **SetParameter()** por cada componente OR.

Propiedad 2 : *Sea s_k un estado alcanzable del sistema y $G \in \mathcal{Q}$ un parámetro del árbol siendo $G.F = \bigvee_{j=1\ldots m}(\bigwedge_{i=1\ldots n_j} F_i)_{OR_j}$ y sea $ID_G \in \mathcal{ID}$. Entonces, $\exists k'' < k' < k$ verificando que:*

2.1: $\pi_{k'} = SetStart(G)$

2.2: $\pi_{k''} = SetReq(G)$

2.3: $\forall F_i \in F_{OR_j}$, $\exists k_i^j$ *tal que* $\pi_{k_i^j} = SetSubfilter(G, F_{OR_j}, F_i)$ *con* $j = 1\ldots m$, $i = 1\ldots n_j$, $k' < k_i^j < k$

2.4: $\forall F_{OR_j}$, $\exists k^j$ *tal que* $\pi_{k^j} = SetParameter(G, F_{OR_j})$ *con* $j = 1\ldots m$, , $k' < k^j < k$

Demostración: La demostración se realiza por reducción al absurdo. Se tienen en cuenta únicamente las acciones relacionadas con el establecimiento de parámetros ya que el resto de acciones invalidan el antecedente de la propiedad o no le afectan.

2.1: Si $\nexists \pi_{k'} = SetStart(G)$, es decir, no se ejecuta esa acción, luego $\forall s_k$, $s_k.SetQ \neq G$ y por tanto no se habilita $SetParameter()$ y no hay ID_G asociado, lo que contradice la premisa de la propiedad.

2.2: Si $\nexists \pi_{k''} = SetReq(G)$ quiere decir que no se habilita un $k', k'' < k'$, tal que $\pi_{k'} = SetStart(G)$ lo que contradice la propiedad 2.1 y con ello se demuestra la propiedad.

2.3: Si $\nexists \pi_{k_i^j} = SetSubfilter(G, F_{OR_j}, F_i)$ es porque no se han habilitado anteriormente otros $SetSubfilter(G, F_{OR_m}, F_i)$, $m = 1\ldots j$, y por tanto tampoco existe un $k', k' < k_i^j$, tal que $\pi_{k'} = SetStart(G)$ lo que contradice la propiedad 2.1.

2.4: Si $\nexists \pi_{k^j} = SetParameter(G, F_{OR_j})$ quiere decir que no se habilita en ningún momento y esto se deberá a que $F_{OR_j} \neq \varepsilon$ lo cual ocurrirá si no se ejecuta la acción $SetSubfilter(G, F_{OR_j}, F_i)$ que contradice la propiedad 2.3. ∎

A partir de la propiedad anterior y debido a las condiciones asociadas a cada filtro de cada parámetro, es sencillo comprobar que esto supone la inserción de manera ordenada de

los nodos subfiltro y nodos parámetro en el árbol. El siguiente corolario determina el orden de las acciones de la propiedad 2.

Corolario 1 : $\forall F_i \in F_{OR_j}$, $\exists k_i^j, j = 1 \dots m, i = 1 \dots n_j$ *tal que* $k_1^1 < k_2^1 < \cdots < k_{n_1}^1 < k^1 < k_1^2 < \cdots < k_{n_2}^2 < k^2 < \cdots < k_1^m < \cdots < k_{n_m}^m < k^m < k$ *siendo* $\pi_{k_i^j} = SetSubfilter(G, F_{OR_j}, F_i)$ *y* $\pi_{k^j} = SetParameter(G, F_{OR_j})$.

Demostración:

Este corolario es evidente sin más que tener en cuenta la propiedad 2 y que los componentes F_{OR_j} y sus condiciones F_i asociadas se insertan en el árbol con un orden FIFO. ■

A partir de la propiedad 2 es fácil comprobar que al insertar la primera de las condiciones de cada componente OR de un filtro, se comienza siempre a partir del nodo *Root* del sistema de filtrado.

Corolario 2 : *Sea* s_k *un estado alcanzable del sistema y* $G \in \mathcal{Q} \wedge ID_G \in \mathcal{ID}$ *tal que* $s_{k-1}.SetQ.F = (F_1 \bullet F_2 \bullet \cdots \bullet F_n)_{OR} \bullet \mu_{F_{OR}} \wedge s_k.SetQ.F = (F_2 \bullet \cdots \bullet F_n)_{OR} \bullet \mu_{F_{OR}}$. *Entonces, se cumple que* $s_k.SetCurrentNode = Root$.

Demostración:

La prueba es evidente a partir de la propiedad 2.3, ya que la primera de las condiciones de cada componente OR del filtro se ejecuta con una acción $SetSubfilter(G, F_{OR_j}, F_1)$. La primera vez se habilita tras la ejecución de la acción $SetStart(G)$. El resto tras la ejecución de la acción $SetParameter(G, F_{OR_j})$. En todos los casos, los efectos de dichas acciones tienen como resultado que $SetCurrentNode = Root$ y por tanto el nodo subfiltro asociado a la condición F_1 depende del nodo raíz. ■

Cuando se solicita el valor del parámetro G, este valor se obtiene de sumar los contadores de los nodos parámetro asociados a cada componente OR (F_{OR_j}). Esto es posible gracias a que todos los componentes OR de un parámetro están asociados al mismo identificador lo cual se demuestra en la siguiente propiedad.

Propiedad 3 : *Dado un parámetro* $G \in \mathcal{Q}$ *en el árbol, todos los* $F_{OR_j} \in G.F$ *comparten el mismo* ID_G.

Demostración: Por reducción al absurdo, suponemos que cada componente OR tiene distinto ID_j. Por cada F_{OR_j} se ejecuta la acción $SetParameter()$ (propiedad 2.4) que hace uso de la variable $SetID$ para asignar el identificador. Sólo genera un nuevo ID_j si $SetID = 0$ para todas las acciones $SetParameter()$ lo que contradice la propiedad 2.1. Así la acción $SetStart()$ que hace $SetID = 0$ se ejecuta una sola vez para todos los F_{OR_j} de un filtro G y, por tanto, sólo tendrá tal valor para la primera iteración. En el resto de las iteraciones, $SetID = ID_G$ se mantiene para el resto de filtros OR. ■

La siguiente propiedad muestra lo que se puede asegurar a partir de la presencia de un parámetro de monitorización en el árbol. Por un lado, dicho parámetro tiene un ID que lo distingue de los demás parámetros y, por otro lado, existen nodos parámetro donde se almacena el valor de dicho parámetro.

Propiedad 4 : *Sea s_k un estado alcanzable del sistema, $G \in \mathcal{Q}$ donde G es un parámetro incluido en el árbol, siendo su identificador $ID_G \in \mathcal{ID}$. Entonces $s_k.Params(ID_G).Q = G \wedge s_k.Params(ID_G).Counters \neq \emptyset$.*

Demostración: Por reducción al absurdo, suponemos la falsedad de uno u otro de los consecuentes.

- Suponemos $s_k.Params(ID_G).Counters = \emptyset$. El parámetro G se añade al árbol tras la ejecución de las acciones $SetParameter(G, F_{OR_j})$ para cada uno de los componentes OR (propiedad 2.4). La ejecución de estas acciones siempre añade elementos a $Counters$, lo cual contradice que este conjunto pueda ser vacío. En el caso de que se ejecutasen otras acciones del sistema, éstas no afectarían a la propiedad salvo la acción $DelParameter(G, F_{OR_j})$ cuyo efectos es hacer que $Params(ID_G).Counters = \emptyset$ y eliminar el parámetro por lo que ya no se verifica el antecedente de la propiedad (propiedad 1).
- Suponemos $s_k.Params(ID_G).Q \neq G$. Por la propiedad 3 se sabe que todos los filtros OR se insertan con el mismo ID y éste sólo puede coger valor con la ejecución de la acción $SetParameter(G, F_{OR_j}, F_i)$. La ejecución de esta acción tiene como efecto añadir el parámetro G en la estructura $Params$ por lo que se contradice la suposición inicial. ■

El siguiente corolario afirma que todo parámetro configurado en el árbol posee un identificador único. Por tanto, a pesar de que dos parámetros sean iguales, a cada uno de ellos se le asigna un identificador diferente (diferentes entradas en la estructura $Params$). Esto puede parecer poco eficiente en el sistema, ya que se monitorizan dos parámetros iguales. Sin embargo, debido a la estructura del algoritmo propuesto, esta duplicidad de parámetros sólo implica una actualización de contadores ya que los filtrados en ambos casos son comunes. La razón por la cual se permite mantener parámetros iguales es por una cuestión de diseño del sistema para el que se prefiere tener parámetros asociados a cada usuario y que se pueda disponer de un mismo parámetro que se actualiza de manera independiente, con diferentes intervalos de muestreo y diferentes instantes de inicio.

Corolario 3 : *Sea s_k un estado alcanzable del sistema y sean dos parámetros $G, H \in \mathcal{Q}$. Si G, H son parámetros del árbol, entonces $\exists ID_G, ID_H \in \mathcal{ID}$ verificándose que $ID_G \neq ID_H$.*

Demostración:

Los parámetros G, H por estar en el árbol tienen un ID cada uno de ellos según la propiedad 4. Por la propiedad 2, corolario 1 y debido a que la inserción de parámetros

no es concurrente sino FIFO, entonces se cumple que dado $SetBuffer = \mu \bullet G \bullet \mu' \bullet H \bullet \mu''$, $\exists k_1, k_2, k_3,\ k_1 < k_2 < k_3 < k$ tal que $\pi_{k_1} = SetParameter(G, G_{OR_j})$, $\pi_{k_2} = SetStart(H)$, $\pi_{k_3} = SetParameter(H, H_{OR_j})$. Por tanto, G tendrá un ID y H otro ya que π_{k_3} le asigna un nuevo ID porque $SetStart(H)$ inicializa previamente la variable $SetID$ a 0. ■

Las condiciones correspondientes a cada filtro siguen una jerarquía en el árbol de filtrado. La siguiente propiedad muestra las condiciones en las que se crea la estructura de árbol del sistema de filtrado. Por un lado, se comprueba cómo la primera condición de cada componente OR se corresponde con un nodo subfiltro el cual depende del nodo *Root* de la estructura. Por otra parte, se comprueba que cada par de condiciones contiguas de un mismo componente OR, $F_i - F_{i+1}$, se establecen jerárquicamente mediante dos nodos subfiltros relacionados entre sí por medio del elemento *Sons* del primer nodo subfiltro.

Propiedad 5 *: Sea s_k un estado alcanzable del sistema y $G \in \mathcal{Q}$ en el árbol siendo $G.F = \bigvee_{j=1\ldots m}(\bigwedge_{i=1\ldots n_j} F_i)_{OR_j}$, $\wedge ID_G \in \mathcal{ID}$. Entonces, se cumple que:*

- ***5.1:*** $\forall j = 1\ldots m, i = 1, \exists q = (F_{OR_j}.F_1, Used, Sons, Parameters)$ *que es un nodo subfiltro en el sistema de filtrado verificándose que* $(s_k.q.Used > 0) \wedge (q \in s_k.Root.Sons)$
- ***5.2:*** $\forall j = 1\ldots m, i = 2\ldots n_j, \exists p, q$ *con* $p = (F_{OR_j}.F_{i-1}, Used, Sons, Parameters)$, *nodo subfiltro, y* $q = (F_{OR_j}.F_i, Used, Sons, Parameters)$, *también nodo subfiltro, en el sistema de filtrado verificándose que* $(s_k.p.Used > 0) \wedge (s_k.q.Used > 0) \wedge (s_k.q \in s_k.p.Sons)$

Demostración: Comprobamos $q.Used > 0$ para 5.1 y 5.2. Las únicas acciones que afectan a esta variable son $SetParameter()$ y $DelParameter()$. La primera incrementa $Used$ ya que, o bien añade un nuevo nodo subfiltro o incrementa el campo $Used$ de un nodo subfiltro existente. La segunda implica que se ha comenzado el borrado de un parámetro y por tanto no verifica el antecedente de la propiedad de que existe tal parámetro en el árbol.

5.1: Por reducción al absurdo. Por la propiedad 2.3 la acción $SetSubfilter(G, F_{OR_j}, F_i)$ se ejecuta para añadir el nodo subfiltro q. Si $q \notin s_k.Root.Sons$, significa que $SetCurrentNode \neq Root$. Esto significa que no es la primera iteración para el componente OR (F_{OR_j}) del parámetro G por lo que corresponde a F_i, $i > 1$, lo que contradice la premisa.

5.2: En este caso se tiene que, por el apartado anterior de esta propiedad, ambos nodos subfiltro existen y que los valores de sus elementos $Used$ verifican la propiedad. Ahora se debe mostrar que mantienen esa dependencia entre ellos. El razonamiento se puede realizar de manera inductiva, el cual se resume a continuación. El nodo subfiltro asociado a cada una de las primeras condiciones de cada componente OR de un filtro está asociado al elemento *Sons* del nodo *Root* (corolario 2) y además éste se ha añadido tras la ejecución de una acción $SetSubfilter(G, F_{OR_j}, F_1)$. El razonamiento que se expone es válido para cada uno de los componentes OR, con lo que haremos referencia a uno genérico F_{OR_j}.

Analizada la primera condición del componente F_{OR_j}, se tiene que, por la propiedad 2.3, se ejecutarán una serie de acciones que introducen los nodos subfiltro en el sistema de

filtrado. Es decir, $\exists k_i^j$ siendo $k'' < k_i^j < \cdots < k_{n_j}^j < k\,(j = 1 \ldots m, i = 1 \ldots n_j)$ tal que $\pi_{k_i^j} = SetSubfilter(G, F_{OR_j}, F_i)$, las cuales se dan en el orden en que se disponen las condiciones en el componente F_{OR_j}. Entonces $p = (F_{OR_j}.F_{i-1}, Used, Sons, Parameters)$ está en el sistema tras la ejecución de la acción $\pi_{k_{i-1}^j} = SetSubfilter(G, F_{OR_j}.F_{i-1})$, mientras que el nodo subfiltro $q = (F_{OR_j}.F_i, Used, Sons, Parameters)$ se ha insertado tras la ejecución de la acción $\pi_{k_i^j} = SetSubfilter(G, F_{OR_j}.F_i)$. La ejecución de la acción $\pi_{k_{i-1}^j}$ tiene dos efectos importantes en los que interviene la variable $SetCurrentNode$: (i) crea la dependencia con el nodo subfiltro apuntado por dicha variable y (ii) actualiza el campo $Used$.

En el primer caso se tiene que el nodo subfiltro creado, establece la conexión de dependencia con otro nodo subfiltro. En el segundo caso se tiene que en el estado alcanzado, $s_{k_{i-1}^j+1}$, la variable $SetCurrentNode$ del nodo que se ha insertado (p) tiene como contenido a $p.Sons(F_{i-1})$. Es decir, $s_{k_{i-1}^j+1}.SetCurrentNode = p.Sons(F_{i-1})$. No hay otra acción del sistema que pueda modificar este valor salvo una acción $SetSubfilter()$. La siguiente acción de este tipo que se puede ejecutar es $\pi_{k_i^j} = SetSubfilter(G, F_{OR_j}, F_i)$. Los efectos de esta acción hacen que el nodo q esté enlazado con el nodo p ya que, como se ha indicado anteriormente, el valor de la variable $SetCurrentNode$ no se ha modificado y el enlazado se hace previo a la modificación de la variable. La modificación hace que, tras la ejecución de la acción $\pi_{k_i^j}$, dicha variable contenga al nodo subfiltro q. ■

La reutilización de bloques de filtrado, en nuestro caso nodos subfiltro, se demuestra en la siguiente propiedad. Ésta determina que dados ciertos filtros del árbol si estos comparten una secuencia, entonces se reutilizan los nodos subfiltro del árbol. Sólo se creará un nodo subfiltro por cada condición común ($\in f \;/\; G|_f H$) lo que permitirá testear a la vez las condiciones de distintos filtros. Es decir, un mismo nodo subfiltro será compartido por diferentes filtrados y con ello se consigue la reutilización de bloques de filtrado comentada.

Propiedad 6 : *Sean $G, H \in \mathcal{Q}$ parámetros insertados en el árbol, si existe alguna secuencia de condiciones común para ambos $f = F_1 \wedge \ldots \wedge F_p$ tal que $G|_f H$, se reutilizan los subfiltros correspondientes a esas condiciones.*

Demostración: Estado inicial, s_0: En el estado inicial $Root.Sons \leftarrow \emptyset$, luego no existen subfiltros definidos y toda nueva inserción de subfiltro supondrá un nuevo nodo subfiltro hijo (propiedad 5.1), verificándose la propiedad. El nodo *Root* no hace ningún filtrado y es utilizado como raíz para todo el procesado sobre el árbol. La inserción de subfiltros comienza en los nodos *Sons* del *Root*.

En un estado s_k suponemos n parámetros establecidos $Q_1, Q_2, \ldots, Q_n$ cada uno compuesto por ciertos componentes OR ($F_{OR_G}^j$) a su vez compuestos por condiciones $F_{G,i}^j$. Existirán ciertas secuencias de condiciones comunes f:

$$F_{OR_G^j} = F_{G,1}^j \wedge F_{G,2}^j \wedge \ldots \wedge F_{G,n_j}^j = F_1 \wedge \ldots \wedge F_p \wedge F_{G,p+1}^j \wedge \ldots \wedge F_{G,n_j}^j = f \wedge g$$
$$F_{OR_H^h} = F_{H,1}^h \wedge F_{H,2}^h \wedge \ldots \wedge F_{H,n_h}^h = F_1 \wedge \ldots \wedge F_p \wedge F_{H,p+1}^j \wedge \ldots \wedge F_{H,n_h}^h = f \wedge q$$

siendo $f = F_1 \wedge F_2 \wedge \ldots \wedge F_p$ el conjunto de condiciones que comparten.

Por reducción al absurdo. Suponemos que $F_{OR_G^j}$ y $F_{OR_H^h}$ aun teniendo la secuencia común f no comparten los nodos subfiltro $F_1 \ldots F_p$. Si no se comparte ninguno, tampoco se compartirán los F_1. Por el corolario 2 ambos F_1 han de ser hijos del nodo *Root*, pero según la acción $SetSubfilter()$ que se ejecuta por la propiedad 2.3 por cada F_i, si ambos son hijos de *Root* y ambos son iguales sólo se creará un nodo subfiltro hijo de *Root* por lo que se contradice la suposición inicial.

Para el caso de que se compartan los primeros nodos subfiltro correspondientes a las $i-1$ primeras condiciones de la secuencia común $f = F_1 \wedge F_2 \wedge \ldots \wedge F_i \wedge \ldots \wedge F_p$, vamos a suponer que existen dos nodos subfiltro diferentes para F_i. Por la propiedad 5.2 tendrán ambos un nodo subfiltro F_{i-1} que hemos supuesto que se reutiliza. Por tanto, a la hora de insertar F_i se hará en los nodos hijo de F_{i-1} mediante la acción $SetSubfilter(G, F_{OR_j}, F_i)$ por la propiedad 2.3. Esta acción muestra que si existe un nodo subfiltro F_i, si se quiere introducir otro F_i se reutiliza el nodo subfiltro creado con anterioridad. ■

Una cuestión final es que el sistema propuesto realiza las operaciones de inserción, borrado, consulta y actualización de parámetros de forma concurrente. Es decir, las acciones son atómicas pero una operación se compone de cierto número de acciones por lo que éstas se pueden intercalar con las de otras operaciones, pero no con acciones de la misma operación relativas a otro parámetro de monitorización. Es decir, no se pueden realizar dos operaciones de inserción de diferentes parámetros de manera simultánea.

Propiedad 7 : *Acciones que correspondan a operaciones de inserción, borrado, consulta o actualización se pueden intercalar siempre que no afecten al mismo parámetro de monitorización.*

Demostración: Cada operación tiene variables globales comunes entre sus acciones y éstas no son accedidas por el resto de acciones, lo que permite llevar el estado de cada operación. De esta forma, es posible la ejecución intercalada de acciones de diferentes operaciones, pero no de la misma operación correspondiente a diferentes parámetros. Además, existen unos condicionantes que se detallan a continuación.

Durante la inserción de un parámetro no es posible realizar el borrado o consulta del parámetro hasta que no se finalice la inserción porque es entonces cuando se devuelve el identificador que permite estas acciones. Sin embargo, si es posible el borrado, consulta y actualización del resto de parámetros.

En el proceso de borrado de un parámetro no es posible ninguna otra acción que se refiera a ese parámetro porque ese parámetro queda deshabilitado en la acción $DelStart()$ al hacer $Params[ID].Counters = \emptyset$ por la propiedad 1. El resto de operaciones son posibles si hacen referencia a otros parámetros.

Durante operaciones de actualización o consulta se pueden intercalar acciones del resto de operaciones referentes a cualquier parámetro. Esto se debe a que no modifican la estructura del árbol. ■

Las propiedades mostradas en este apartado han permitido comprobar que el sistema de filtrado propuesto en este trabajo aprovecha el análisis de los campos de cada paquete para actualizar más de un parámetro simultáneamente, sin repetir filtrados comunes.

2.6 Conclusiones

En el presente capítulo se ha presentado el algoritmo de filtrado PAM-Tree y para ello se han establecido las definiciones de filtros, condiciones, subfiltros, etc. necesarios para la configuración de parámetros de monitorización en un sistema de filtrado. La estructura de datos de PAM-Tree es un árbol, lo que permite una jerarquía en la disposición de los subfiltros y aprovecharla para reducir el número de comprobaciones por paquete. Este árbol incluye en su estructura información de filtrado (nodos subfiltro) e información de monitorización (nodos parámetro), lo que permite proveer gran parte de las características del sistema. Además, debido a la flexibilidad en la definición de filtrados asociados a parámetros de monitorización y la concatenación AND/OR de condiciones, el algoritmo propuesto permite gran libertad en la definición de tareas de monitorización. Las operaciones básicas de inserción, borrado, actualización y consulta de parámetros permiten describir parte de sus funcionalidades.

En cuanto a la formalización mediante teoría de autómatas de entrada/salida, permite detallar el funcionamiento del algoritmo. Además, permite proponer las acciones del sistema y la demostración de las propiedades principales. Entre estas propiedades destaca la reutilización de bloques de filtrado, que proporciona gran parte de la eficiencia del algoritmo propuesto.

Capítulo 3

Análisis de prestaciones

3.1 Introducción

Una vez presentado el algoritmo PAM-Tree se hace necesario realizar un análisis de sus prestaciones en una aplicación de monitorización. Primeramente, se presentará un análisis del algoritmo propuesto frente a la alternativa tradicional de independencia entre filtrados. Para ello, se propondrá un modelo analítico para el estudio de PAM-Tree y se presentarán sus resultados.

Posteriormente se presentará una comparativa experimental del PAM-Tree frente a BPF y LSF (Linux Socket Filter, implantación del filtrado BPF en Linux) tomados como referencia de los sistemas de filtrado actuales. No se ha comparado con NNstat por sus altos costes computacionales en la inserción y borrado de filtros, debido a la estructura estática que implementa y que supone rehacer toda la estructura de filtrado cada vez que se modifique la configuración de filtros. Esto hace que LSF/BPF mejoren las características de NNstat [14], por lo que usaremos los primeros como referencia. Se comprobará el comportamiento de PAM-Tree y LSF/BPF ante diferentes situaciones en cuanto a número de filtros establecidos, el tipo de éstos y la carga de la red.

3.2 Evaluación analítica del PAM-Tree

El objetivo del presente apartado es la definición de un modelo para calcular el coste de realizar el filtrado de paquetes mediante el algoritmo PAM-Tree. Este algoritmo provee gran parte de sus funcionalidades gracias a separar los filtros en los distintos nodos de un árbol de filtrado, de manera que filtrados que tengan condiciones comunes a niveles inferiores compartan nodos subfiltro del árbol. En lo que sigue denominaremos con subfiltro indistintamente al nodo subfiltro o a la condición asociada F_i al nodo subfiltro.

Esta propuesta de separar los filtros en las unidades más pequeñas posibles denominadas subfiltros es opuesta a la técnica de componer subfiltros en subfiltros más grandes. Esta segunda alternativa ya no va a permitir que se compartan subfiltros a niveles inferiores, a

cambio de implicar un coste de filtrado en principio menor cuando el número de filtros sea pequeño. Esto se consigue al realizar el testeo del subfiltro sobre varios campos del paquete de manera simultánea cuando sea posible y tratar cada subfiltro de manera independiente de los demás sin aprovechar subfiltrados comunes. Esta técnica es la habitualmente utilizada en los packet filters.

En este apartado vamos a proponer un modelo que nos permita comparar el coste de ambas técnicas. Comprobaremos como el separar subfiltros según un árbol de filtrado es más eficiente en la gran mayoría de los casos.

3.2.1 Modelo de filtrado

Como ya se presentó en el capítulo anterior, cada nodo del árbol representa un subfiltro, y una determinada configuración de enlaces entre nodos corresponderá a un filtro.

El árbol de filtrado se puede modelar como un sistema M/G/1 (figura 3.1), asemejando el tiempo de servicio de este sistema con el coste de realizar el filtrado sobre el árbol de filtrado, $coste = 1/\mu$. Para aprovecharnos de este modelo habrá que suponer una distribución del tiempo entre llegadas exponencial y un tiempo de servicio general provocado por un servidor para cada campo a testear. Este servidor lo asociaremos a un nivel de filtrado, es decir, al testeo de determinado campo. Por tanto, subfiltros que testeen el mismo campo pertenecerán al mismo servidor.

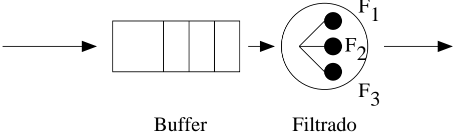


Figura 3.1: Modelo del clasificador

Suponemos que el sistema recibe una serie de paquetes compuestos por una serie de campos sobre los cuales podemos estar interesados en aplicar un filtrado. Cada campo a testear supone un subfiltro F_i que modelaremos con un servidor. Si el campo en cuestión ya está siendo testeado por otro subfiltro, lo añadiremos al servidor existente.

A la hora de calcular el coste de un filtro genérico $F = F_{OR_1} \vee F_{OR_2} \vee \ldots \vee F_{OR_m}$ se puede considerar cada componente OR de manera independiente como si fueran otros filtros establecidos en el árbol. Por tanto, nos centraremos en el coste de un filtrado $F_{OR_j} = F_1 \wedge F_2 \wedge \ldots \wedge F_{n_j}$ donde F_i es cada una de las condiciones representadas con un nodo subfiltro en el árbol.

Cada campo i, asociado a un nivel del sistema, tiene O_i posibles tipos de ocurrencias: por ejemplo, si el campo tuviese n bits, se podrían tener a lo sumo 2^n posibilidades para ese campo, luego $O_i = 2^n$. Por otro lado, debido a la existencia de otros filtros, puede haber definidos K_i subfiltros sobre ese campo. Es decir, de los O_i subfiltros que se podrían definir sobre el campo i tenemos K_i. Por ejemplo, en la figura 3.2 se muestra un paquete sobre el que se testean 3 campos, cada uno K_i veces lo que supone 3 nodos subfiltro en el árbol y por tanto 3 servidores

en el modelo con una única cola de entrada. Una posible configuración del árbol de filtrado y su mapeo al modelo se presentan en la figura 3.3.

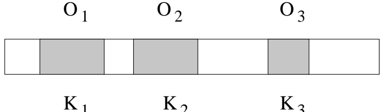


Figura 3.2: Campos de un paquete a filtrar

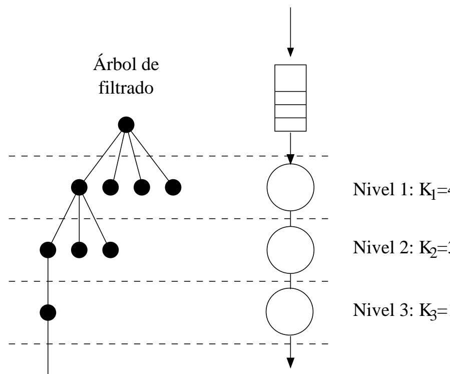


Figura 3.3: Ejemplo de relación entre el árbol de filtrado y el modelo propuesto

El servidor del sistema se asimila al nodo del árbol de filtrado. En cada nodo i del árbol de filtrado se debe comprobar K_i posibles valores del campo i del paquete. Dependiendo del tipo de búsqueda, el coste de esta comprobación será diferente. En el caso general, $\frac{1}{\mu_i} = f(K_i)$ donde $f(K_i)$ depende del algoritmo de búsqueda.

Vamos a suponer una búsqueda lineal típica para realizar el filtrado, más costosa que otros algoritmos de búsqueda, como el hash o el algoritmo del árbol Patricia (apéndice A), pero también más flexible a la hora de añadir o eliminar filtros sin apenas sobrecarga computacional.

De todos los servidores que corresponden a cierta configuración del árbol de filtrado, nos interesa analizar los servidores que contienen algún subfiltro del filtro que se esté estudiando. Por tanto, nos queda un conjunto de servidores en serie que se corresponden con el modelo de un sistema M/G/1 como el de la figura 3.4. Este sistema nos modela el coste de un filtro junto con el efecto de subfiltros compartidos con otros filtros. El coste será el tiempo de servicio del sistema:

$$coste = tiempo\ de\ servicio = \overline{x} = 1/\mu \tag{3.1}$$

Un paquete que entra en el sistema pasa al primer servidor que evalúa los K_1 subfiltros a ese nivel. Si hay éxito se pasa al siguiente servidor que corresponda y si no lo hay sale del sistema y termina el proceso.

La probabilidad de que un paquete satisfaga un subfiltro de un campo es $p_i = \frac{K_i}{O_i}$. El paso del paquete por distintos subfiltros se puede modelar como un sistema de M servidores en

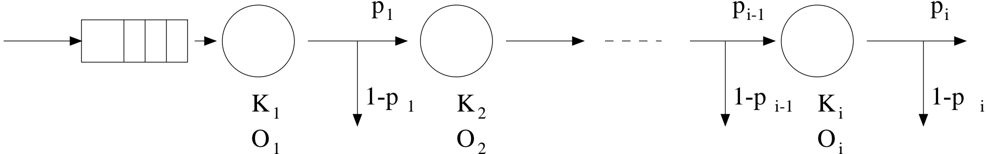


Figura 3.4: Unión de nodos

serie donde cada uno de ellos representa el campo i y donde M representa la rama más larga del árbol. En la figura 3.4 se puede observar el sistema completo.

Se trata de comparar el coste de separar filtros en subfiltros y de esta forma poder aprovechar subfiltros comunes a niveles inferiores frente a la de intentar componer subfiltros de manera que el filtro esté compuesto por el menor número de subfiltros posible. A la primera alternativa la denominaremos *sistema fragmentado*, que es la que se utiliza en el algoritmo PAM-Tree, y a la segunda *sistema compuesto*.

El coste de realizar una búsqueda en el árbol de filtrado se puede asimilar al tiempo de servicio del sistema M/G/1. En este apartado estudiaremos el tiempo de servicio del sistema, que cuanto menor sea hará que podamos soportar un mayor número de filtros. Este modelo supone una distribución del tiempo entre llegadas exponencial y un tiempo de servicio general provocado por un único servidor (en verdad M servidores en serie que equivalen a uno único de diferente tiempo de servicio según las probabilidades de paso a uno u otro servidor).

Por otro lado nos interesarán las características de tiempo real del sistema. En un sistema de monitorización es importante tener los datos de cada intervalo de monitorización en el menor tiempo posible. Por ejemplo, puede hacerse necesario conocer en cada segundo cual ha sido la utilización de una red y tener este dato cuanto antes. Este tiempo tras el segundo de medida en el que tendremos el dato estará relacionado con el tiempo medio en el sistema de nuestro modelo que también será objeto de estudio.

Los parámetros del sistema, que dependerán de los filtros que tengamos en determinado momento, serán K_i, O_i de cada nivel de filtrado $i = 1...M$ y el número de filtros establecidos n. Además λ, la tasa de llegadas, fijará la utilización del sistema, ρ.

Tendremos que determinar la media del tiempo de servicio, $\overline{x} = E[x]$ y el segundo momento del tiempo de servicio $\overline{x^2} = E[x^2]$ de los sistemas compuesto y fragmentado, para poder compararlos. Así se podrá estudiar el tiempo medio de espera en cola dado por la siguiente expresión para un sistema M/G/1 ([83, capítulo 8] [84, capítulo 5]):

$$W_q = \frac{\lambda \overline{x^2}}{2(1-\rho)} \tag{3.2}$$

De manera inmediata el tiempo medio en el sistema será:

$$W = W_q + \overline{x} \tag{3.3}$$

En lo que sigue tomaremos una operación de comparación como la referencia de coste computacional y la haremos igual a la unidad para el cálculo del tiempo de servicio.

3.2.2 Análisis de prestaciones para dos niveles de filtrado

En lo sucesivo vamos a suponer el caso de dos nodos cuyos resultados serán posteriormente extensibles al caso de N nodos. Para dos nodos, es decir, dos niveles de subfiltros, tenemos una situación como la mostrada en la figura 3.5. En el nivel 1 tenemos un filtrado sobre un campo del paquete con O_1 posibilidades de las cuales estamos interesados en K_1 valores de ese campo. Si se verifica alguno de estos valores, se volverá a repetir el proceso con los posibles O_2 valores del campo asociado al nivel 2. Si el campo coincide con alguno de los K_2 valores se habrá verificado el filtro conjunto. En caso contrario el filtro no se verifica.

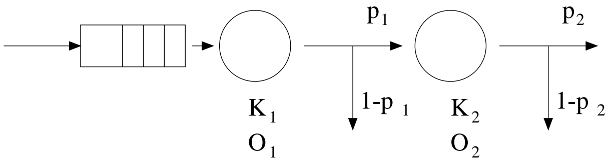


Figura 3.5: Modelo para 2 nodos del árbol

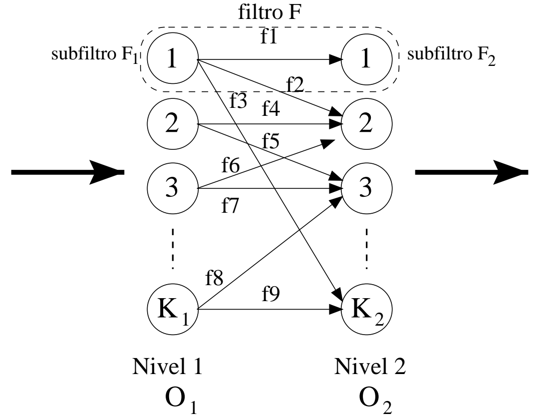


Figura 3.6: Filtros formados entre nodos

En la figura 3.6 tenemos un ejemplo de disposición de K_1 y K_2 subfiltros. Los subfiltros de los dos niveles se pueden unir de formas diferentes dando lugar a los f1, f2,.., f9 filtros de la figura. El número de filtros simultáneos (9 en la figura 3.6) será un parámetro determinante a la hora de calcular el coste total del sistema.

Tendremos por tanto los dos casos extremos, según el número de comparaciones que se deben realizar para determinar si se verifica un filtro:

- Caso mejor, (figura 3.7.a), será aquel caso con el mínimo número de filtros pero conectando todos los subfiltros de cada nivel. El número de filtros mínimo será $max(K_1, K_2)$.

- Caso peor, (figura 3.7.b), será aquel caso con el mayor número de filtros sin repetir unión entre los mismos pares de subfiltros. El número de filtros máximo será $K_1 \cdot K_2$.

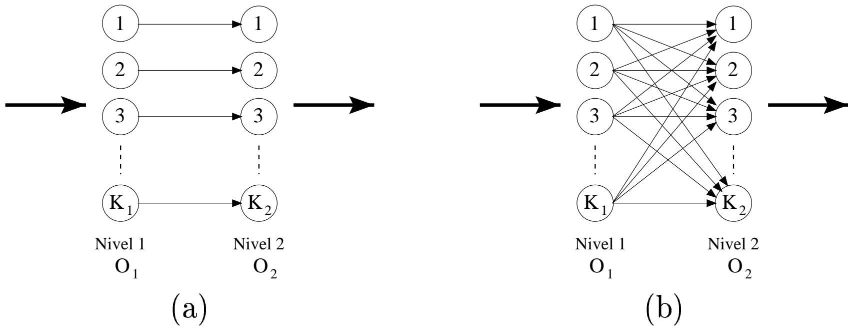


Figura 3.7: Filtros formados entre nodos a) para el caso mejor y b) para el caso peor

En el presente análisis se van a presentar las prestaciones del sistema fragmentado frente al compuesto para el caso concreto de dos niveles de filtrado. Por tanto, los filtros a considerar en este apartado serán aquellos que tengan dos subfiltros, necesariamente uno en cada nivel. En la figura 3.8 se presentan las dos configuraciones de sistema compuesto y sistema fragmentado para 2 niveles de filtrado.

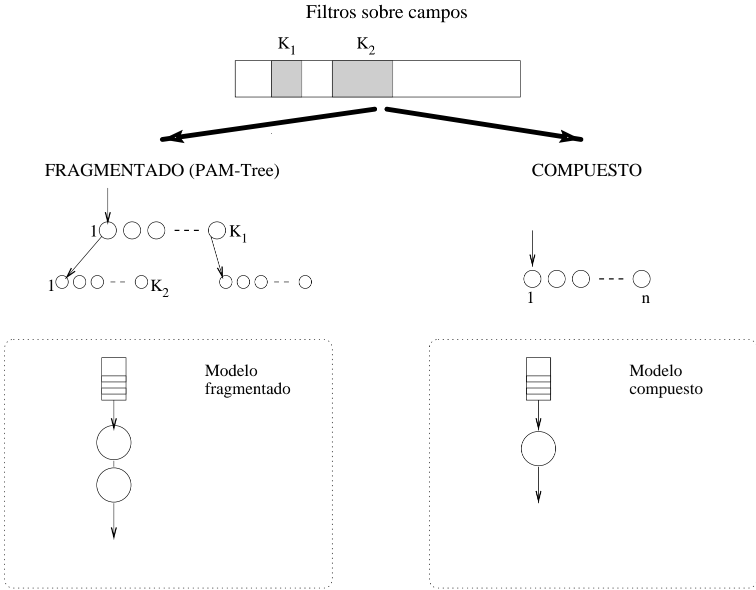


Figura 3.8: Sistema fragmentado y compuesto para 2 niveles de filtrado

3.2.2.1 Sistema compuesto

La composición de filtros consiste en unir subfiltros en un único subfiltro. Si componemos el par de subfiltros de la figura 3.5, el subfiltro resultante tendrá un único servidor en el modelo. Ahora por cada par de subfiltros de antes (2 comparaciones de dos campos) tendremos un sólo subfiltro con un coste del que supondremos dos casos:

- Coste 1: coste de un sólo subfiltro anterior, es decir, de una única comparación. Por tanto, se está suponiendo el caso más favorable en la unión de subfiltros en el que se pueden unir las comparaciones de los dos subfiltros anteriores en una única comparación nueva. Por ejemplo, si suponemos que queremos filtrar por puerto origen y destino de un paquete TCP/IP, ambos campos suman 32 bits y se pueden testear de una vez juntando ambos campos (en un procesador genérico de 32 bits)

- Coste 2: coste equivalente a los dos subfiltros anteriores, es decir, dos comparaciones. Será el caso más habitual debido a la dificultad de unir comparaciones de campos en una única comparación, unas veces por su tamaño u otras porque se encuentren separados en la cabecera del paquete.

De esta forma, el coste del subfiltro resultante de esta composición de filtros corresponde al coste medio de una búsqueda lineal que será aproximadamente $coste \approx S/2$ donde S es el número de subfiltros existentes a ese nivel, suponiendo el caso coste 1. Este nuevo número de subfiltros corresponderá con el número de filtros establecidos entre pares de subfiltros en el sistema fragmentado. Así, para el ejemplo de la figura 3.6 resultará un servidor compuesto formado por 9 subfiltros.

El número de subfiltros compuestos podrá variar como se ha visto en el apartado anterior entre $max(K_1, K_2)$ y $K_1 \cdot K_2$ para el mejor y peor caso respectivamente, verificándose que:

$$\frac{max(K_1, K_2)}{2} \leq coste \leq \frac{(K_1 \cdot K_2)}{2} \tag{3.4}$$

Consideraremos x como variable aleatoria que representa el tiempo de servicio del sistema, es decir, el tiempo de procesado de un paquete. Si se tienen n filtros establecidos entre dos nodos del árbol de filtrado, el servidor compuesto resultante poseerá n subfiltros (el filtro está compuesto por un único subfiltro resultado de operar de manera simultánea con los dos subfiltros originales) y por tanto el tiempo medio de servicio condicionado al número de filtros n será el tiempo de recorrido de una búsqueda lineal, supuesta uniforme la distribución de los subfiltros a buscar. Así, para el caso de coste 1:

$$E_{c1}[x|n] = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

$$E_{c1}[x] = E[E_{c1}[x|n]] = \frac{1}{K_1K_2 - max(K_1, K_2) + 1} \sum_{n=max(K_1,K_2)}^{K_1K_2} \frac{n+1}{2} \tag{3.5}$$

En estas expresiones $E[x|n]$ indica la esperanza de x condicionada a n, y resulta ser el coste medio de una búsqueda lineal de n elementos. Hacer notar que sólo se puede verificar uno de los filtros para cada paquete de entrada porque si no se trataría del mismo filtro (compuesto de los mismos subfiltros). Respecto al segundo momento del tiempo de servicio:

$$E_{c1}[x^2|n] = \frac{1}{n}\sum_{i=1}^{n} i^2 = \frac{(n+1)(2n+1)}{6}$$

$$E_{c1}[x^2] = E[E_{c1}[x^2|n]] = \frac{1}{K_1K_2 - max(K_1, K_2) + 1} \sum_{n=max(K_1,K_2)}^{K_1K_2} \frac{(n+1)(2n+1)}{6} \tag{3.6}$$

Igualmente se obtienen las expresiones para el sistema compuesto caso 2:

$$E_{c2}[x|n] = E[E_{c2}[x|n]] = \frac{1}{n}\sum_{i=1}^{n} 2i = n+1$$

$$E_{c2}[x] = E[E_{c2}[x|n]] = \frac{1}{K_1K_2 - max(K_1, K_2) + 1} \sum_{n=max(K_1,K_2)}^{K_1K_2} n+1 \tag{3.7}$$

$$E_{c2}[x^2|n] = \frac{1}{n}\sum_{i=1}^{n} (2i)^2 = \frac{2(n+1)(2n+1)}{3}$$

$$E_{c2}[x^2] = E[E_{c2}[x^2|n]] = \frac{1}{K_1K_2 - max(K_1, K_2) + 1} \sum_{n=max(K_1,K_2)}^{K_1K_2} \frac{2(n+1)(2n+1)}{3} \tag{3.8}$$

3.2.2.2 Sistema fragmentado, PAM-Tree

La fragmentación del filtrado consiste en la separación de filtros en subfiltros y corresponde al modelo del árbol de filtrado PAM-Tree en estudio. Si consideramos los dos subfiltros de la figura 3.5 por separado, el primero se testeará siempre y el segundo sólo en el caso de que se verifique alguno de los valores del primero, es decir, con probabilidad:

$$p_1 = \frac{K_1}{O_1} \tag{3.9}$$

Por tanto, el coste resultante del conjunto de ambos subfiltros será:

$$E_f[x] = (1 - p_1)E[x_{N,1}] + p_1(E[x_{S,1}] + E[x_2]) \tag{3.10}$$

donde:

- $E[x_{N,1}]$ es el tiempo empleado en el primer nivel cuando no se verifica ningún subfiltro que corresponderá a una búsqueda lineal en la que se llega siempre hasta el final de los elementos K_1 porque no se encuentra coincidencia.

$$E[x_{N,1}] = K_1 \tag{3.11}$$

- $E[x_{S,1}]$ es el tiempo empleado en el primer nivel cuando se verifica algún subfiltro. En esta búsqueda cuando se verifica un subfiltro no se continúa con la búsqueda lineal (sólo se puede cumplir un subfiltro en cada nivel) por lo que el coste medio será la mitad del número de elementos a recorrer más uno.

$$E[x_{S,1}] = \frac{1}{K_1}\sum_{i=1}^{K_1} i = \frac{K_1+1}{2} \tag{3.12}$$

- $E[x_2]$ es el tiempo empleado en el segundo nivel que desglosaremos a continuación.

La esperanza del tiempo de servicio se puede poner en función de la esperanza del tiempo de servicio condicionada a n:

$$E_f[x] = E[E[x|n]] = \frac{1}{K_1K_2 - max(K_1,K_2)+1}\sum_{n=max(K_1,K_2)}^{K_1K_2} E[x|n] \tag{3.13}$$

Por tanto se puede poner:

$$\begin{aligned} E[x|n] &= (1-p_1)E[x_{N,1}|n] + p_1(E[x_{S,1}|n] + E[x_2|n]) = \\ &= (1-p_1)E[x_{N,1}] + p_1(E[x_{S,1}] + E[x_2|n]) = \\ &= \left(1-\frac{K_1}{O_1}\right)K_1 + \left(\frac{K_1}{O_1}\right)\left(\frac{K_1+1}{2} + E[x_2|n]\right) = \\ &= K_1\left(1+\frac{1-K_1}{2O_1}\right) + \left(\frac{K_1}{O_1}\right)(E[x_2|n]) \end{aligned} \tag{3.14}$$

El valor de $E[x_2|n]$ depende de a_i que es el número de filtros establecidos entre cada subfiltro i del primer nivel con subfiltros del segundo nivel. En la figura 3.9 se observa un caso de $a_i = 4$ filtros establecidos con el segundo nivel. Se trata de una variable aleatoria uniforme que verifica:

$$\begin{cases} a_1 + a_2 + \ldots + a_n = n \\ a_i \in [max\,(1,(n-(K_1-1)K_2))\,, min\,(K_2, n-(K_1-1))] \end{cases} \tag{3.15}$$

Por un lado la suma de todos los a_i establecidos entre los dos niveles debe ser el número de filtros total n y por otro existen unos valores que definen su rango:

- El valor inferior de a_i será el mayor de 1 (siempre al menos un filtro establecido) y el número de filtros que faltan para completar los n para el caso de que el resto de subfiltros de primer nivel tengan filtros establecidos con todos los subfiltros de segundo nivel $(n-(K_1-1)K_2)$.
- El valor superior de a_i será el menor entre K_2 (como máximo puede estar unido con todos los K_2 filtros del segundo nivel) y el número de filtros que faltan para completar los n para el caso de que el resto de subfiltros del primer nivel estén conectados con un sólo filtro $n-(K_1-1)$.

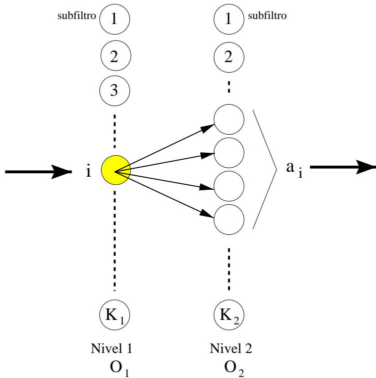


Figura 3.9: Filtros establecidos con el nodo i del primer nivel

Para calcular la probabilidad de que haya a_i filtros en el segundo nivel podemos suponer que tenemos K_1 segmentos, uno por cada subfiltro del primer nivel, de K_2 posiciones (figura 3.10) que representa a cada uno de los K_2 subfiltros del segundo nivel con los que un subfiltro de primer nivel puede tener establecido un filtro. En la figura 3.10 se representa esta disposición, con una X marcando los filtros establecidos. Debe haber al menos un filtro establecido con un subfiltro del segundo nivel: al menos debe haber un filtro establecido entre todo subfiltro del primer nivel y alguno del segundo para que lo estemos considerando, si no el subfiltro correspondería a otro servidor de un nivel y no a éste, ya que estamos modelando el caso de dos niveles.

Figura 3.10: Modelo de establecimiento de filtros entre subfiltros a 2 niveles

Calculamos la probabilidad de tener a_i filtros para todo el rango mostrado en la expresión 3.15. Para ello el número de combinaciones de filtros posibles es:

$$\text{Número total de ocurrencias posibles} = \binom{K_1(K_2-1)}{n-K_1} \tag{3.16}$$

La expresión 3.16 tiene en cuenta las combinaciones posibles de filtros que son $K_1(K_2-1)$ (número de huecos donde establecer filtros, teniendo siempre uno al menos fijo por lo que queda K_1 segmentos por (K_2-1) huecos por segmento) tomados en $(n-K_1)$ (número de filtros a establecer fijado uno por segmento).

De esta forma, la probabilidad de tener un subfiltro de primer nivel conectado con a_i subfiltros de segundo nivel vendrá dada por:

$$P(a_i = j) \quad = \quad \frac{\binom{K_2-1}{j-1}\binom{(K_1-1)(K_2-1)}{n-K_1-(j-1)}}{\binom{K_1(K_2-1)}{n-K_1}} =$$

$$= \frac{C_{K_2-1}(j-1)C_{(K_1-1)(K_2-1)}(n-K_1-(j-1))}{C_{K_1(K_2-1)}(n-K_1)} \tag{3.17}$$

La expresión 3.17 consiste en el producto de las combinaciones de $j-1$ subfiltros (1 es fijo) sobre K_2-1 posibles correspondientes al subfiltro i del primer nivel en estudio. Esto se multiplica por las combinaciones del resto de subfiltros del primer nivel con el segundo nivel que son las combinaciones de $(n-K_1-(j-1))$ filtros que quedan por asignar sobre las posibilidades con el resto de subfiltros de primer nivel con el segundo que es $(K_1-1)(K_2-1)$ (K_1-1 subfiltros de primer nivel quitado el i con K_2-1 subfiltros de segundo nivel quitando 1 que debe existir). Todo ello se divide por el número total de ocurrencias posibles. Se ha hecho uso de la expresión 3.18 para simplificar las expresiones combinacionales.

$$C_a(b) = \binom{a}{b} = \frac{a!}{b!(a-b)!} \tag{3.18}$$

Y como toda distribución de probabilidad, verifica que:

$$\sum_{j=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} P(a_i = j) = 1 \tag{3.19}$$

Resulta que la expresión 3.17 sigue una *distribución hipergeométrica* de la forma expresada en la expresión 3.20 [85, 86], donde $M = K_2 - 1$, $x = a_i - 1$, $N = K_1(K_2-1)$ y $n' = n - K_1$. Por simplificar, nombraremos la nueva variable n' con n.

$$f(x) = \frac{\binom{M}{x}\binom{N-M}{n-x}}{\binom{N}{n}} \tag{3.20}$$

Esta distribución posee una función generadora $G(s) = \sum_k s^k f(k)$ que resulta ser el coeficiente de x^n en la siguiente expresión 3.21 [87] (desarrollando el binomio). De esta expresión, se podrán obtener los momentos de la función de distribución sin más que derivar y tomar el resultado en $s = 1$ de la función generadora ya que $G^{(k)}(1) = E[a_i(a_i-1)\cdots(a_i-k+1)]$ [87, 88]. Los momentos resultantes se muestran en las expresiones 3.22-3.24.

$$Q(s,x) = (1+sx)^b(1+x)^{N-M}/\binom{N}{n} \tag{3.21}$$

$$E[a_i] = G'(1) = \frac{Mn}{N} = \frac{n-K_1}{K_1} \tag{3.22}$$

$$\begin{aligned} E[a_i^2] &= G''(1) + G'(1) = \frac{Mn}{N}\left(\frac{N-n}{N-1}\right)\left(1 + \frac{M(n-1)}{N-n}\right) = \\ &= \frac{n-K_1}{K_1}\left(\frac{K_1K_2-n}{K_1(K_2-1)-1}\right)\left(1 + \frac{(K_2-1)(n-K_1-1)}{K_1K_2-n}\right) \end{aligned} \tag{3.23}$$

$$E[a_i^3] = G'''(1) + 3G''(1) + G'(1) = \frac{Mn}{N}\left[\frac{(M-1)(n-1)}{N-1}\left(\frac{(M-2)(n-2)}{N-2} - 3\right) - 1\right] =$$

$$= \frac{n-K_1}{K_1}\left(\frac{(K_2-2)(n-K_1-1)}{K_1(K_2-1)-1}\left(\frac{(K_2-3)(n-K_1-2)}{K_1(K_2-1)-2}+3\right)+1\right) \quad (3.24)$$

La esperanza del tiempo de servicio condicionado a n, $E[x_2|n]$, se puede poner en función de la esperanza del tiempo de servicio condicionado a a_i:

$$\begin{aligned} E[x_2|n] &= E[E[x_2|n|a_i]] = \\ &= \sum_{a_i=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} E[x_2|n|a_i]P(a_i) \end{aligned} \quad (3.25)$$

La expresión $E[x_2|n|a_i]$ corresponde al tiempo de servicio medio de un subfiltro i del primer nivel que tiene a_i filtros establecidos con a_i subfiltros del segundo nivel. Se puede simplificar así la expresión porque $E[x_2|n|a_i]$ no depende de i, es decir, del subfiltro que se tome del primer nivel. Si lo desarrollamos:

$$E[x_2|n|a_i] = (1-p_2)E[x_{N,2}] + p_2E[x_{S,2}] \quad (3.26)$$

donde:

- $p_2 = a_i/O_2$ es la probabilidad de que se verifique algún filtro de este segundo nivel.
- $E[x_{N,2}]$ es el tiempo empleado en el segundo nivel cuando no se verifica ningún subfiltro. Como en el primer nivel: $E[x_{N,2}] = a_i$
- $E[x_{S,2}]$ es el tiempo empleado en el segundo nivel cuando se verifica algún subfiltro. Como en el primer nivel: $E[x_{S,2}] = \frac{1}{a_i}\sum_{j=1}^{a_i} j = \frac{a_i+1}{2}$

De lo que la expresión 3.26 queda como:

$$E[x_2|n|a_i] = a_i\left(1+\frac{1-a_i}{2O_2}\right) \quad (3.27)$$

Sustituyendo las expresiones 3.22, 3.23 y 3.27 en 3.25 tenemos:

$$\begin{aligned}
E[x_2|n] &= \sum_{a_i=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} a_i\left(1+\frac{1-a_i}{2O_2}\right)P(a_i) = \\
&= \sum_{a_i=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} a_iP(a_i) + \frac{1}{2O_2}a_iP(a_i) - \frac{1}{2O_2}a_i^2P(a_i) = \\
&= E[a_i] + \frac{1}{2O_2}\left(E[a_i]-E[a_i^2]\right) = \\
&= \frac{n-K_1}{K_1}\left(1+\frac{1}{2O_2}\Big(\frac{(K_2-2)(K_1+1-n)}{K_1(K_2-1)-1}\Big)\right)
\end{aligned} \tag{3.28}$$

De esta forma, sustituyendo en la expresión 3.14 tendremos:

$$E[x|n] = K_1\left(1+\frac{1-K_1}{2O_1}\right)+\left(\frac{K_1}{O_1}\right)\left[\frac{n-K_1}{K_1}\left(1+\frac{1}{2O_2}\Big(\frac{(K_2-2)(K_1+1-n)}{K_1(K_2-1)-1}\Big)\right)\right] \tag{3.29}$$

Por tanto, la expresión de la media del tiempo de servicio quedará de esta forma sustituyendo en 3.13:

$$\begin{aligned}
E_f[x] = \frac{1}{K_1K_2-max(K_1,K_2)+1}\sum_{n=max(K_1,K_2)}^{K_1K_2}\left(K_1\left(1+\frac{1-K_1}{2O_1}\right)+\right. \\
\left.+\left(\frac{K_1}{O_1}\right)\left[\frac{n-K_1}{K_1}\left(1+\frac{1}{2O_2}\Big(\frac{(K_2-2)(K_1+1-n)}{K_1(K_2-1)-1}\Big)\right)\right]\right)
\end{aligned} \tag{3.30}$$

Para el cálculo del segundo momento se procederá de igual forma.

$$E_f[x^2] = E[E[x^2|n]] = \frac{1}{K_1K_2-max(K_1,K_2)+1}\sum_{n=max(K_1,K_2)}^{K_1K_2}E[x^2|n] \tag{3.31}$$

Suponiendo igual que antes $x_{N,i}$ el tiempo de servicio del nivel i si no se verifica ningún subfiltro y $x_{S,i}$ el tiempo de servicio del nivel i si se verifica algún subfiltro tendremos:

$$E[x^2|n] = (1-p_1)E[x_{N,1}^2|n] + p_1 E[(x_{S,1}+x_2)^2|n] =$$
$$= \left(1-\frac{K_1}{O_1}\right)K_1^2 + \left(\frac{K_1}{O_1}\right)\left(E[x_{S,1}^2|n] + E[x_2^2|n] + 2E[x_{S,1}\cdot x_2|n]\right) \quad (3.32)$$

donde:

- Por independencia entre las variables del coste de nivel 1 y del coste de nivel 2:

$$E[x_{S,1}\cdot x_2|n] = E[x_{S,1}|n]E[x_2|n] \quad (3.33)$$

- $E[x_{S,1}|n]$ es el de la expresión 3.12 y $E[x_2|n]$ es el de la expresión 3.28.
- El tiempo de servicio $x_{S,1}$ corresponde al de encontrar el subfiltro por lo que el segundo momento será:

$$E[x_{S,1}^2|n] = \frac{1}{K_1}\sum_{j=1}^{K_1} j^2 = \frac{(K_1+1)(2K_1+1)}{6} \quad (3.34)$$

- Como antes se puede poner el segundo momento del tiempo de servicio para el segundo nivel en función de a_i:

$$E[x_2^2|n] = \sum_{a_i=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} E[x_2^2|n|a_i]P(a_i) \quad (3.35)$$

$$E[x_2^2|n|a_i] = (1-p_2)E[x_{N,2}^2|n|a_i] + p_2 E[x_{S,2}^2|n|a_i] \quad (3.36)$$

- El tiempo de servicio $x_{N,2}$ es el coste de recorrido de búsqueda lineal cuando no se verifica ningún subfiltro que será a_i constante y por tanto:

$$E[x_{N,2}^2|n|a_i] = a_i^2 \quad (3.37)$$

- El tiempo de servicio $x_{S,2}$ es el coste de recorrido de búsqueda lineal cuando se verifica algún subfiltro de los a_i existentes, que será por tanto:

$$E[x_{S,2}^2|n|a_i] = \frac{1}{a_i}\sum_{j=1}^{a_i} j^2 = \frac{(a_i+1)(2a_i+1)}{6} \quad (3.38)$$

Con todo ello se obtiene la siguiente expresión:

$$E_f[x^2] = \frac{1}{K_1K_2 - max(K_1,K_2)+1} \sum_{n=max(K_1,K_2)}^{K_1K_2} \left[\left(1 - \frac{K_1}{O_1}\right) K_1^2 + \left(\frac{K_1}{O_1}\right) \left[\frac{(K_1+1)(2K_1+1)}{6} + \right.\right.$$
$$\left.\left. + \left(\sum_{a_i=max(1,(n-(K_1-1)K_2))}^{min(K_2,n-(K_1-1))} \left(a_i \left(1 + \frac{1-a_i}{2O_2}\right) (K_1+1) + \frac{(6O_2+3)a_i^2 - 4a_i^3 + a_i}{6O_2} \right) P(a_i) \right) \right] \right] =$$
$$= \frac{1}{K_1K_2 - max(K_1,K_2)+1} \sum_{n=max(K_1,K_2)}^{K_1K_2} \left[\left(1 - \frac{K_1}{O_1}\right) K_1^2 + \left(\frac{K_1}{O_1}\right) \left[\frac{(K_1+1)(2K_1+1)}{6} + \right.\right. \tag{3.39}$$
$$\left.\left. + \left(\frac{-2}{3O_2} E[a_i^3] + \left(\frac{2O_2+1-K_1}{2O_2}\right) E[a_i^2] + \left((K_1-1)(1+\frac{2}{2O_2}) + \frac{1}{6O_2}\right) E[a_i] \right) \right] \right]$$

donde $E[a_i^k]$ es el correspondiente de las expresiones 3.22, 3.23 y 3.24.

Después de haber obtenido estos cálculos ya estamos en condiciones de estudiar el tiempo medio de servicio y el tiempo medio de espera en cola para las dos alternativas de sistema compuesto y de sistema fragmentado. Las expresiones resultantes son complejas y dependen de multitud de parámetros por lo que se van a presentar una serie de resultados numéricos para poder extraer conclusiones.

3.2.2.3 Comparativa del tiempo de servicio

En la figura 3.11 se observa el tiempo medio de servicio para el sistema fragmentado $E_f[x]$ y para el sistema compuesto $E_{c1}[x]$ (caso coste 1) y $E_{c2}[x]$ (caso coste 2), junto con barras de error que representan el máximo y mínimo para cada número de filtros n, obtenidos a partir de las expresiones anteriores. Para realizar esta gráfica se han variado los parámetros $1 \leq O_1, O_2 \leq 5$, $1 \leq K_1 \leq O_1$ y $1 \leq K_2 \leq O_2$, para una tasa $\lambda = 0.01$ ($\rho = \lambda E[x]$). Se observa como para un bajo número de filtros el sistema compuesto $E_{c1}[x]$ es mejor, pero en cuanto aumenta el número de filtros simultáneos el sistema fragmentado es la mejor opción. Este resultado era el previsible en un principio porque con pocos filtros son mejores implementaciones monolíticas siempre que se considere el caso de coste 1, $E_{c1}[x]$.

Si en cambio se considera el caso compuesto de coste 2, $E_{c2}[x]$, es decir, aquel en el que el coste al componer dos subfiltros es la suma de los costes de los dos subfiltros por separado, el sistema fragmentado es mejor en todos los casos como también se puede apreciar en la figura 3.11. En lo que sigue cuando hablemos de sistema compuesto nos estaremos refiriendo a los de coste 1 mientras no se diga lo contrario y de esta forma habrá que tener en cuenta que se está tomando el mejor caso del compuesto.

Aumentando el número de filtros simultáneos la tendencia es la misma como se puede observar en la figura 3.12 correspondiente a diversos sistemas con parámetros en los rangos $1 \leq O_1, O_2 \leq 10$, $1 \leq K_1 \leq O_1$ y $1 \leq K_2 \leq O_2$, para una tasa de llegadas $\lambda = 0.01$.

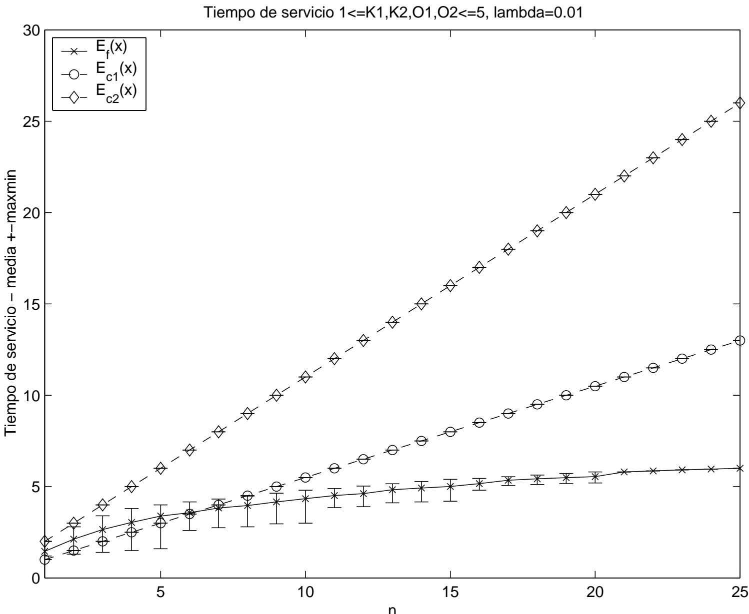


Figura 3.11: Tiempo de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 25 filtros

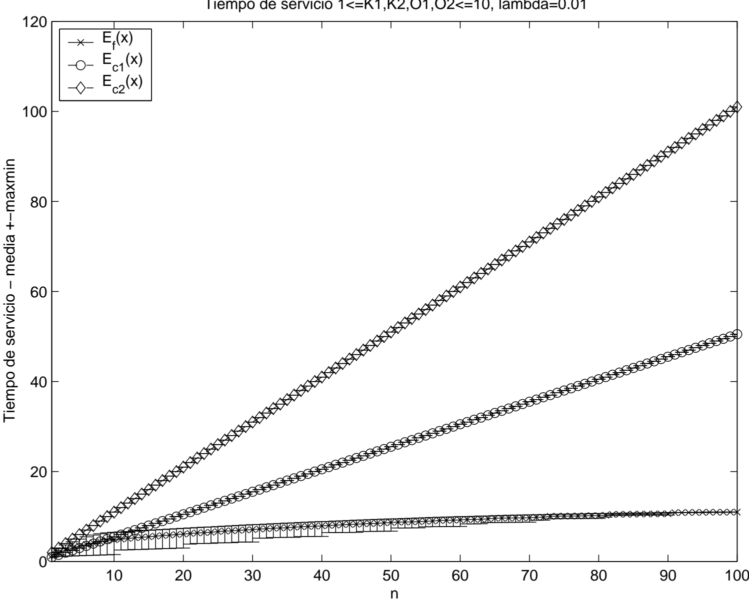


Figura 3.12: Tiempo de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 100 filtros

Por tanto, con un número de filtros medio y alto como ocurre con sistemas de monitorización, el sistema fragmentado muestra un menor tiempo de servicio y por tanto un menor factor de utilización con igual tasa de entrada que hace que se pueda soportar un mayor número de filtros simultáneos sobre el árbol de filtrado.

En la figura 3.13 se muestra la probabilidad de que el sistema fragmentado sea mejor que el compuesto según n, con $1 \leq K_1, K_2, O_1, O_2 \leq 100$ de los 10.000 filtros posibles ($K_1 = O_1$ y $K_2 = O_2$). Se muestra la parte de la gráfica correspondiente a un bajo número de filtros en los que se observa que el compuesto es mejor para menos de 10 filtros simultáneos y el sistema fragmentado es la mejor opción para mayor número de filtros.

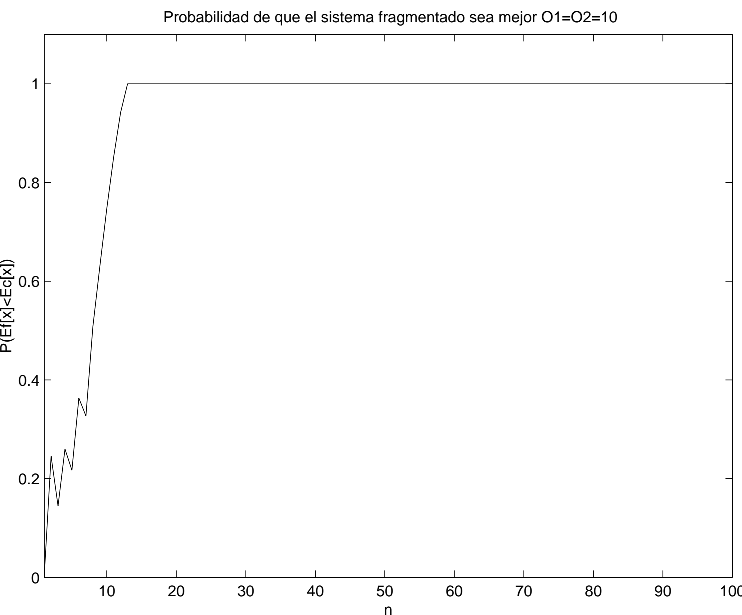


Figura 3.13: Probabilidad de que el sistema fragmentado sea mejor que el compuesto en función del número de filtros simultáneos n

3.2.2.4 Comparativa del tiempo de espera en cola

En la figura 3.14 se presenta el tiempo medio de espera en cola en función del número de filtros simultáneos para una tasa $\lambda = 0.01$ y sistemas con diversos parámetros en los rangos $1 \leq O_1, O_2 \leq 5$, $1 \leq K_1 \leq O_1$ y $1 \leq K_2 \leq O_2$. Las barras de error representan el máximo y mínimo del tiempo medio de espera en cola según la configuración de K_1, K_2, O_1 y O_2.

De nuevo, el tiempo medio de espera en cola es menor para el sistema compuesto de coste 1 y para un bajo número de filtros simultáneos, mientras que para un número mayor ($n > 6$ en la figura 3.14) el sistema fragmentado posee menor tiempo medio de espera en cola. Para un sistema de monitorización, este menor tiempo medio de espera en cola significa que se puede tener los datos de la medición de cierto intervalo en un tiempo menor. Si se compara con el sistema compuesto de coste 2 la mejora del sistema fragmentado es más evidente.

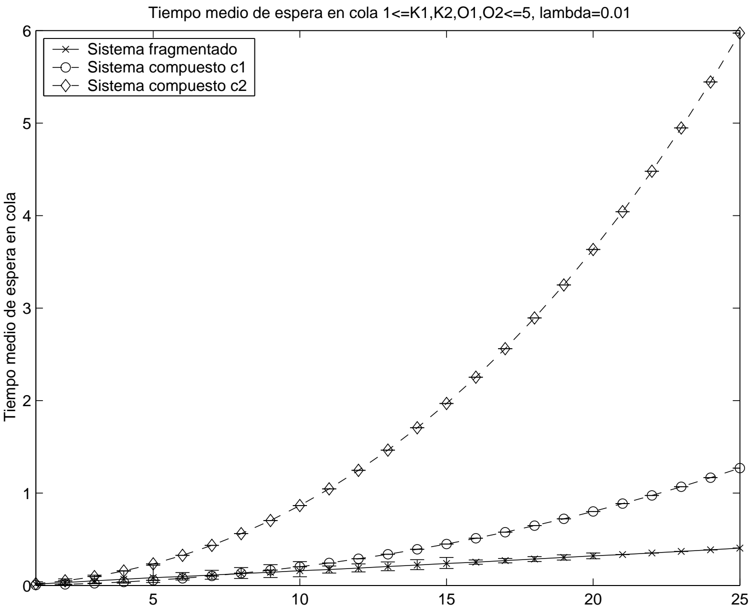


Figura 3.14: Tiempo de espera en cola para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 25 filtros

En la figura 3.15 se presenta también el tiempo medio de espera en cola para unos rangos de parámetros K_1, K_2, O_1 y O_2 que permite aumentar el número de filtros simultáneos hasta 100. Se observa el mismo comportamiento que en la figura 3.14 pero ahora más apreciable al tener un mayor rango del número de filtros simultáneos.

3.2.2.5 Condiciones de filtrado real

Vamos a suponer a continuación valores reales de K_1, K_2, O_1 y O_2 para filtrados que se puedan usar en la práctica. Estos resultados son totalmente dependientes de los filtrados que se escojan, pero tratan de estudiar un filtrado típico que se pueda definir en un sistema de monitorización.

3.2.2.5.1 Filtro IP - TCP Supongamos que se quiere realizar un filtro IP-TCP, es decir, un subfiltro que verifique que el paquete es IP y otro que es TCP. Dará combinaciones para establecer un pequeño número de filtros.

- IP: comprueba en la cabecera Ethernet que el campo Ethertype (2 bytes) sea 0x0800 (IP)

$$\begin{cases} O_1 = 2 & \text{básicamente suponemos que hay ARP e IP en esa red} \\ K_1 = 1 & \text{y que sólo estamos filtrando IP} \end{cases} \tag{3.40}$$

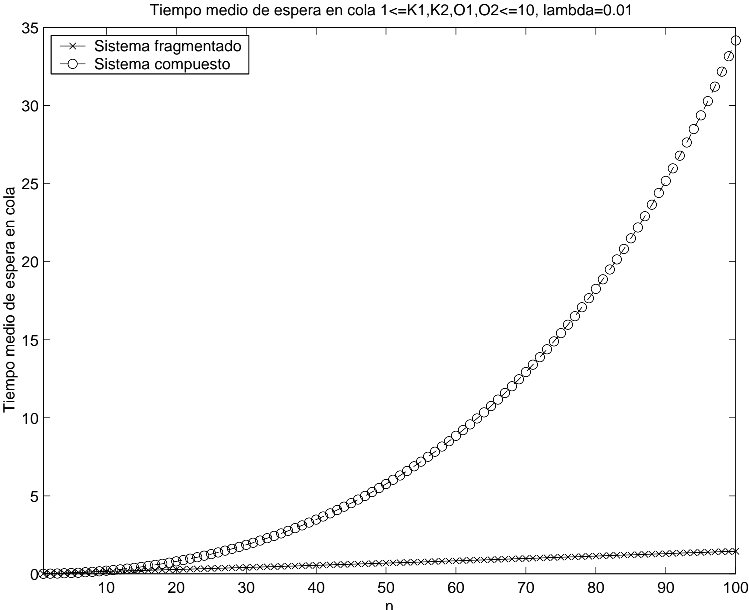


Figura 3.15: Tiempo de espera en cola para el sistema fragmentado y compuesto en función del número de filtros simultáneos n, hasta 100 filtros

- TCP: comprueba en la cabecera IP que el campo Protocolo (1 byte) sea 0x06 (TCP)

$$\begin{cases} O_2 = 3 & \text{básicamente suponemos que hay TCP, UDP e ICMP} \\ K_2 = 3 & \text{y que estamos filtrando en esos tres protocolos} \end{cases} \quad (3.41)$$

Con esta configuración sólo es posible establecer n=3 filtros obteniendo los valores que aparecen en la tabla 3.1 para $\lambda = 0.01$.

Parámetro	*Sistema fragmentado*	*Sistema compuesto coste 1*	*Sistema compuesto coste 2*
Media del Tiempo de Servicio	2	2	4
Tiempo medio de espera en cola	0.0476	0.0238	0.0972

Tabla 3.1: Comparativa para un caso real de filtrado IP-TCP

Es decir, los tiempos medios de servicio coinciden para el sistema fragmentado y compuesto (de coste 1), mientras que el tiempo medio de espera en cola es menor en el compuesto de coste 1. Al tratarse en esta ocasión de un número tan bajo de filtros el sistema fragmentado no sería la mejor opción.

3.2.2.5.2 Filtro IP origen - IP destino Supongamos ahora que se quiere realizar un filtro dirección IP origen - dirección IP destino, es decir, un subfiltro que verifique que el

paquete es de cierta dirección IP origen y otro que verifique que es de cierta dirección IP destino. Sería una matriz de tráfico típica entre parte de las direcciones IP de una subred. Por tanto, dará posibilidades para establecer un alto número de filtros simultáneos.

- Dirección IP origen: comprobar en la cabecera IP que el campo dirección IP origen (4 bytes) contenga cierto valor.

$$\begin{cases} O_1 = 65536 \text{ básicamente suponemos una red B} \\ K_1 = 256 \text{ y que nos interesan 256 direcciones IP origen} \end{cases} \tag{3.42}$$

- Dirección IP destino: comprobar en la cabecera IP que el campo dirección IP destino (4 bytes) contenga cierto valor.

$$\begin{cases} O_2 = 65536 \text{ básicamente suponemos una red B} \\ K_2 = 256 \text{ y que nos interesan 256 direcciones IP destino} \end{cases} \tag{3.43}$$

Con esta configuración, según el número n de filtros establecidos tendremos el tiempo medio de servicio mostrado en la figura 3.16.a). Se comprueba que a partir de 500 filtros (que son el 0.76% de los filtros que se pueden llegar a tener simultáneamente, $256 \cdot 256 = 65536$) el sistema fragmentado tiene un menor tiempo de servicio. En el tiempo de espera en cola se obtienen resultados paralelos como se muestra en la figura 3.16.b). Hay que hacer notar que este tipo de sistema compuesto de coste 1 sólo será posible sobre un procesador de 64 bits que pueda testear los dos campos de direcciones IP de manera simultánea.

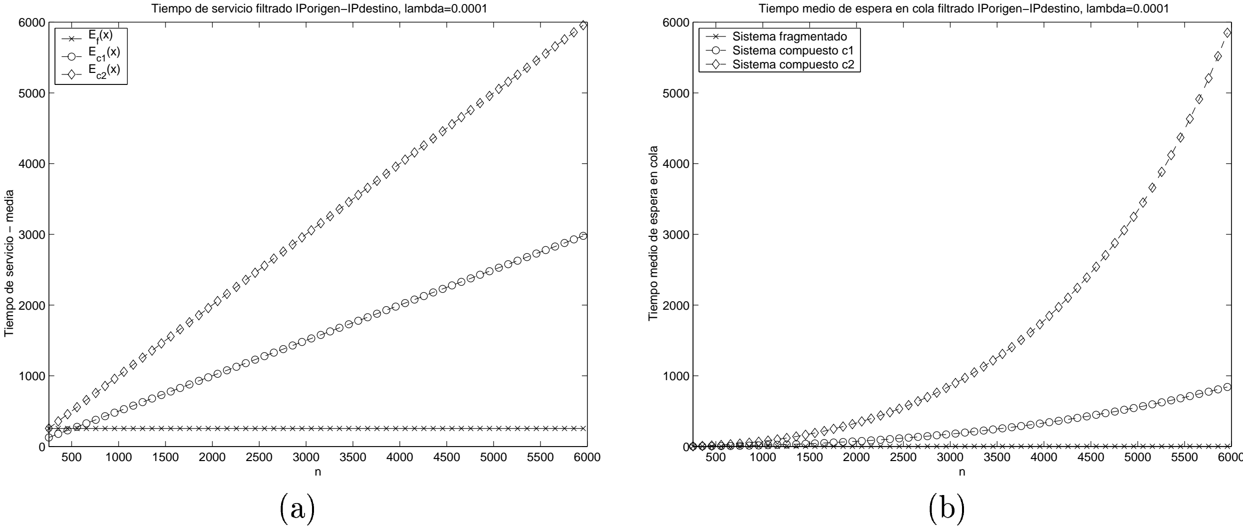


Figura 3.16: a) Tiempo medio de servicio para el sistema fragmentado y compuesto en función del número de filtros simultáneos n para un filtrado dirección IP origen - dirección IP destino b) Tiempo medio de espera en cola correspondiente

3.2.3 Generalización para más de dos niveles de filtrado

En el estudio realizado hasta el momento se ha visto cómo el sistema fragmentado ofrece mejores prestaciones en la mayoría de las ocasiones frente al mejor caso del sistema compuesto,

pero limitando el problema a un sistema de dos nodos. Podemos extender el razonamiento para 3 nodos y con ello generalizarlo para cualquier número de nodos. En la figura 3.17 se presenta el nuevo escenario con 3 niveles de árbol en el sistema fragmentado y uno único en el compuesto.

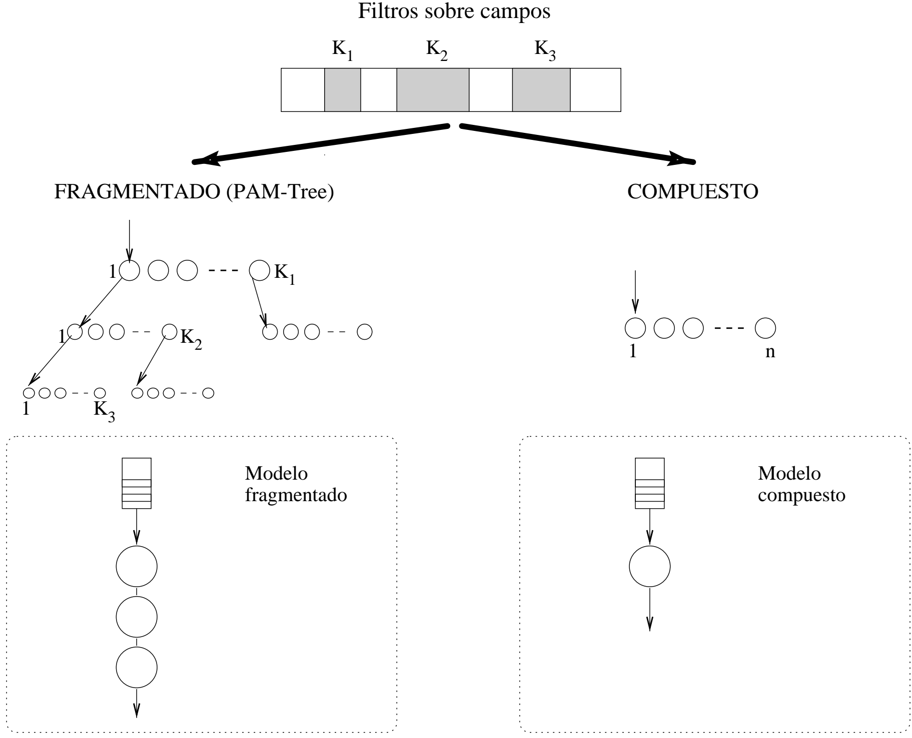


Figura 3.17: Modelo para el caso de 3 filtros

Procediendo como en el caso de dos subfiltros, para el caso compuesto suponemos que el subfiltro compuesto resultante tiene un coste de comparación idéntico al de la comparación de un subfiltro original. Hacer notar que estamos suponiendo de esta forma el mejor caso ya que no se pueden unir subfiltros a igual coste de manera indefinida: en algún momento se tendrá que convertir en dos o más comparaciones. Si se tienen n filtros simultáneos $max(K_1, K_2, K_3) \leq n \leq (K_1 \cdot K_2 \cdot K_3)$ y el tiempo medio de servicio queda como en el sistema de dos subfiltros:

$$E_c[x|n] = \frac{n+1}{2} \tag{3.44}$$

La expresión 3.44 será valida para cualquier número de niveles porque sólo depende del número total de filtros.

De la misma forma, para el caso fragmentado de 3 subfiltros, considerando x_S y x_N tiempos

de servicio cuando se verifica y no se verifica algún filtro respectivamente (apartado 3.2.2.2):

$$E_{f,3}[x|n] = (1-p_1)E[x_{N,1}|n] + p_1(E[x_{S,1}|n] + E[x_2|n])$$

$$E[x_2|n] = \sum_{a_2=max(1,(n_{imax}-(a_1-1)K_2))}^{min(K_2,n_{imax}-(a_1-1))} \Big((1-p_2)E[x_{N,2}|n] + p_2(E[x_{S,2}|n] + E[x_3|n])\Big)P_2(a_2)$$

$$E[x_3|n] = \sum_{a_3=max(1,(n_{imax}-(a_2-1)K_3))}^{min(K_3,n_{imax}-(a_2-1))} \Big((1-p_3)E[x_{N,3}|n] + p_3 E[x_{S,3}|n]\Big)P_3(a_3) \tag{3.45}$$

Generalizando para cualquier número de niveles de subfiltros M:

$$E_{f,M}[x|n] = (1-p_1)E[x_{N,1}|n] + p_1(E[x_{S,1}|n] + E[x_2|n])$$

$$E[x_2|n] = \sum_{a_2=max(1,(n_{imax}-(a_1-1)K_2))}^{min(K_2,n_{imax}-(a_1-1))} \Big((1-p_2)E[x_{N,2}|n] + p_2(E[x_{S,2}|n] + E[x_3|n])\Big)P_2(a_2)$$

$$\ldots$$

$$E[x_M|n] = \sum_{a_M=max(1,(n_{imax}-(a_{M-1}-1)K_M))}^{min(K_M,n_{imax}-(a_{M-1}-1))} \Big((1-p_M)E[x_{N,M}|n] + p_M E[x_{S,M}|n]\Big)P_M(a_M) \tag{3.46}$$

donde:

$$a_1 = K_1 \tag{3.47}$$

$$p_i = a_i/O_i \tag{3.48}$$

$$E[x_{N,i}|n] = a_i \tag{3.49}$$

$$E[x_{S,i}|n] = (a_i+1)/2 \tag{3.50}$$

$$P_i(a_i) = \frac{C_{K_i-1}(a_i-1)C_{(a_{i-1}-1)(K_i-1)}(n_{imax}-a_{i-1}-(a_i-1))}{C_{a_{i-1}(K_i-1)}(n_{imax}-a_{i-1})} \tag{3.51}$$

$$max(K_1, K_2, \ldots, K_M) \leq n \leq \prod_{j=1}^{M} K_j \tag{3.52}$$

En las expresiones 3.45, 3.46 y 3.51 se tiene n_{imax} como el máximo número de filtros que se pueden establecer entre subfiltros de nivel $i-1$ e i, y en todo caso será el mínimo de entre los filtros que quedan por establecer y el máximo número de filtros posible entre ambos niveles:

$$n_{imax} = min(n_i, a_{i-1} \cdot K_i) \tag{3.53}$$

donde n_i será:

$$n_i = \lfloor n_{i-1}/a_{i-1} \rfloor \tag{3.54}$$

Se puede demostrar por inducción está expresión del tiempo de servicio para M niveles. Supongamos que la expresión 3.46 se verifica para M niveles, vamos a comprobar que es aplicable para $M+1$ niveles. Para $M+1$ niveles tendremos:

$$\begin{aligned}
&E_{f,M+1}[x|n] = E[x_M|n] + \sum_{a_M} \sum_{a_{M+1}} E[x_{M+1}|n|a_M|a_{M+1}] P_{M+1}(a_{M+1}) P_M(a_M) \\
&E[x_{M+1}|n|a_M|a_{M+1}] = (1 - p_{M+1}) E[x_{N,M+1}|n] + p_{M+1} E[x_{S,M+1}|n] \\
&E[x_{N,M+1}|n] = a_{M+1} \\
&E[x_{S,M+1}|n] = (a_{M+1} + 1)/2
\end{aligned} \tag{3.55}$$

Con lo que queda la siguiente expresión para $M+1$ niveles:

$$E_{f,M+1}[x|n] = (1-p_1)E[x_{N,1}|n] + p_1(E[x_{S,1}|n] + E[x_2|n])$$

$$E[x_j|n] = \sum_{a_j=max(1,(n_{imax}-(a_{j-1}-1)K_j))}^{min(K_j,n_{imax}-(a_{j-1}-1))} \Big((1-p_j)E[x_{N,j}|n] + p_j(E[x_{S,j}|n] + E[x_{j+1}|n])\Big)P_j(a_j)$$

$$\ldots$$

$$E[x_M|n] = \sum_{a_M=max(1,(n_{imax}-(a_{M-1}-1)K_M))}^{min(K_M,n_{imax}-(a_{M-1}-1))} \Big((1-p_M)E[x_{N,M}|n]+p_M(E[x_{S,M}|n]+E[x_{M+1}|n])\Big)P_M(a_M)$$

$$E[x_{M+1}|n] = \sum_{a_{M+1}=max(1,(n_{imax}-(a_M-1)K_{M+1}))}^{min(K_{M+1},n_{imax}-(a_M-1))} \Big((1-p_{M+1})E[x_{N,M+1}|n]+p_{M+1}(E[x_{S,M+1}|n])\Big)P_{M+1}(a_{M+1}) \quad (3.56)$$

que coincide con la expresión esperada.

Haciendo uso de las expresiones de los momentos de la distribución hipergeométrica calculadas en las expresiones 3.22 y 3.23 se puede eliminar el sumatorio de la expresión del término j-ésimo tal y como se muestra en la siguiente expresión.

$$E[x_j|n] = \left(1 + \frac{E[x_{j+1}|n]}{O_j} + \frac{1}{2O_j}\left(1 - \left(\frac{K_1K_2-n}{K_1(K_2-1)-1}\right)\left(1 + \frac{(K_2-1)(n-K_1-1)}{K_1K_2-n}\right)\right)\right)\frac{n-K_1}{K_1} \quad (3.57)$$

De manera paralela, el segundo momento del tiempo de servicio es el siguiente:

$$E_f[x^2|n] = (1-p_1)E[x_{N,1}^2|n] + p_1(E[(x_{S,1}+x_2)^2|n])$$

$$E[(x_{S1}+x_2)^2|n] = E[(x_{S,1}^2|n] + E[x_2^2|n] + 2E[(x_{S,1}+x_2)|n] =$$

$$= E[x_{S,1}^2|n] + E[x_2^2|n] + 2E[x_{S,1}|n]E[x_2|n]$$

$$E[x_2^2|n] = \sum_{a_2=max(1,(n_{imax}-(K_1-1)K_2))}^{min(K_2,n_{imax}-(K_1-1))} \Big((1-p_2)E[x_{N,2}^2|n] + p_2(E[(x_{S,2}+x_3)^2|n])\Big)P_2(a_2) \quad (3.58)$$

Generalizando la expresión para M niveles, y haciendo uso de los momentos calculados para la distribución hipergeométrica:

$$
\begin{aligned}
&E_f[x^2|n] = (1-p_1)E[x_{N,1}^2|n] + p_1(E[x_{S,1}^2|n] + E[x_2^2|n] + 2E[x_{S,1}|n]E[x_2|n])\\
&E[x_{N,i}^2|n] = a_i^2\\
&E[x_{S,i}^2|n] = \frac{(a_i+1)(2a_i+1)}{6}\\
&E[x_i^2|n] = \sum_{a_i=max(1,(n_{imax}-(a_{i-1}-1)K_i))}^{min(K_i,n_{imax}-(a_{i-1}-1))} \big((1-p_i)E[x_{N,i}^2|n]+p_i(E[x_{S,i}^2|n]+E[x_{i+1}^2|n]+2E[x_{S,i}|n]E[x_{i+1}|n])\big)P(a_i)=\\
&= \left(\tfrac{-2}{3O_i}\right)E[a_i^3] + \left(1+\tfrac{1}{2O_i}+\tfrac{E[x_{i+1}|n}{O_i}\right)E[a_i^2] + \tfrac{1}{O_i}\left(\tfrac{1}{6}+E[x_{i+1}^2|n]+E[x_{i+1}|n]\right)E[a_i]\\
&i = 2\ldots M, \quad E[x_{M+1}^2|n] = 0
\end{aligned}
\tag{3.59}
$$

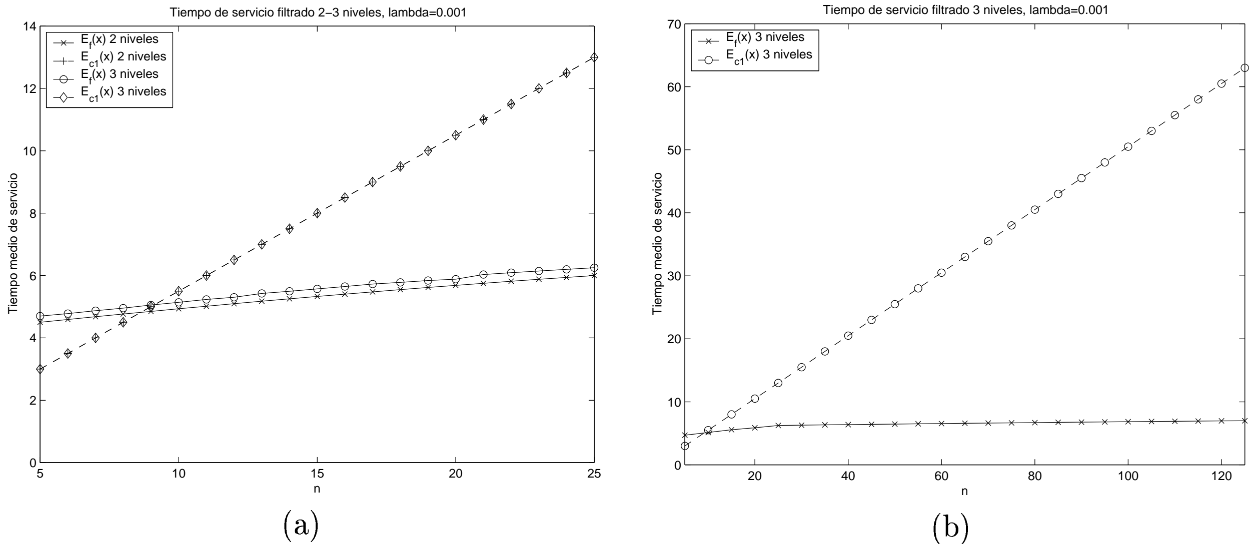
Estas expresiones se demuestran por inducción como lo ya visto para el cálculo del primer momento.

Si repetimos el cálculo del tiempo de servicio según estas expresiones para 3 subfiltros, los puntos de corte del número de filtros donde el sistema fragmentado empieza a ser mejor se desplazan a valores un poco mayores (figura 3.18), ya que se ha tomado un valor optimista del caso compuesto en el que componer 3 subfiltros tiene un coste por comparación idéntico a la de un subfiltro.

En la figura 3.18.a) se compara el tiempo medio de servicio para un filtrado de dos niveles ($K_1 = K_2 = 5, O_1 = O_2 = 10$) con otro filtrado de 3 niveles e iguales parámetros ($K_1 = K_2 = K_3 = 5, O_1 = O_2 = O_3 = 10$). Se comprueba como ya se ha comentado que el tiempo medio de servicio se incrementa levemente debido a la cada vez menor probabilidad de pasar al tercer nivel. En la figura 3.18.b) se presenta el tiempo medio de servicio del filtrado a 3 niveles para un mayor número de filtros observándose para el sistema fragmentado (PAM-Tree) como apenas crece con el número de filtros. En la figura 3.19 se presenta el tiempo medio de espera en cola correspondiente.

3.2.4 Conclusiones

Mediante el modelo propuesto para el estudio analítico del coste de un sistema fragmentado frente a uno compuesto, se ha observado claramente la supremacía del sistema fragmentado conforme se eleva el número de filtros simultáneos. Por tanto, el algoritmo del árbol de filtrado PAM-Tree, que hace uso de esta fragmentación, es la mejor alternativa en cuanto se tiene un número elevado de filtros que es lo habitual en un sistema de monitorización.



(a) (b)

Figura 3.18: Tiempo medio de servicio para a) sistema fragmentado y compuesto en función del número de filtros simultáneos n para un filtrado de 2 y 3 niveles b) sistema de 3 niveles y mayor n

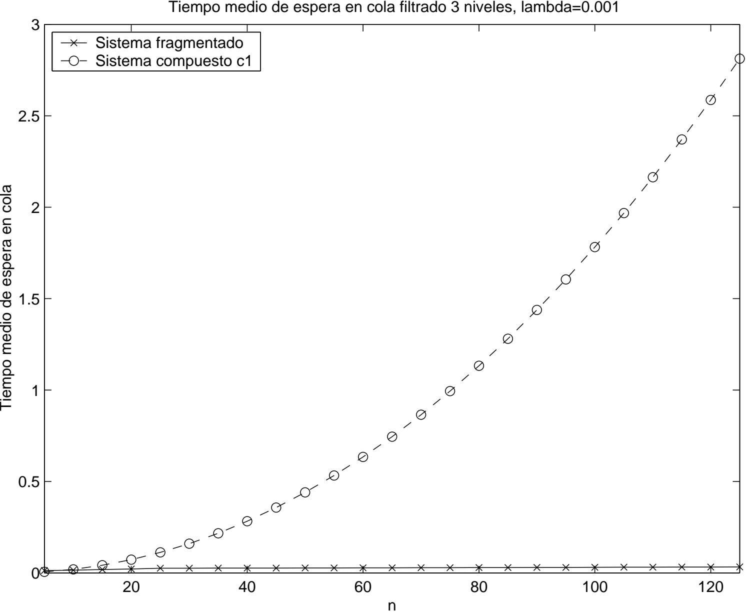


Figura 3.19: Tiempo medio de espera en cola para 3 niveles $K_1 = K_2 = K_3 = 5, O_1 = O_2 = O_3 = 10$

3.3 Comparativa experimental

3.3.1 Introducción

En el presente apartado se analizan las características de PAM-Tree frente a otras alternativas de algoritmos de filtrado existentes en la actualidad como BPF.

En nuestro caso vamos a hacer uso de la implementación del packet filter BPF sobre Linux denominada Linux Socket Filter (LSF) [89] debido a que se ha utilizado este sistema operativo para la implementación de sistemas de monitorización mediante PAM-Tree. Además, la implementación del packet filter de LSF a nivel de kernel es un porte del BPF y se ha comprobado su similar eficiencia como se mostrará posteriormente.

Se ha utilizado la librería Libpcap [16, 90] como interfaz para acceso a las funcionalidades de filtrado de LSF. Las medidas se han realizado con la versión 0.4-16 de libpcap y con algunas versiones posteriores 0.5.2 y 0.6.2 [17] con las que las diferencias son mínimas, básicamente se añade soporte de nuevos protocolos, mayores opciones de funcionamiento y corrección de *bugs*, pero la arquitectura de filtrado o el lenguaje de definición de filtros se mantiene. Estas versiones sí incorporan, a diferencia de la versión 0.4 original, el soporte en Linux de dos modos para leer paquetes en modo *raw* (capturar paquetes completos con cabeceras de nivel de enlace): el antiguo SOCK_PACKET y el nuevo SOCK_RAW (de kernels 2.2.x). En la implementación original de Libpcap v0.4 [16] el filtrado lo tenía que realizar la propia librería Libpcap ejecutándose a nivel de usuario porque el kernel no proporcionaba esa funcionalidad. Con versiones más recientes de Libpcap [17] y aparición de soporte de filtrado a nivel de kernel mediante Linux Socket Filter de los kernels 2.2.x, Libpcap deja en manos del kernel las labores de filtrado igual que la implementación originaria sobre BSD con BPF.

3.3.2 Midiendo el consumo de CPU del Kernel

Debido a que tanto el filtrado de LSF como la llamada al sistema para leer paquetes desde un proceso de usuario se ejecutan a nivel de kernel, aparece el problema de medir el uso de CPU de manera fiable. Los sistemas operativos BSD y Linux no reportan una medida del consumo total de CPU (kernel+usuario), y muchas veces contabilizan ese consumo a procesos de usuario que pueden no estar relacionados con las tareas que realiza el kernel [91]. Esto se debe a que el kernel no está diseñado para medirse a sí mismo y sólo a los procesos de usuario que de él derivan. Los diferentes kernels Linux testeados 2.2.12, 2.2.14, 2.2.18 y 2.4.1 tienen la misma problemática, y aunque a partir del 2.2.18 el kernel empieza a reflejar el consumo de CPU de ciertos eventos del sistema, la parte de red que nos incumbe no se abarca. Por tanto hemos tenido que acudir a diferentes métodos para la medición de CPU que nos permitirá comparar ambas implementaciones PAM-Tree y LSF.

Por una parte, existen multitud de *debuggers* a nivel de kernel de Linux, pero no reportan la información que deseamos de ocupación de la CPU. Más bien se necesita una herramienta que permita tracear los eventos del kernel y su duración.

Otra posibilidad es el *The User-Mode Linux Kernel* [92] que consiste también en una serie

de parches que permiten correr dentro de un kernel otro kernel como si fuese un proceso de usuario y por tanto medible con las herramientas habituales (*top*, *strace*,..). Sin embargo, el soporte de red de este kernel secundario recibe copias de los paquetes desde el primario, lo que unido a continuos cambios de contexto desvirtúa en demasía la medida.

El *Linux Trace Toolkit* [93] es un parche del kernel 2.2.14 que nos permite registrar eventos del sistema operativo y por tanto, en principio, conocer el consumo de CPU por el kernel. El número de puntos de control es muy pequeño, y apenas nos da información de la interrupción producida por la tarjeta de red con la llegada del paquete, la finalización y la atención a esa interrupción, la interrupción del reloj, el procesado del *Bottom Half* [75] de la parte de red, etc. Esta será la opción utilizada para los sistemas Linux, añadiendo puntos de control al kernel en concreto en las entradas/salidas de IRQs, llamadas al sistema y *Bottom Halves*. La sobrecarga que introduce en el sistema es escasa, inferior al 1% en todo caso, por lo que el error cometido por otros factores como la ejecución de tareas de mantenimiento del kernel será mayor.

Para la medida del consumo de CPU en sistemas BSD hemos optado por el método de crear un programa de baja prioridad pero que haga uso intensivo de la CPU (por ejemplo, aumento continuo de un contador). Si partimos de un sistema en reposo y suponiendo el uso de CPU nulo por parte de otros procesos y del kernel, lo que cueste una iteración de ese programa t_0 será la referencia de uso de CPU de Kernel 0. Si el kernel empieza a consumir CPU, el programa se ralentizará, y con el nuevo tiempo t_k que dure una iteración podremos saber el consumo de CPU aproximado del kernel: $\%CPU = 100\,\frac{t_k - t_0}{t_k}$. Esta medida es fiable hasta un consumo del kernel de aproximadamente un 90%, porque es el momento a partir del cual las gráficas pierden la linealidad observada a consumos más bajos.

3.3.3 Sistemas de monitorización PAM-Tree y LSF

Para realizar la comparativa entre ambos sistemas se ha implementado un sistema de monitorización simple sobre PAM-Tree y LSF. Vamos a suponer un filtrado del tipo “paquetes UDP y puerto destino X”, compuesto por tanto por los siguientes subfiltros:

- subfiltro del campo *Ethertype* de la cabecera Ethernet - que sea un paquete IP.
- subfiltro del campo *Protocol* de la cabecera IP - que sea un paquete UDP.
- subfiltro del campo *puerto destino* de la cabecera UDP - que sea un paquete con cierto puerto X destino.

Como parámetro a monitorizar asociado a este filtro, nos va a interesar llevar cuenta de los bits por segundo de los paquetes que verifiquen este filtro dentro del sistema de monitorización que utilice por una parte el algoritmo PAM-Tree y por otra LSF.

En las pruebas se va a tratar con filtros simultáneos que podrán ser:

- iguales: indicará que se filtra el mismo puerto destino para todos los filtros para comprobar si se produce reutilización de bloques de subfiltrado.

- diferentes: indicará que se filtra diferente puerto destino para cada filtro.
- verificados: cuando el tráfico sobre la red es tal que se cumplen con cada paquete todos los filtros definidos.
- no verificados: cuando el tráfico sobre la red es tal que no se verifica ninguno de los filtros definidos con ninguno de los paquetes. Se tratará de tráfico UDP por lo que aunque no se verifiquen los filtros completos sí lo harán los subfiltros IP-UDP.

Implementar esta funcionalidad con PAM-Tree es muy sencillo. El sistema de monitorización sobre PAM-Tree se limita a tener capacidades para añadir, actualizar y eliminar parámetros, recayendo la mayoría de trabajo sobre el propio algoritmo de filtrado PAM-Tree.

Para poder imitar estas funcionalidades mediante LSF se hace necesario implementar un sistema de monitorización que sondee los filtros establecidos según una disciplina Round-Robin ya que los filtros se encuentran definidos de manera independiente. Cuando un filtro se verifica, se recibe el paquete en el sistema de monitorización que podrá entonces actualizar el parámetro asociado al filtro. Mediante el Linux Trace Toolkit se pueden analizar los eventos producidos a nivel de kernel y el esquema de funcionamiento de LSF básicamente consiste en:

- Recepción de IRQ por parte de la tarjeta de red y atención del *device driver* que se limita a añadir el paquete a buffers del kernel.
- Salida de la IRQ.
- Ejecución del Bottom Half correspondiente a las tareas de red (BHnet), entre ellas el filtrado de LSF en el que se procesa el paquete tantas veces como filtros se hubieran definido.

Los experimentos se han realizado en una red Fast Ethernet aislada con los sistemas de monitorización funcionando en un ordenador PentiumII 350Mhz y Linux 2.2.14. A continuación se van a comparar ambos algoritmos de filtrado aplicados a sendos sistemas de monitorización.

3.3.4 Comparativa PAM-Tree / LSF

En primer lugar se presenta en la figura 3.20 una comparativa del consumo de CPU del sistema de monitorización basado en PAM-Tree con la implementación sobre LSF, en función del número de filtros establecidos de diferente naturaleza y para una carga de red constante de 3Mbps y paquetes de 1500 bytes de tamaño.

En el PAM-Tree se observa como:

- El filtrado es independiente del número de filtros cuando son iguales, se verifiquen o no, debido a la reutilización de bloques de subfiltrado. Cuando los filtros son diferentes el consumo de CPU crece con el número de subfiltros debido al coste de los subfiltros de tercer nivel (puerto destino) que no se pueden reutilizar y han de ser diferentes para cada filtro porque se está interesado en diferentes puertos. El resultado se asimila al obtenido en el modelo fragmentado propuesto en el capítulo anterior.

- El sólo hecho de verificar los filtros, aunque sean iguales, supone un leve incremento de CPU debido al coste de actualizar los contadores de todos los filtros que se verifican a la vez por tratarse del mismo filtro.
- El hecho de tener filtros diferentes supone una mayor carga debido al procesado independiente de los subfiltros no comunes, en nuestro caso el puerto destino. Es dependiente del número de filtros porque no se puede reutilizar el último subfiltro que es el que difiere en todos los filtros.
- Existe una carga de CPU base, que aparece en cuanto se tenga un filtro definido, debido al coste de la copia del paquete al nivel de usuario.

En cuanto a LSF:

- El filtrado depende del número de filtros, independientemente de que sean o no iguales, por lo que se demuestra que no se reutilizan bloques de subfiltrado. Este crecimiento es lineal asimilándose al resultado del modelo compuesto estudiado en el capítulo anterior.
- El hecho de verificar los filtros, aunque sean iguales, supone un incremento de CPU muy fuerte ya que al coste del filtrado se le une el coste de copia del paquete desde el nivel de kernel al de usuario por cada filtro verificado. Esto hace que se empiecen a perder paquetes a partir del punto en el que no se presenta la gráfica de filtros iguales verificados.
- El hecho de tener filtros diferentes supone un coste similar al de filtros iguales debido a la falta de optimización que realiza LSF.

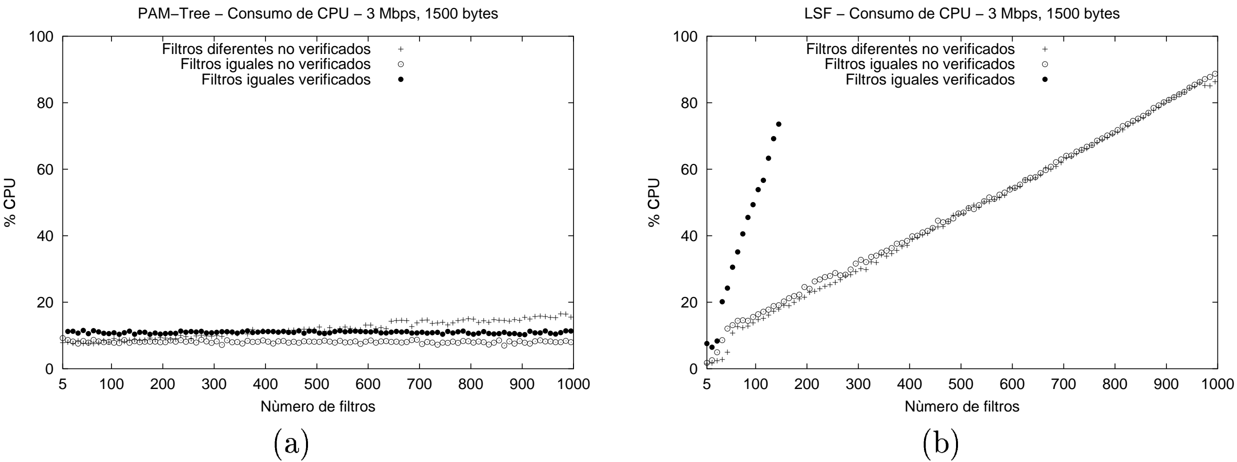


Figura 3.20: Consumo de CPU en función del número de filtros y su naturaleza a) PAM-Tree b) LSF

En la figura 3.21 se presentan los resultados de la comparativa en función de la carga de la red para una configuración fija de 200 filtros y paquetes de tamaño 1500 bytes. Los filtros no se verifican.

En ambos casos el consumo de CPU es lineal con la carga de tráfico en la red como era de prever, ya que supone un crecimiento lineal del número de paquetes a filtrar por unidad de tiempo. En el caso del PAM-Tree el coste es levemente superior para filtros diferentes, al tener menor reutilización de subfiltros.

En el caso de LSF el consumo de CPU es lineal e independiente de que los filtros sean iguales o diferentes como se ha visto antes, debido a la no reutilización de subfiltros.

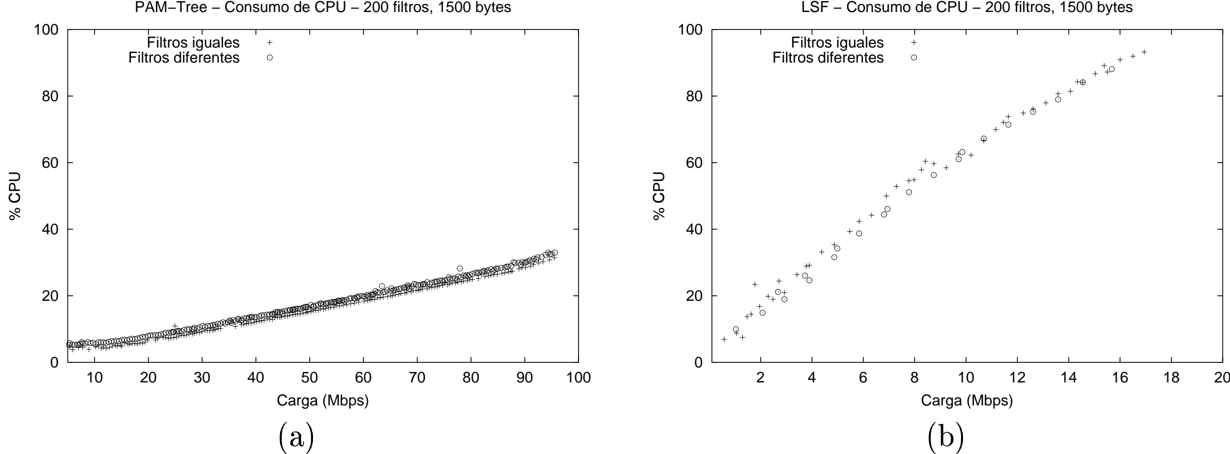


Figura 3.21: Consumo de CPU en función de la carga de red a) PAM-Tree b) LSF

En cuanto al efecto del tamaño del paquete, en la figura 3.22 se muestra la eficiencia de cada sistema con diferente tamaño de paquete y número de filtros. La figura corresponde a una carga de red de 3 Mbps y sin verificar ningún filtro.

Para PAM-Tree el consumo de CPU a igual tasa depende del tamaño de paquete. A más paquetes por segundo (paquete pequeño) el coste de la labor de filtrado se incrementa.

En el caso del LSF como los paquetes no verifican los filtros, se elimina el coste extra por copias de memorias correspondientes al paso del paquete al nivel de usuario que sería dependiente del tamaño de los paquetes. Por tanto, el consumo de CPU depende directamente de la tasa de paquetes por unidad de tiempo, y para una carga de red constante dependerá del tamaño de paquete. A más paquetes por unidad de tiempo (a igual carga de red corresponderá a la de paquetes más pequeños) el coste del filtrado se incrementa.

3.3.5 Implementación BPF

Si bien se ha revisado la implementación de BPF sobre Linux (LSF) en comparación con PAM-Tree, la implementación BPF sobre BSD se comporta de manera muy similar. Para demostrarlo en la figura 3.23 se muestra la comparativa de consumo de CPU para la implementación BPF sobre FreeBSD3.2 [94] frente a PAM-Tree de funcionalidades recortadas en la misma plataforma. El entorno de medida es el comentado anteriormente, con carga constante de 3Mbps y tráfico que verifica de manera aleatoria uniforme los filtros definidos. En este caso se observa:

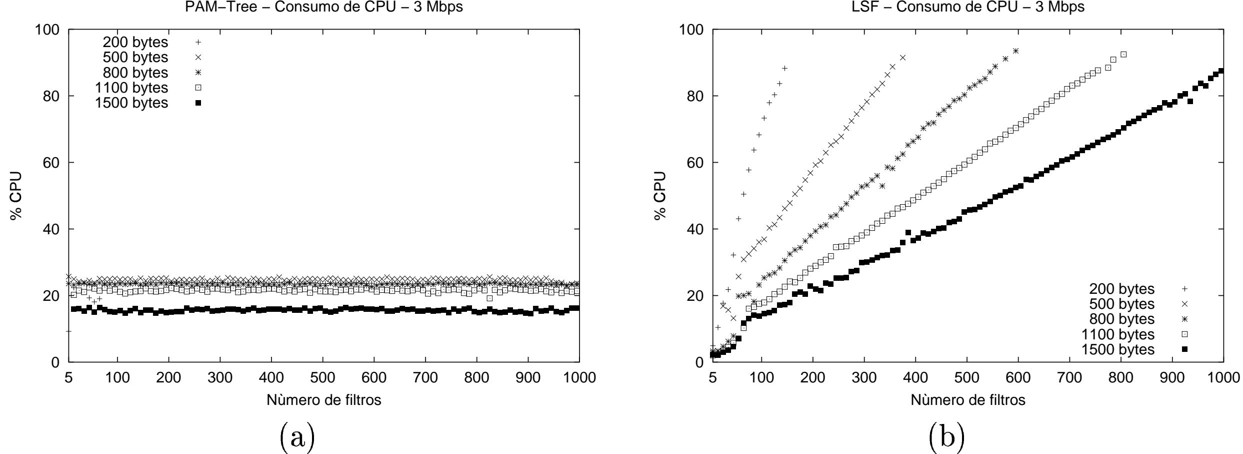


Figura 3.22: Consumo de CPU en función del tamaño de paquete a) PAM-Tree b) LSF

- La misma tendencia que con el LSF, de mejora con pequeño número de filtros.
- Existe un buffer intermedio entre el filtro y el proceso de usuario, de manera que los paquetes no son pasados al proceso de usuario hasta que se llena ese buffer o se cumple un *timeout* configurable. De esta manera se reduce el número de llamadas al sistema y con ello la sobrecarga del mismo.
- Limitación a 256 filtros simultáneos debido a la necesidad de asociar cada filtro con un dispositivo */dev/bpfXXX* cuyo número viene limitado por el *minor number* del sistema de ficheros de los sistemas UNIX.

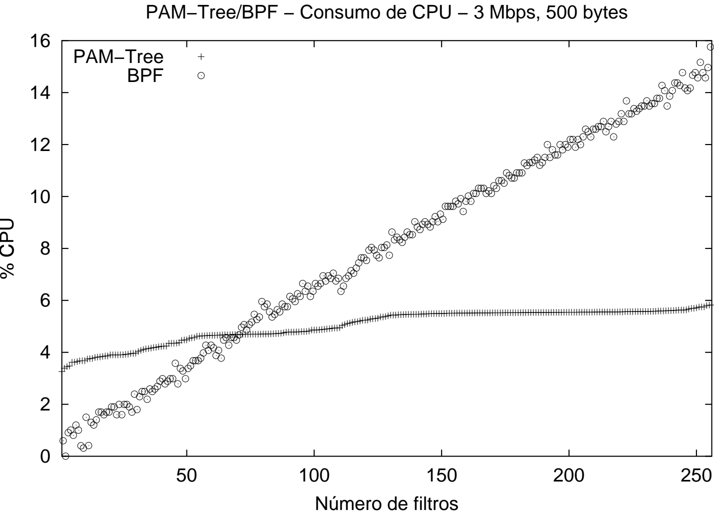


Figura 3.23: Comparativa de consumo de CPU PAM-Tree/BPF en un sistema FreeBSD frente al número de filtros establecidos

Se comprueba aquí una vez más que el BPF como el resto de packet filters son útiles para aquellas aplicaciones en las que se rechacen más paquetes que los que se acepten, lo cual es contrario a la filosofía de un sistema de monitorización que ha de ser capaz de procesar todos los paquetes.

3.3.6 Conclusiones

El presente capítulo ha aportado un modelo analítico para el algoritmo de filtrado PAM-Tree. Esta modelización no se encuentra habitualmente en el estudio de sistemas de filtrado.

Del estudio anterior, PAM-Tree resulta la mejor opción cuando se desea monitorizar redes de comunicaciones con cientos y miles de filtrados simultáneos, que es lo habitual en sistemas de monitorización.

En el caso de verificar filtros o de estudiar el consumo de CPU en función del tamaño de los paquetes el algoritmo PAM-Tree es superior. Sin embargo, LSF/BPF son la mejor opción cuando existe un bajo número de filtros que no se verifican debido a que evita la copia del paquete del nivel de kernel al nivel de usuario, que es lo que impone la sobrecarga base del PAM-Tree. Pero en un sistema de monitorización real con muchos filtros definidos, los paquetes verificarán uno u otro filtro por lo que lo importante es entonces optimizar el filtrado que es lo que consigue el PAM-Tree.

Capítulo 4

Implantación industrial

4.1 Introducción

El análisis realizado del PAM-Tree prueba su eficiencia en la monitorización de redes de datos. Sin embargo, la prueba definitiva es la implementación del sistema y su aplicación sobre una red real. Una de las grandes aportaciones de este trabajo ha sido la implementación de la arquitectura de filtrado propuesta en dos herramientas de monitorización de red que además están siendo utilizadas en la actualidad. Fruto de todo ello es la titularidad de una patente con utilización industrial.

Las herramientas desarrolladas han sido PROMIS y MONET. El sistema de monitorización PROMIS ha sido desarrollado para la empresa de fabricación de automóviles Volkswagen Navarra S.A. Se trata de un sistema de monitorización distribuido con una sonda que implementa una versión recortada de PAM-Tree, con flexibilidad limitada a filtrados fijos a determinados niveles de protocolos. Posteriormente se ha desarrollado el sistema MONET para la empresa adjudicataria de la demarcación de cable para Navarra RETENA S.A. Este sistema es una evolución del anterior, implementando todas las funcionalidades del PAM-Tree y, por tanto, ofreciendo mayor flexibilidad en el funcionamiento global de la herramienta.

Como consecuencia del algoritmo de filtrado propuesto en este trabajo, se posee una patente nacional titulada *"Sistema de monitorización de redes de comunicaciones empleando un método jerárquico de análisis de tráfico y almacenamiento de medidas de tráfico"* [95] bajo explotación en las herramientas comentadas.

Estos desarrollos han servido para comprobar la eficiencia de PAM-Tree en entornos de red más o menos cargados, con diferente número de máquinas, con aplicaciones industriales u ofimáticas, junto con la incorporación de nuevos protocolos, su utilización en un sistema de monitorización distribuido y por diferentes usuarios. También se ha podido ver la utilidad de la flexibilidad en la definición de filtros, que hace que se puedan adaptar a cualquier situación global o problema puntual. Finalmente ha permitido comprobar la posibilidad de implementación de un sistema de monitorización sobre una plataforma PC sin necesidad de acudir a soluciones hardware comerciales de alto coste.

4.2 Sistema PROMIS

La herramienta PROMIS desarrollada durante los años 1998-2000 ha sido la primera implementación del algoritmo PAM-Tree. Se enmarca dentro del proyecto OTRI (Oficina de Transferencia de Resultados de Investigación) de la Universidad Pública de Navarra con la empresa Volkswagen Navarra S.A titulado *"Analizador de redes PROMIS para redes de área extensa"* con el Dr. Javier Aracil Rico y el Dr. Jesús Villadangos Alonso como investigadores principales. Esta primera versión se caracterizaba principalmente por la limitación de los subfiltros a determinados campos de la jerarquía de niveles de protocolos, pero se ajustaba a las necesidades requeridas por la empresa [96, 97, 98, 99, 100].

PROMIS es un sistema de monitorización distribuido compuesto por dos elementos principales: sondas y consolas. Las sondas, o agentes de monitorización, se colocan en puntos de la red de comunicaciones donde se quiere realizar el análisis de monitorización. La consola, o gestor, se puede colocar en cualquier punto de la red y no es mas que un interfaz de usuario para la presentación de la información de monitorización obtenida por las sondas. En este esquema toda la carga de análisis y filtrado recae en las sondas mientras que la consola permite el acceso unificado a la información procedente de todas las sondas y también sirve para seleccionar los parámetros a monitorizar en cada sonda.

4.2.1 Funcionalidades del sistema

El sistema de monitorización PROMIS es capaz de ofrecer cuatro grandes tipos de información:

- Monitorización en tiempo real. Se recibe en la consola el valor de un parámetro Q tan pronto como se haya calculado. Es decir, cada intervalo, determinado por la resolución elegida, se recibe el valor del parámetro en la consola. Por tanto, tendremos su evolución temporal $Q(t)$.
- Capturas. Dado un instante temporal de comienzo y finalización, almacena en el disco de la sonda la evolución del parámetro seleccionado con la resolución deseada. Posteriormente, la consola podrá solicitar esta información.
- Informes periódicos. Son una serie de capturas predefinidas que se repiten cada hora, día, semana, mes o año.
- Alarmas. Se pueden definir umbrales de un parámetro que una vez superados activen una alarma que se reciba de inmediato en la consola. Se utilizarán para indicar condiciones anormales en la red.

Además posee otras funcionalidades como la detección de direcciones IP duplicadas (que no haya dos direcciones MAC compartiendo la misma dirección IP) o una base de datos de máquinas vistas en la red con nombre simbólico asociado.

Todos estos tipos de informaciones se pueden realizar a partir de cualquier parámetro de monitorización. Un parámetro consiste en una secuencia de subfiltros a distintos niveles de

protocolos, junto al valor de bits o paquetes que se quiera contabilizar de los paquetes que superen dichos subfiltros y un intervalo de muestreo que fijará la resolución de nuestra medida. Un tipo especial de parámetro será la captura de paquetes completos de la red para su posterior visualización mostrando al usuario la decodificación del mismo campo a campo.

Un aspecto importante es el soporte de protocolos SNA (*Systems Network Architecture* de IBM) además de TCP/IP debido a su fuerte presencia en la red de comunicaciones para la que se implementó la herramienta.

4.2.2 Sonda

La sonda [101] es el elemento fundamental del sistema que, colocado en un segmento de red, es el encargado de procesar todos los paquetes que circulen por él y llevar cuenta de los parámetros de monitorización solicitados por la consola. Por ello, el algoritmo de filtrado y análisis de paquetes debe estar optimizado para que haga uso de la menor cantidad de recursos posibles, así como que la velocidad de análisis sea suficiente para el estudio de todo el tráfico que circule por la red.

Se ha desarrollado en lenguaje C sobre una plataforma PC (AMD K6-2 450Mhz, 64MB RAM, 1.5GB HD, tarjeta red 10/100-BaseT) con sistema operativo Linux (RedHat 6.0) y soporte de redes Ethernet 10/100. La arquitectura interna de la sonda es la mostrada en la figura 4.1. En ella, cada bloque representa un proceso. Todos los procesos se ejecutan concurrentemente y se intercomunican a través de un sistema de memorias compartidas. Estos procesos realizan las funcionalidades de la arquitectura básica compuesta por subsistema de captura, filtrado y servidor.

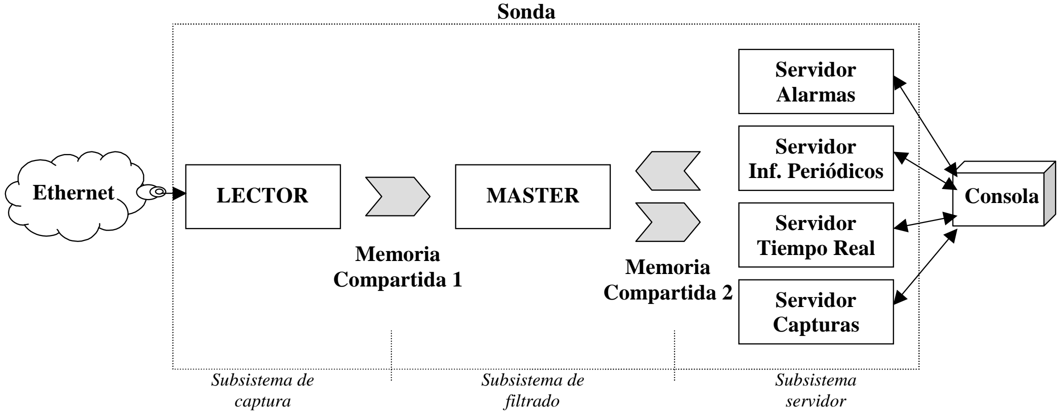


Figura 4.1: Estructura en bloques de la sonda PROMIS

El proceso Lector realiza la tarea del subsistema de captura. Básicamente coloca el interfaz de red en modo promiscuo, lee todos los paquetes de la red y los coloca junto con su timestamp y tamaño en la primera memoria compartida.

El proceso Master realiza la tarea del subsistema de filtrado. Como ya se ha comentado se basa en una estructura en árbol recortada, en la que los subfiltros se limitan a testeos sobre

campos completos de determinados niveles de protocolos. Este proceso lee los paquetes de la primera memoria compartida y actualiza los contadores de los parámetros monitorizados en la segunda memoria compartida. En la figura 4.2 se presenta la estructura en árbol con parte de los filtrados disponibles. Se puede observar una estructura mucho más rígida que el PAM-Tree y la gran dependencia de los protocolos, porque la inteligencia para el soporte de protocolos está implementada sobre el propio árbol.

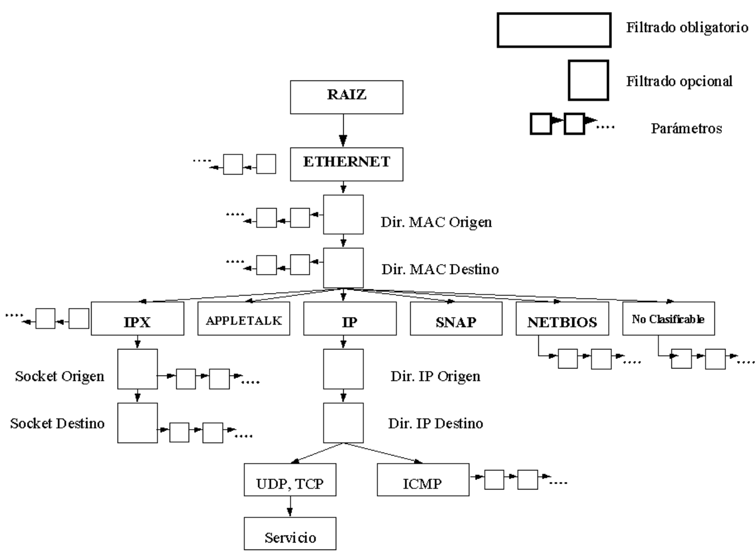


Figura 4.2: Árbol de filtrado en PROMIS

El subsistema servidor está formado por los 4 procesos servidores de la figura 4.1 y es el encargado de muestrear los parámetros de monitorización que se encuentran en la segunda memoria compartida con el intervalo de muestreo deseado, bien sea para mandarlo directamente a la consola o para almacenarlo localmente en la sonda. El servidor de tiempo real recibe conexiones de la consola a la que le manda el parámetro monitorizado en cuanto se disponga de él, por ejemplo si se monitorizan bits por segundo, la sonda mandará a la consola el valor de bits medidos cada segundo. El servidor de capturas almacena los parámetros monitorizados en la sonda para su posterior transferencia a la consola cuando ésta los solicite. El servidor de alarmas se encarga de comprobar la superación de los umbrales definidos para las alarmas y en tal caso transmitir la alarma a la consola. Finalmente el servidor de informes periódicos confecciona dichos informes, que no son mas que capturas que se repiten cíclicamente, que posteriormente se podrán acceder desde un interfaz web.

4.2.3 Consola

La consola [101] es el elemento que centraliza toda la información que generan el conjunto de sondas que forman el sistema de monitorización distribuida. De esta forma, desde una consola se puede tener una visión global de todo el sistema, así como una visión detallada de cada una de las subredes y de las máquinas que las componen. Aunque existan diferentes consolas en una misma red que interactuen con las sondas, se sigue teniendo la visión global de toda

la red con toda la información que generan todas las sondas, ya que las consolas comparten una base de datos única donde almacenan la información que reciben de las sondas.

Se han implementado dos tipos de consola. Por un lado una desarrollada con VisualC++ sobre WindowsNT que frece todas las funcionalidades excepto la visualización de informes periódicos, y por otro lado un interfaz web que permite el acceso a información restringida de monitorización como los informes periódicos desde cualquier navegador web (ver figuras 4.3 y 4.4).

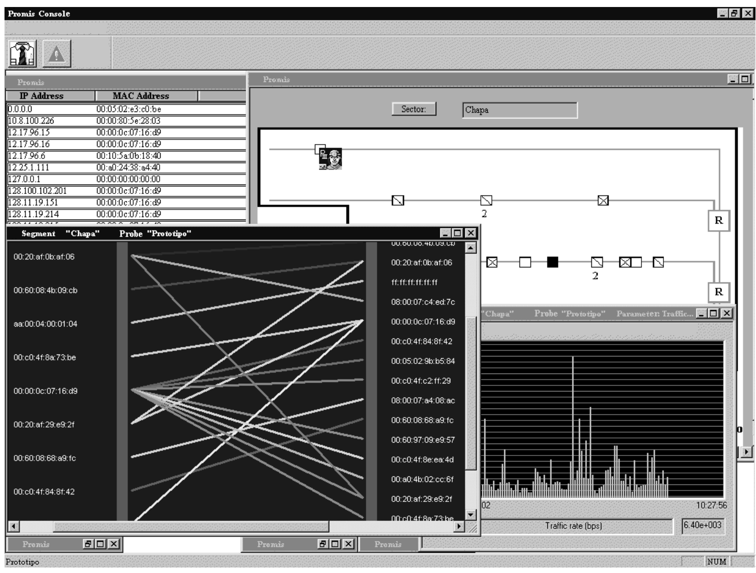


Figura 4.3: Consola PROMIS sobre Windows NT

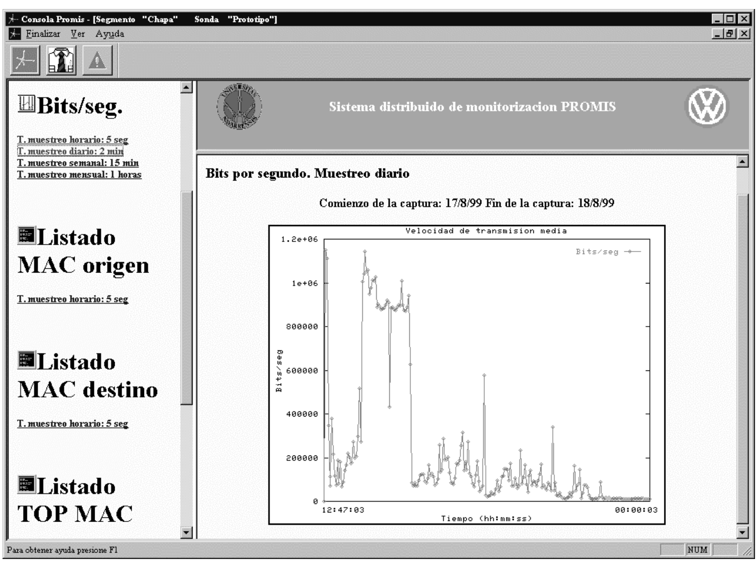


Figura 4.4: Consola PROMIS sobre WWW

La consola sobre Windows NT ofrece un interfaz visual en el que se puede representar toda la información de monitorización en forma de gráficas, listados o cuadros de configuración. De esta forma se hace que la configuración remota de sondas y la representación de resultados sea muy intuitiva y visual. También se puede representar la información que se recibe en tiempo real desde una sonda. La comunicación con las sondas se realiza mediante un modelo cliente - servidor sobre TCP/IP. La consola realiza peticiones de información a la sonda y según el tipo de las mismas recibirá la información en una única transferencia (captura o informe periódico), o irá recibiendo el parámetro actualizado conforme se calcule (monitorización en tiempo real o alarmas).

4.2.4 Implantación

Durante el año 1999 se hicieron las primeras pruebas del sistema de monitorización en redes de VW Navarra, en las que se ajustaron las características del sistema a los requerimientos de los gestores de red de VW Navarra y se testearon las funcionalidades del sistema en un periodo de funcionamiento prolongado.

En abril de 2000 se han implantado 10 sondas en otras tantas redes de producción de Volkswagen-Navarra, que son redes 10Base-5 (coaxial grueso). De esta forma, desde la sala de operadores se tiene una visión completa de la red en todo momento.

Debido a la confidencialidad de los datos del proyecto y a las dificultades encontradas en Volkswagen Navarra S.A. para autorizar que figurasen en este trabajo los resultados obtenidos con la implantación de la herramienta PROMIS, no nos es posible presentar aquí estos datos. Sin embargo, todo este estudio figura en un informe interno del proyecto titulado *"Informe final de implantación del sistema de monitorización PROMIS en la red de Volkswagen Navarra S.A."* [101]. En este informe aparecen los manuales del sistema y un análisis detallado de la carga y aplicaciones de producción sobre las redes durante el año 2000, información obtenida gracias al sistema PROMIS. Mediante la herramienta se ha comprobado la uniformidad del tráfico de estas redes, debido sobre todo a las características del tráfico generado por las aplicaciones de producción que es muy periódico.

Resultado de este proyecto es la obtención del *Premio a la Tecnología de la Información en Navarra* otorgado por PriceWaterHouseCoopers, PCWEEK y COMPAQ en Pamplona, el 7 de Octubre de 1999.

4.3 Sistema MONET

Se trata de una herramienta de monitorización distribuida, evolución de la herramienta anterior PROMIS, que proporciona una mayor flexibilidad gracias al uso de un arquitectura de filtrado completa basada en el algoritmo PAM-Tree. Su desarrollo e implantación (1999-2001) ha sido financiado por un proyecto FEDER/Plan Nacional de I+D, titulado *"Migración de redes HFC a redes multiservicio"* (Proyecto TIC 2FD97-0960-C05-04) y un convenio OTRI con la empresa RETENA S.A. de igual título, con investigadores principales Dr. Javier Aracil Rico y Dr. Jesús Villadangos Alonso. Por ir destinado a una empresa operadora de cable,

MONET incorpora ciertas peculiaridades para el mejor soporte de cabeceras de redes de cablemódem. De nuevo dos son los elementos principales de este sistema: la sonda y la consola.

4.3.1 Funcionalidades del sistema

Las funcionalidades que ofrece el sistema están íntimamente relacionadas con los tipos de tareas disponibles, ya que la información que proporciona la sonda a la consola es la generada por las tareas que esta última haya configurado.

La sonda ofrece diversos tipos de tareas. Por una parte están las tareas de procesado, que permiten obtener el valor de determinado parámetro, y por otra las tareas de presentación de la información, que permiten recopilar parámetros de monitorización para mostrarlos al usuario. Ambas se combinan para multiplicar las posibilidades de información de monitorización en la consola.

Las tareas de procesado se pueden clasificar en:

- Capturas de paquetes. Este tipo de tareas permite capturar paquetes completos junto con una marca de tiempo del instante en el que se realiza la captura. Esta tarea ya existía en el sistema PROMIS, pero sólo para capturas a disco duro. Ahora se pueden recibir en tiempo real en la consola, incorporarlos a informes, etc.

- Monitorizaciones de un parámetro del tráfico. Permiten contabilizar el número de bits o de paquetes, o una función genérica sobre los campos del paquete que cumple las condiciones impuestas por la red de filtrado en intervalos de tiempo configurables. La incorporación de la función genérica aporta mayor flexibilidad con respecto al sistema PROMIS.

- Descubrimiento de los valores que toma un parámetro del tráfico. Se trata de un tipo de tareas que emplea un tipo especial de parámetro que requiere que el último subfiltro de la red de filtrado sea de tipo autodescubrimiento. Éste es un tipo especial de subfiltro que permite el paso de todos los paquetes que procesa pero que genera un campo de bits como salida que únicamente es aceptado por las tareas de tipo descubrimiento. Las tareas de este tipo contabilizan el número de bits o de paquetes asociados a cada uno de los diferentes campos de bits de salida del último filtro que se haya localizado, en intervalos de tiempo configurables. Además, en este tipo de tareas es posible seleccionar el número de estas parejas, formadas por un campo de bits y el contador del número de bits o paquetes asociado al mismo que se desea que la tarea proporcione en los resultados. Esta funcionalidad no existía en el sistema PROMIS y es muy útil para definir de manera dinámica matrices de tráfico, máquinas TOP o descubrimiento de puertos destino.

Las tareas de presentación son similares a las del sistema PROMIS pero ahora existe mayor flexibilidad en la definición de parámetros sobre los que se realizan estas tareas. La clasificación de las tareas de presentación es la siguiente:

- Tareas en línea. Este tipo de tarea permite desarrollar labores de monitorización cuyos resultados son enviados a la consola según van siendo generados.

- Tareas fuera de línea. Posibilitan la configuración en las sondas de labores de monitorización que se desarrollen durante un intervalo de tiempo definido. Los resultados de estas tareas son almacenados en la sonda hasta que la consola los requiera.

- Tareas periódicas. Tienen un funcionamiento similar al de las tareas fuera de línea, pero además cuentan con la característica de que se repiten indefinidamente cada cierto periodo de tiempo (cada minuto, hora, día, semana, mes o año). Permite conservar el número deseado de informes ya pasados de cada parámetro.

- Alarmas. El parámetro se compara contra un umbral y si lo supera durante un periodo de tiempo determinado (por encima o por debajo) se envía de manera inmediata una alarma a la consola.

4.3.2 Sonda

La implementación de la sonda [102] consiste en un programa escrito en el lenguaje de programación C++ y que se ejecuta sobre un PC PentiumII 350MHz con sistema operativo Linux RedHat 6.1. Este equipo dispone de un adaptador de red que permite capturar el tráfico a monitorizar. El trabajo que debe de realizar la sonda se ha estructurado en cuatro grandes bloques encargados de la captura de paquetes, el procesado de esos paquetes, la gestión de las tareas a desarrollar y la comunicación con la consola como se muestra en la figura 4.5. Estos bloques se asimilan al subsistema de captura (bloque lector), de filtrado (bloque de procesado) y servidor (bloques servidores y el gestor) mostrados para la arquitectura general de un sistema de monitorización.

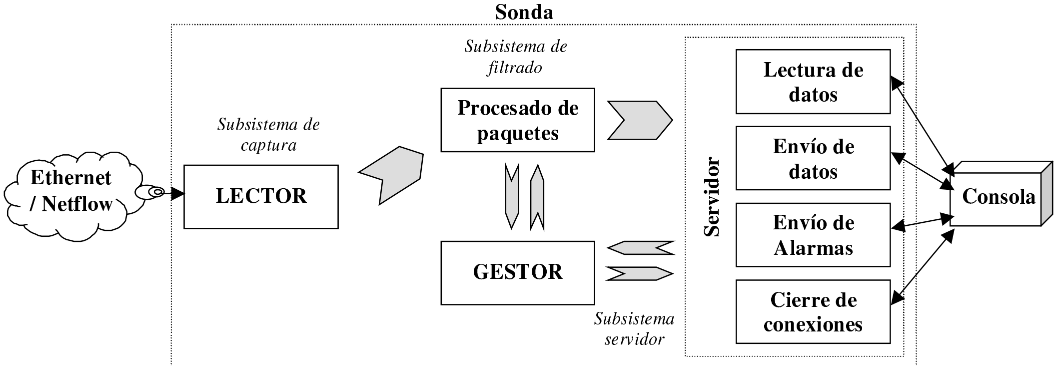


Figura 4.5: Estructura en bloques de la sonda MONET

El funcionamiento de cada uno de los bloques que componen la sonda puede resumirse del siguiente modo:

- El bloque lector es el bloque encargado de la adquisición de los paquetes.

- El bloque de procesado de paquetes es el que realiza el filtrado de los paquetes capturados que requiera cada una de las solicitudes de monitorización realizadas por la consola. Este bloque se comunica con el bloque lector para conseguir los paquetes capturados y con el bloque servidor para enviar a las consolas los resultados generados por las tareas que se estén desarrollando. Implementa el algoritmo PAM-Tree.

- El bloque gestor se encarga de añadir y eliminar del bloque de procesado de paquetes las tareas solicitadas por la consola a través del bloque servidor. Este bloque también se encarga de enviar, empleando los mecanismos que proporciona el bloque servidor, los resultados de determinados tipos de solicitudes que realizan las consolas.

- El bloque servidor es el bloque que realiza el trabajo relacionado con la comunicación con las consolas: aceptar conexiones, recibir datos, enviar datos, enviar alarmas y cerrar conexiones. Este bloque se comunica con el bloque gestor para proporcionarle las solicitudes que realicen las consolas y tomar los valores de los parámetros de monitorización a suministrar a las consolas.

La sonda requiere que el trabajo que realizan estos bloques se desarrolle concurrentemente. Para ello, el bloque servidor cuenta con cuatro hilos, cada uno de los cuales lleva a cabo el trabajo de uno de los sub-bloques del servidor. El bloque gestor cuenta con un hilo que se encarga de comprobar qué tareas deben de ser añadidas o eliminadas del bloque de procesado de paquetes, y el bloque de procesado de paquetes cuenta con otro hilo que se encarga de extraer los paquetes del bloque lector y de realizar el procesado sobre el árbol de filtrado. La comunicación entre bloques que emplean distintos hilos es controlada mediante secciones críticas que garantizan la integridad de las estructuras de datos empleadas.

La sonda soporta la captura de paquetes de redes típicas como Ethernet o cualquier otra, quedando en la consola la tarea de mapear los protocolos en los subfiltros correspondientes del árbol. Un aspecto interesante además es el soporte del protocolo NetFlow [103]. Se trata de un flujo de información UDP que los routers Cisco son capaces de enviar con información resumida del tráfico que ven por sus interfaces y con información por flujo del tipo dirección IP origen-destino, puertos, protocolo de transporte, duración del flujo, bytes del flujo, etc. Esto será muy útil para llevar control de las cabeceras de cablemódems porque con una sonda se podrá monitorizar de manera simultánea todos los interfaces de la misma cabecera de cablemódems. Sin embargo, la monitorización del tráfico basada en la información ofrecida por NetFlow reduce considerablemente las posibilidades ofrecidas por el sistema en los siguientes aspectos:

- No es posible realizar capturas de paquetes, puesto que NetFlow no proporciona los paquetes que circulan por la red, sino sólo un resumen de ellos.

- Se reducen notablemente las posibilidades de parámetros de monitorización debido a que NetFlow tan sólo proporciona una breve información del tráfico que circula por la red, en concreto información resumida por flujo.

- En caso de que el número de flujos activos sobrepase el número máximo de flujos que puede aceptar la caché de la cabecera de cablemódem, parte del tráfico cursado no podrá ser monitorizado.

- Se reduce notablemente la resolución temporal de las monitorizaciones. Esto es debido a que NetFlow sólo proporciona información del tráfico cuando caducan los flujos o venza cierto timeout (mínimo de un minuto). Por lo tanto, puesto que no se puede garantizar que la máxima duración de un flujo sea inferior a un minuto, no puede emplearse una resolución temporal inferior a varios minutos sin que el error debido al efecto de borde sea elevado. Este efecto borde se produce por ejemplo cuando un flujo comienza en un intervalo y finaliza en el siguiente: se contabilizará únicamente en el segundo intervalo y por tanto cuanto más grande sea el intervalo mayor será el error.

El soporte NetFlow ha sido útil para la monitorización de las cabeceras de cablemódems Cisco de la empresa Retena, como complemento a la sonda genérica con captura de paquetes de la red.

4.3.3 Consola

La consola [104] está implementada en JAVA JDK 1.2.2 con acceso JDBC a una base de datos de protocolos en la que se definen las cabeceras, campos de los protocolos y sus posibles valores, lo cual facilitará la definición de filtros a través de un interfaz muy intuitivo en la consola. Partiendo de un tipo de red Ethernet o monitorización NetFlow, al usuario se le presentan las opciones de filtrado dejando flexibilidad para testear el valor de ciertos bits de un campo de la cabecera de protocolo actual, pasar a la siguiente cabecera de protocolo o definir el tipo de procesado en que se está interesado.

Provee un interfaz unificado para acceder a la información de monitorización de las distintas sondas, permitiendo visualizar y exportar los datos. En la figura 4.6 se presenta una foto de la consola en funcionamiento. En ella se observa una gráfica temporal de carga de la red en línea, una matriz de tráfico y una gráfica de barras de porcentaje por servicios.

4.3.4 Implantación

Las primeras pruebas se han realizado en la propia cabecera de cablemódems Cisco uBR7246 [105] situada en la central de Retena y han consistido en instalar dos sondas sobre la misma cabecera: una en el enlace Fast Ethernet de la cabecera a la troncal y otra recibiendo la información NetFlow a través de un puerto Fast Ethernet dedicado en la propia cabecera, tal y como se muestra en la figura 4.7. Está prevista la implantación definitiva del sistema a lo largo del segundo semestre de 2001, con sondas NetFlow en las cabeceras y Fast Ethernet en las granjas de servidores, clientes especiales y acceso a Internet.

En este apartado se van a presentar una serie de datos obtenidos de la prueba piloto en la red de Retena. Los datos presentados corresponden a monitorizaciones configuradas en las sondas desde el 21 de Noviembre del 2000 al 1 de Enero de 2001 [106, 107, 108, 109] sobre ambas sondas de la cabecera de cablemódems mencionada.

Primeramente se puede comparar el funcionamiento de la sonda Ethernet y NetFlow. En la figura 4.8 se presenta la carga en bits por segundo medida en la cabecera durante la semana 6-12 de diciembre de 2000, procedente de los informes semanales configurados en ambas sondas. En

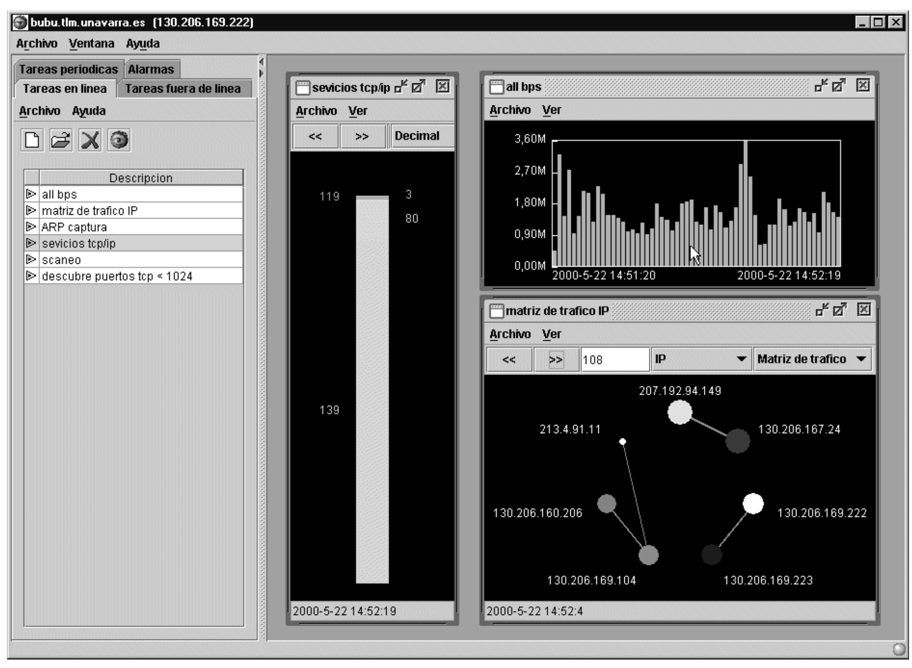


Figura 4.6: Interfaz de consola MONET

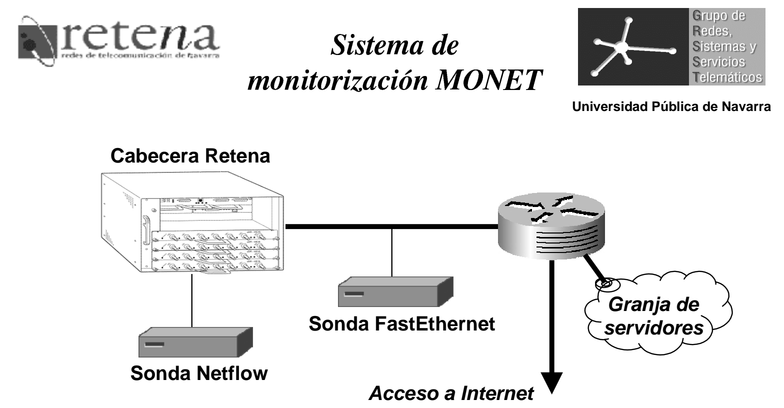


Figura 4.7: Implantación de sondas MONET en la cabecera de la central de RETENA

primer lugar se observa la mayor resolución de la sonda Ethernet, 15s que podría bajarse hasta 1s pero que no se hace para no tener excesivo número de muestras, frente a la sonda NetFlow que es de 1 hora para tener resultados fiables debido al efecto frontera en la medida de flujos caducados y para reducir el tráfico extra y la carga de CPU del router. Con la sonda Ethernet podemos llegar a resoluciones de 1 segundo sin problemas mientras que con NetFlow el límite está en 1 minuto debido al timeout mínimo de caducidad de los flujos que se puede configurar en la cabecera Cisco de cablemódems. Además, esta resolución supone una sobrecarga de consumo de CPU apreciable en la cabecera, muchas veces no asumible. Realizando medidas con idéntica resolución temporal de más de 10 minutos se obtienen resultados parejos en ambas sondas, sin embargo, para resoluciones inferiores la sonda Ethernet es la única alternativa valida como se verá posteriormente.

En las figuras se observa que la red está muy poco cargada, con una tasa media en la semana de 389,28Kbps (frente a los 30Mbps que es capaz de soportar la cabecera en bajada y unos pocos Mbps en subida). Al ser menor la resolución de la sonda NetFlow suaviza la medida de tráfico. El perfil diario se mantiene toda la semana observándose bajas tasas a la madrugada y primeras horas de la mañana, y posteriormente a primera hora de la tarde.

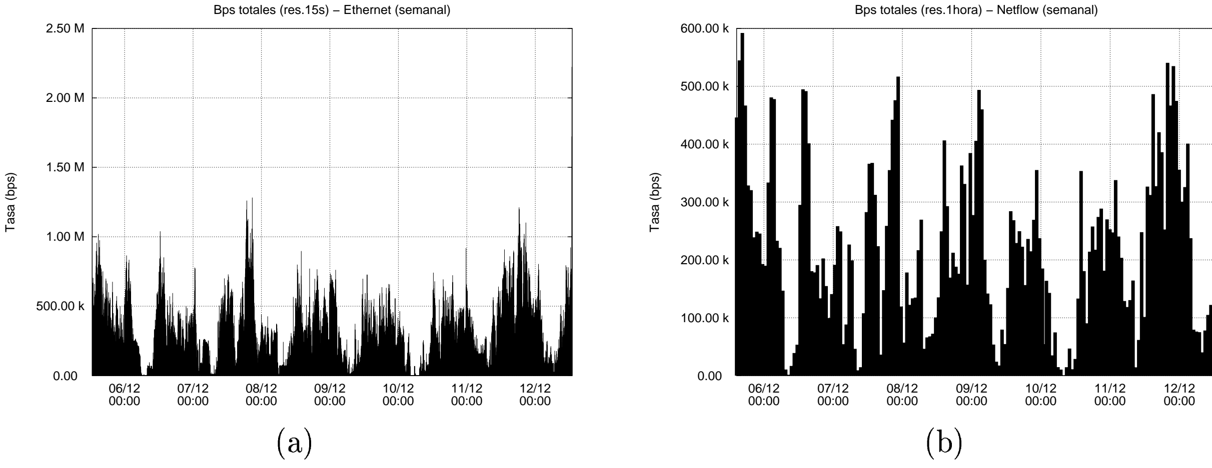
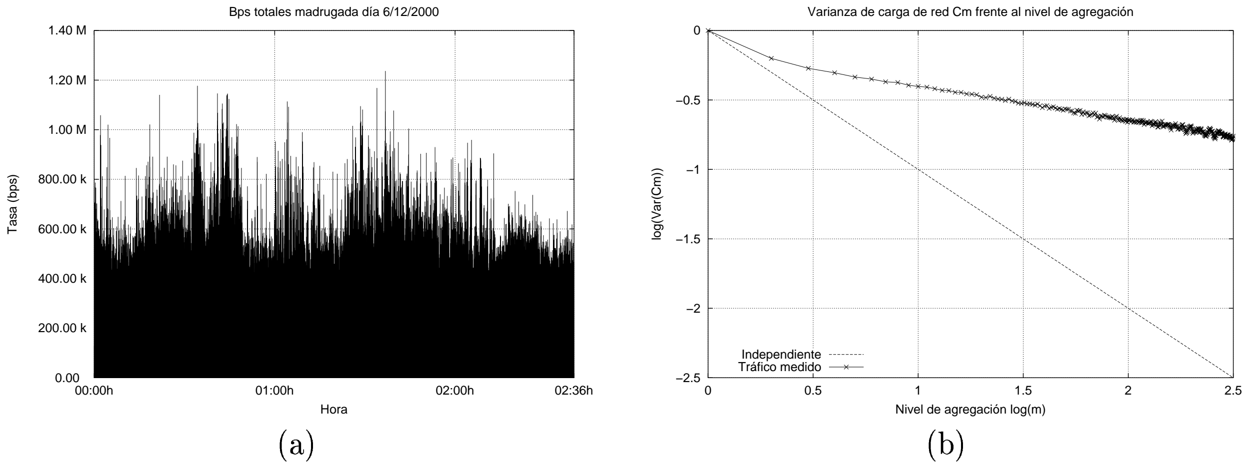


Figura 4.8: Carga de la red semana 6-12/12/2000 a) Sonda Ethernet b) Sonda NetFlow

Si nos centramos en un periodo de tiempo en el que la carga de la red se pueda considerar estacionaria, se va a comprobar si el tráfico que circula por la red es de incrementos independientes a cualquier escala de tiempo por medio de un test de variabilidad.

El test que aquí se presenta consiste en representar el logaritmo del nivel de agregación (m) frente al logaritmo de la varianza del proceso de carga de la red C_m [110]. El nivel de agregación es el número de intervalos en que se divide la zona en estudio, y si el proceso es independiente, por el Teorema Central del Límite, la varianza del proceso debe caer con la inversa del nivel de agregación, es decir, una recta de pendiente -1 si se toman logaritmos. Esto es lo que ocurriría en un proceso que siguiese una distribución de Poisson.

Tomamos el periodo de tiempo 00:00h a 02:36h del día 6 de diciembre de 2000 en el que la carga de la red se pueda considerar estacionaria (figura 4.9a) y se va a comprobar que el tráfico no es de incrementos independientes. En efecto, en la figura 4.9b se representa con aspas el logaritmo de la varianza del proceso de carga de la red frente al logaritmo del nivel



(a) (b)

Figura 4.9: a) Carga de la red del día 6/12/2000 00:00-2:36h b) Test de varianza frente a nivel de agregación

de agregación, procesado con un intervalo mínimo de 1 segundo (para $log(m) = 0$) y con un tamaño de traza de 2h36min. Si fuera un proceso de generación de incrementos independientes caería según la recta intermitente dibujada, pero se observa como cae más despacio. Esto demuestra que no existe independencia en la tasa de bits por segundo entre intervalos. En consecuencia, no se puede suponer que la carga por intervalo sea independiente e idénticamente distribuida como hacemos por ejemplo al suponer una distribución de Poisson. Los modelos de tráfico adecuados para este escenario son los procesos autosimilares (*self-similar*) [111].

En el transcurso de esa semana se han detectado una serie de servicios sobre puertos más o menos conocidos. En las tablas 4.1 y 4.2 se presentan los puertos origen TCP y UDP que más tráfico han generado en el conjunto de la semana detectados por la sonda Ethernet. El 85% del tráfico total TCP está repartido entre los 8 primeros servicios, predominando como era de esperar el de Web, que se lleva casi un 47%. En la figura 4.10 se muestra el tráfico web en esa misma semana y se observa que el tráfico web sigue un perfil proporcional al tráfico total, con un tráfico destino (peticiones) poco significativo y con un tráfico de bajada (bajada de páginas del servidor) que supone la parte importante. Aparece también un servicio novedoso como Napster en las primeras posiciones.

En cuanto a los flujos UDP, destacan principalmente los dedicados a servicios de juegos en red como Quake III o Half-Life. Resulta que estos servicios reparten el tráfico de manera bastante simétrica entre los flujos cliente a servidor y en sentido contrario. Si se sumara el tráfico generado por los juegos en red Quake III y Half-Life en todos esos puertos, el porcentaje de tráfico de estos servicios supondría un 67-68% respecto al tráfico total UDP de la red.

Un ejemplo de flexibilidad de la herramienta es la posibilidad de contabilizar los usuarios conectados en cada momento, que consiste en observar paquetes con direcciones IP origen o destino que pertenezcan a las de usuarios de Retena (10.1.8.1/255.255.248.0 en el mapeo de direcciones privadas que desde Enero 2001 ha migrado a direcciones públicas de Internet). En la figura 4.11 se muestra este número de usuarios conectados con intervalos de 15 s, comprobándose que este número no sobrepasa nunca los 20, lo que justifica que la cabecera bajo análisis esté tan descargada. Se trata de una zona de poca cobertura con escasos clientes en

Bits/semana	*% Total*	*Puerto*	*Servicio*
6,0542E+10	46,81%	80	WWW
2,1002E+10	16,24%	20	FTP-data
1,6679E+10	12,90%	6699	Napster
5309031792	4,11%	6688	Napster/Gnutella
4321271928	3,34%	8090	***
1744557576	1,35%	110	POP-3
921257376	0,71%	119	NNTP

Tabla 4.1: Puertos TCP origen

Bits/semana	*% Total*	*Puerto*	*Servicio*
486821464	21,85%	27960	Quake III
398299024	17,87%	27005	Half-Life cliente
355970928	15,97%	27015	Half-Life servidor
261771336	11,75%	53	DNS
220563464	9,90%	27017	Half-Life servidor
205648424	9,23%	2048	DLS-monitor
68911488	3,09%	67	Bootps

Tabla 4.2: Puertos UDP origen

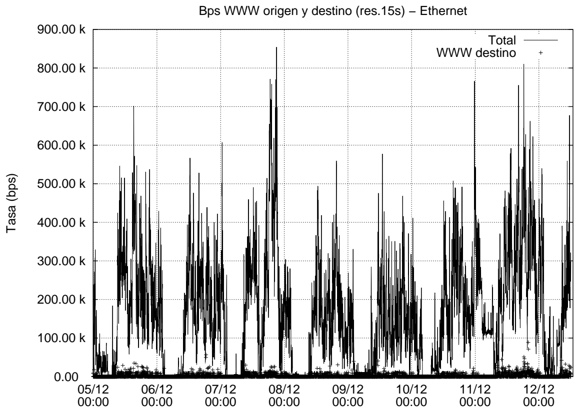


Figura 4.10: Bps WWW

el momento de las pruebas.

En la figura 4.12 se presenta el detalle de tráfico de una dirección IP interna 10.1.9.53 que podrá estar asignada a usuarios diferentes en cada intervalo de tiempo debido a la asignación dinámica de direcciones IP (DHCP). Corresponde a un usuario tipo AVE256 que tiene contratado 256kbps de ancho de banda y se puede comprobar en la figura que nunca se supera dicha tasa. Las conexiones de usuarios con esta dirección IP varía de los 2min 30s con 3.27 Kbps de tasa media a los 11h 48min con 12.02 Kbps de tasa media.

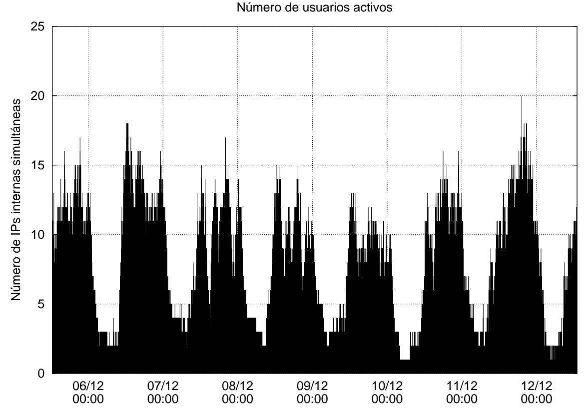


Figura 4.11: Número de usuarios simultáneos

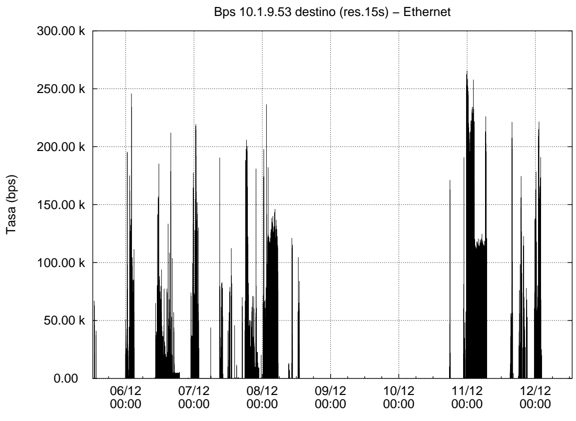


Figura 4.12: Bps IP destino 10.1.9.53

Obtener información de las máquinas TOP origen mediante MONET es inmediato: se define un filtrado hasta nivel IP y finalmente autodescubrimiento del campo IP origen y procesado de bps. En la tabla 4.3 se presentan las máquinas de más tráfico para esta semana. Las dos primeras direcciones IP corresponden a máquinas servidoras de una empresa privada que utiliza la infraestructura de Retena para su salida a Internet.

En lo que respecta a protocolos en las figuras 4.13 y 4.14 se presentan los perfiles de tráfico para dos servicios no muy usuales, distinguiendo el tráfico como destino y el total. Por un lado, en la figura 4.13 el protocolo *dls-monitor* procede de una aplicación de gestión que periódicamente pide datos de estado a la cabecera por lo que muestra un perfil muy constante aunque también escaso. Por otro lado, en la figura 4.14 se presenta el protocolo *bootps* utilizado en el arranque de los cablemódems para pasarles la configuración inicial y por tanto su perfil es proporcional al número de usuarios conectados o al de carga de la red. En esta misma semana, la distribución de protocolos por encima de IP se presenta en la tabla 4.4. El protocolo de transporte que más tráfico genera es el de TCP, que soporta el tráfico de servicios como Web, Napster, FTP-data y otros más, que predominan sobre el tráfico generado con el protocolo UDP, primordialmente referente a servicios como los juegos en red, Half-Life y Quake III entre otros.

Finalmente, en la figura 4.15 se presenta la carga total de la red durante 4 semanas, en la que hay que destacar la periodicidad del tráfico a lo largo de los días e incluso de las semanas, siendo en todos los casos tasas muy bajas. Los periodos de mayor carga coinciden con los de mayor número de usuarios simultáneos y se observan los bajones bruscos durante el mediodía y la noche.

Bits/semana	*% Total*	*Máquina origen*	*Nombre simbólico*
13363670984	9,47%	212.21.227.3	www.navalur.com
11138632144	7,89%	212.21.226.2	www.navalur.com
8181100240	5,80%	192.168.201.2	(interna)
2533986776	1,80%	209.1.224.10	res9.geocities.yahoo.com
2152814360	1,53%	209.207.205.9	
1970507064	1,40%	207.188.7.36	
1654471576	1,17%	10.1.8.36	(interna)
1426651648	1,01%	195.235.97.200	smtp.ole.com
1164659576	0,83%	207.188.7.40	chillout.real.com
1133286288	0,80%	205.141.206.69	
1132180880	0,80%	208.145.129.10	epgctac.com
1010705048	0,72%	212.21.224.4	
1010096352	0,72%	10.1.8.54	(interna)
957020800	0,68%	10.1.9.83	(interna)
799173928	0,57%	206.204.217.116	vamscomdl2.www.conxion.com
751447464	0,53%	64.41.209.5	
743590224	0,53%	10.1.9.57	(interna)
723442680	0,51%	206.132.163.167	colo00-167.xoom.com
716854384	0,51%	195.53.249.52	belenos.elmundo.es
688970304	0,49%	213.4.91.11	telenews.teleline.es

Tabla 4.3: TOP direcciones IP

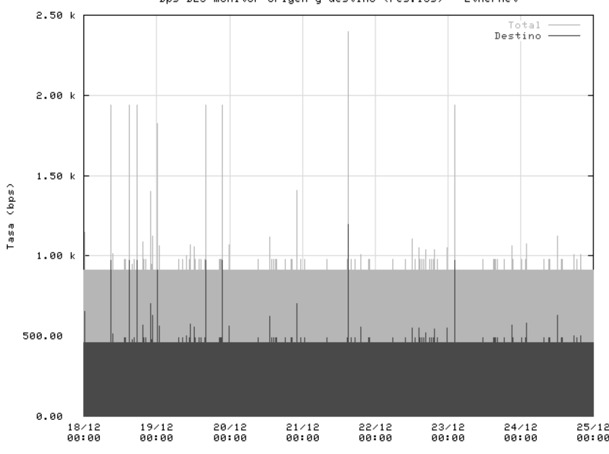


Figura 4.13: Bps aplicación dls-monitor

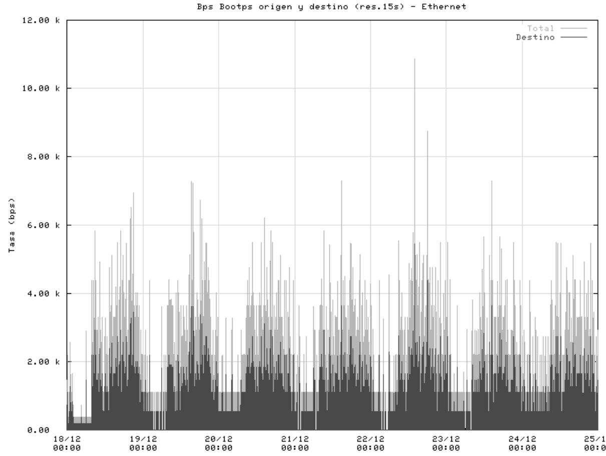


Figura 4.14: Bps aplicación bootps

Bits/semana	*% Total*	*ID protocolo*	*Protocolo*
168401347888	93,46080%	6	TCP (Transmission Control)
10242125232	5,68426%	47	GRE (General Routing Encapsulation) [RFC2784]
1289365968	0,71558%	17	UDP (User Datagram)
179550512	0,09965%	88	EIGRP Enhanced IGRP (Cisco's Interior Gateway Routing Protocol) [CISCO,GXS]
70045320	0,03887%	1	ICMP (Internet Control Message)
1600640	0,00089%	54	NARP (NBMA Address Resolution Protocol) [RFC1735]

Tabla 4.4: TOP protocolos IP

Las muestras de la figura 4.15 corresponden a intervalos de un segundo con lo que se puede apreciar toda la variabilidad del tráfico. En concreto, existen unos picos que llegan hasta 4 Mbps por estar cortados para graficar y que corresponden en verdad a picos de 20 Mbps que se realizaron a lo largo de la prueba piloto para testear la capacidad de la sonda ante ráfagas de tráfico con tamaño de paquete mínimo 64 bytes en el sentido descendente, cabecera a usuario. En todos las pruebas, no hubo pérdida de paquetes ni ralentización notable en la obtención de parámetros on-line.

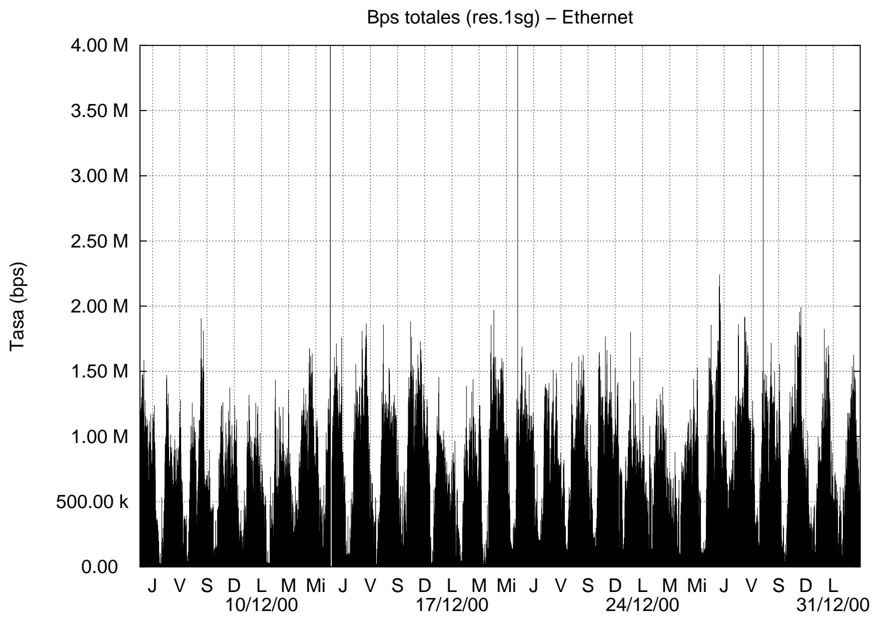


Figura 4.15: Tráfico durante 4 semanas

El estudio de la hora más cargada se muestra en la tabla 4.5. Se observa que la hora cargado varía según el día, aunque nunca aparece para muy primeras horas de la mañana, comprobándose la no estacionariedad del tráfico durante ese periodo.

Con todo esto, en el apartado se ha tratado de ofrecer una serie de resultados prácticos obtenidos con la herramienta MONET en condiciones de tráfico real, dando una pequeña idea de la utilidad de un sistema de monitorización de este tipo.

Día	*Hora más cargada*	*Bps medio*
5 Martes	14	632990
6 Miércoles	12	587806
7 Jueves	18	652982
8 Viernes	19	418340
9 Sábado	00	543221
10 Domingo	12	373947
11 Lunes	18	668406
12 Martes	17	665226
13 Miércoles	20	894841
14 Jueves	11	679473
15 Viernes	20	764956
16 Sábado	01	520586
17 Domingo	16	379350
18 Lunes	10	827060

Día	*Hora más cargada*	*Bps medio*
19 Martes	17	506658
20 Miércoles	11	426056
21 Jueves	15	581740
22 Viernes	10	580323
23 Sábado	02	628299
24 Domingo	12	503587
25 Lunes	21	437744
26 Martes	19	935660
27 Miércoles	20	789380
28 Jueves	17	612830
29 Viernes	19	994034
30 Sábado	19	709899
31 Domingo	16	681711
1 Lunes	00	256910

Tabla 4.5: Horas cargadas por día

Capítulo 5

Conclusiones y trabajos futuros

5.1 Conclusiones

En este trabajo se ha propuesto, analizado e implementado un algoritmo de filtrado de paquetes para sistemas de monitorización de redes de datos. Las contribuciones más destacables son las siguientes:

- Se han analizado los requerimientos de filtrado en un sistema de monitorización, y se ha llegado a determinar los condicionantes fundamentales de este tipo de filtrado de paquetes: elevado número de filtros, necesidad de reutilizar bloques de subfiltrado, necesidad de incorporar los parámetros de monitorización a la propia estructura de filtrado y capacidad de filtrado para las redes de alta velocidad. Las alternativas de filtrado de paquetes existentes en la actualidad no verifican los condicionantes anteriores y por tanto su eficiencia a la hora de aplicarlas a sistemas de monitorización es limitada.

- Se ha propuesto PAM-Tree (Protocol-Adaptive Monitoring Tree) como técnica de filtrado que permite optimizar el funcionamiento de un sistema de monitorización. PAM-Tree se basa en una estructura en árbol en la que cada nodo corresponde a un subfiltro. El paquete a filtrar recorrerá una serie de subfiltros siempre que verifique los subfiltros padre. Los caminos recorridos por el paquete indicarán los filtros verificados y por tanto se podrá actualizar el parámetro correspondiente. PAM-Tree soporta concatenación AND/OR de subfiltros, reutilización de bloques de subfiltrado, sigue un esquema de filtrado jerárquico e incorpora los parámetros de monitorización en el propio árbol de filtrado.

- Hemos presentado la formalización del algoritmo mediante teoría de autómatas. Esto nos permite plantear el problema, mostrar el funcionamiento de PAM-Tree y probar formalmente las propiedades del sistema de filtrado: reutilización, jerarquía e incorporación de los parámetros de monitorización en la propia estructura de filtrado.

- Se ha realizado una análisis teórico y experimental de PAM-Tree comparándolo con otras técnicas de filtrado de paquetes, demostrándose la mejora en el caso de PAM-Tree

conforme crece el número de filtros. Por tanto, su aplicación a sistemas de monitorización es adecuada.

- El algoritmo presentado se ha implementado en herramientas de monitorización utilizadas en la actualidad en redes industriales y de una operadora de cable. Ambos sistemas se han basado en plataformas PC de bajo coste demostrando su elevada capacidad.

- El algoritmo PAM-Tree se encuentra reflejado en una patente con utilización industrial.

Los resultados obtenidos demuestran la necesidad de algoritmos como PAM-Tree para realizar el filtrado de paquetes de manera eficiente en sistemas de monitorización. Además, la flexibilidad obtenida con este algoritmo de filtrado permite ofrecer innumerables funciones de monitorización, al poder realizar cualquier combinación de subfiltros asociados mediante operaciones lógicas AND/OR para definir el parámetro de monitorización deseado. De esta forma, la mayor parte del trabajo de monitorización recae sobre el algoritmo de filtrado, mientras que al resto de subsistemas de la sonda le quedan tareas de interfaz y gestión de la comunicación con la consola.

5.2 Líneas de trabajo futuras

A partir de aquí el trabajo se orienta en diferentes aspectos que describiremos a continuación.

En primer lugar, el desarrollo del algoritmo de filtrado PAM-Tree a nivel de kernel de un sistema operativo Linux o BSD mejorará aún mas los resultados obtenidos. Con ello se evita el coste de la copia de paquetes del kernel al nivel de usuario, que hasta ahora se debía producir para todos los paquetes. De esta forma, se estará en igualdad de condiciones con los sistemas packet filter.

La filosofía del sistema de monitorización que se ha considerado en el trabajo ha sido orientada al filtrado de paquetes, que es la más habitual. Sin embargo, se pueden ampliar las funcionalidades del sistema de monitorización si adaptamos el algoritmo al tratamiento de conexiones, útil por ejemplo para obtener datos como la duración de las conexiones, bytes intercambiados por conexión, asimetría de la conexión, efecto de los mecanismos de TCP sobre estas conexiones, número de conexiones establecidas por usuario o por servicio, etc. Esto permitirá un mejor control de parámetros de red que tendrán impacto en la calidad de servicio ofrecida a los usuarios.

Otro aspecto interesante es el estudio del comportamiento del PAM-Tree en redes de alta velocidad, con utilización intensiva de filtrados y número elevado de paquetes por segundo. Aquí entran en juego factores limitantes del sistema operativo o de la plataforma PC, que con cada paquete recibido producen una interrupción en el sistema y por tanto la atención de estas interrupciones supone de por sí una carga elevada. Entonces se hace necesario acudir a soluciones hardware.

Se va a realizar la implementación del algoritmo sobre un sistema embebido, en concreto del Motorola MPC8260 [112] dotado de un procesador PowerPC y de un Módulo Procesador de Comunicaciones (CPM) que facilita la utilización de diferentes interfaces de red: 155 Mbps ATM (AAL0, AAL1, AAL2, AAL5), 10/100 Mbps Ethernet, 45 Mbps HDLC, UTOPIA o serie. Este desarrollo se enmarca dentro del proyecto ESTIGMA para dotar de funcionalidades de monitorización al terminador de red TR-BA de la empresa INTELNET utilizado por Telefónica para terminar su red en los puntos de conexión de sus clientes a la red de banda ancha.

Con todo ello, se puede completar el algoritmo propuesto para convertirlo en una solución global, sobre plataformas de propósito general o sobre diseños específicos.

Apéndice A

Algoritmos de búsqueda/clasificación generales

A.1 Introducción

Para acceder a filtros y datos de monitorización es necesario utilizar estructuras de datos que permitan su acceso rápido y con los menores recursos de memoria posibles. En el siguiente apéndice exponemos los algoritmos clásicos de búsqueda, que permitirán el diseño eficiente del sistema de filtrado para monitorización.

El problema de búsqueda general consiste en cómo encontrar el dato que ha sido almacenado con determinado identificador. Denominaremos a este identificador *clave* de la búsqueda y a los datos entre los que se realiza la búsqueda *registros*. La colección de registros se denominará *tabla*. En general veremos que los algoritmos presentados requieren que las claves sean diferentes para cada dato. El resultado de la búsqueda podrá ser éxito si se encuentra el registro con cierta clave, o fracaso si no se encuentra.

Los algoritmos de búsqueda se pueden clasificar de acuerdo a varios criterios [62, 113, 114]:

- Internos, cuando los registros se encuentran en memoria, y externos, cuando no todos los registros se encuentran en memoria. Los primeros permiten una mayor flexibilidad.

- Estáticos, cuando los registros no cambian y, por tanto, es importante minimizar el tiempo de búsqueda más que el de configuración de la tabla, y dinámicos, cuando la tabla sufre continuas inserciones y borrados.

- Según se basen en comparaciones entre claves o en propiedades binarias de las claves.

- Según trabajen con las claves directamente o con transformaciones de las claves.

A continuación se presentan los algoritmos clásicos de búsqueda secuencial, por comparación de claves, búsqueda digital y por hashing.

A.2 Búsqueda secuencial

Los algoritmos de búsqueda secuencial son los considerados “de fuerza bruta”. Una primera versión consiste en empezar la búsqueda por el comienzo de la tabla, y avanzar registro a registro comparando la clave hasta encontrar la coincidencia. Si se llega al final de la tabla no se ha obtenido éxito en la búsqueda.

Una mejora, la búsqueda secuencial rápida, añade al final de la tabla un registro comodín indicador de final de tabla con la misma clave que la buscada. De esta forma se reducen de dos a una las comparaciones en cada iteración.

Otra mejora de la búsqueda secuencial rápida consiste en dividir por dos el número de iteraciones para completar el recorrido de la tabla, manteniendo el registro comodín final con la clave. Para ello en cada iteración se compara la igualdad del registro actual y la desigualdad del registro siguiente.

Si la tabla contiene los registros ordenados, el algoritmo búsqueda secuencial rápida se puede aprovechar de este hecho: mientras la clave buscada sea mayor que la actual se avanza y se detiene cuando es igual (éxito) o menor (no éxito).

Para optimizar la búsqueda es interesante colocar los registros que se buscan con mayor probabilidad al principio de la tabla. Si no se dispone de las probabilidades de búsqueda de cada elemento a priori será necesario calcularlas conforme se realizan búsquedas y por tanto la tabla tendrá que ser auto-organizativa. En lugar de realizar este cálculo en línea se pueden aplicar otras técnicas como poner al principio el registro encontrado en la última búsqueda o intercambiar el registro encontrado con el que le antecede si no es el primero. Ambas técnicas ofrecen buenos resultados cuando búsquedas sucesivas no son independientes (pequeñas series de claves suelen aparecer en grupos).

A.3 Búsqueda por comparación de claves

En este apartado se discuten los métodos de búsqueda basados en el orden lineal de las claves (alfabético o numérico, y operadores $<, =, >$).

A.3.1 Búsqueda binaria

Consiste en comparar la clave con el registro intermedio de la tabla (entero superior a $\frac{N}{2}$, donde $N+1$ es el número de entradas que almacena la estructura) y el resultado determina qué mitad de la tabla se debe tomar para repetir el proceso. Como mucho tras $\log_2 N$ comparaciones encontraremos la clave o concluiremos que no se encuentra. A este método también se le denomina búsqueda logarítmica o bisección.

Una forma inmediata de representar este método de búsqueda es mediante un árbol binario como el mostrado en la figura A.1, pero la estructura de datos real del algoritmo es una tabla estática con la secuencia de números.

En la figura A.1 se tiene $N+1 = 9$ luego la primera comparación se producirá para K_4 (K_i

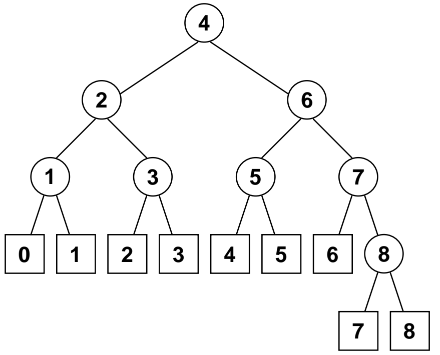


Figura A.1: Ejemplo de árbol binario

es el número de clave i-ésimo), representado por el nodo circular 4. Entonces si $K < K_4$ se continúa con el subárbol izquierdo y si $K > K_4$ se continúa con el derecho. Si no se encuentra la clave, se llegará a uno de los nodos externos representados con cuadrados en la figura y numerados de 0 a N. Por ejemplo, se alcanzará el 6 si y sólo si $K_6 < K < K_7$.

Con este método se realizan en media $\log_2 N - 1$ comparaciones en búsquedas exitosas y ningún otro método basado en comparación obtiene mejores resultados. Existen diversas variantes de este algoritmo que se presentan a continuación.

A.3.1.1 Búsqueda binaria uniforme

Una variante es la búsqueda binaria uniforme que aprovecha el hecho de que no es necesario llevar cuenta de las longitudes de los intervalos para calcular el elemento central sino que es suficiente con llevar cuenta de un factor Δi para cada nivel que nos dirá cuanto aumentar o disminuir el índice para apuntar al elemento central del siguiente intervalo. Esto puede hacer que haya comparaciones redundantes justo antes de terminar pero es más eficiente para el caso de no éxito en la búsqueda. Para optimizar, incluso se puede tener calculados estos factores en una tabla a priori. En la figura A.2 se muestra un ejemplo de búsqueda binaria uniforme.

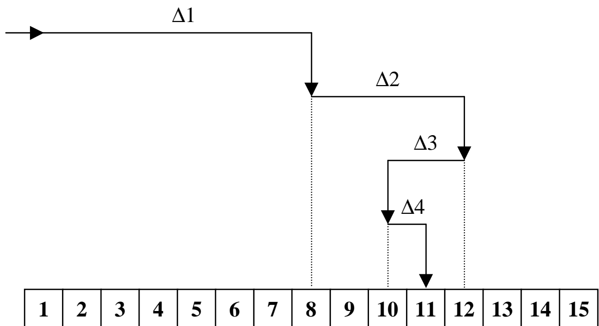


Figura A.2: Ejemplo de búsqueda binaria uniforme

A.3.1.2 Búsqueda de Fibonacci

Los números de Fibonacci pueden realizar una labor análoga al de las potencias de 2 en las búsquedas binarias. El primer número de Fibonacci es el 0, el siguiente es el 1, y el resto se calculan con la suma de los dos anteriores.

Para explicarlo hacemos uso del árbol de Fibonacci de orden k, F_k, que tiene $F_{k+1}-1$ nodos internos (circulares) y F_{k+1} externos (cuadrados) que se construye como sigue:

- Si $k=0$ o $k=1$ sólo existe el nodo 0.
- Si $k \geq 2$ la raíz es F_k, el subárbol izquierdo es un árbol de Fibonacci de orden $k-1$ y el derecho de orden $k-2$ con los números incrementados F_k.

En la figura A.3 se presenta un árbol de Fibonacci de orden 5.

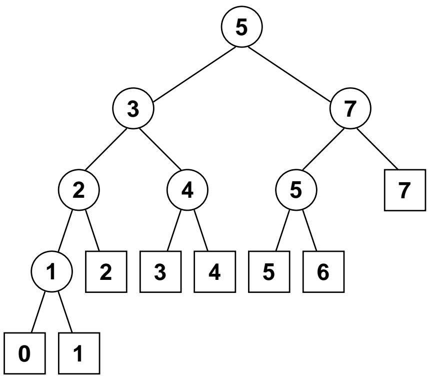


Figura A.3: Árbol de Fibonacci de orden k=5

El resultado es una búsqueda ligeramente más rápida que la Búsqueda Binaria Uniforme en media, aunque en el caso peor sea un poco más lento, dependiendo de la distribución que sigan los números a localizar.

A.3.1.3 Búsqueda por interpolación

Para un K entre K_q y K_u se puede elegir el siguiente registro a comparar en la posición $(K-K_q)/(K_u-K_q)$ del camino entre q y u (supone claves numéricas y crecientes en el intervalo). Este cálculo extra no se compensa con la disminución del número de comparaciones, a no ser que interese acceder más rápido a unos rangos frente a otros, por ejemplo por ser más probables los números en ese rango.

A.3.2 Búsqueda por árbol binario

Para un valor dado de N entradas, el árbol correspondiente a una búsqueda binaria tiene el mínimo número teórico de comparaciones necesarias para realizar una búsqueda en una tabla

a base de comparaciones. Sin embargo, los métodos anteriores son apropiados para tablas estáticas, ya que si fueran dinámicas se perdería más tiempo en las inserciones y borrados que el que se ganaría usando la búsqueda binaria.

Una estructura en árbol binario explícita permite borrados e inserciones rápidas, y es útil como método de búsqueda y ordenación. El árbol creado tiene un subárbol derecho cuyas claves son mayores (numérica o alfabéticamente) que las del subárbol izquierdo. Leyendo los nodos internos de izquierda a derecha las claves se obtienen ordenadas.

Un problema que existe en estos métodos es el del balanceado. Si los nodos se insertan en el árbol en orden, éste sólo crecería por una rama convirtiéndose la búsqueda en secuencial. Esto es lo que se denomina árbol degenerado en el que el coste de la búsqueda es proporcional a N. El caso contrario se denomina árbol bien balanceado y el coste de la búsqueda es proporcional a $\log N$. Si además la probabilidad de buscar cada nodo es la misma, el coste medio de esa búsqueda es proporcional a $\sqrt{N}$.

El borrado de nodos es inmediato, teniendo sólo cuidado de que el árbol resultante siga siendo binario.

Existe un algoritmo que encuentra el árbol binario óptimo dada una serie de registros y la probabilidad de búsqueda de cada uno, pero no es de nuestro interés porque es necesario tener estos datos a priori cosa que no ocurre en nuestra aplicación. Además es necesario conocer de antemano las probabilidades de búsqueda de cada clave. Sin embargo, en caso de claves igualmente probables se demuestra que el árbol binario balanceado es el mejor.

A.3.3 Árboles balanceados

La altura de un árbol se define como su máximo nivel, la longitud del camino más largo desde la raíz a un nodo externo. Un árbol binario se dice balanceado si la altura del subárbol izquierdo de todo nodo nunca difiere más de ± 1 de la altura de su subárbol derecho.

Los algoritmos de inserción y borrado se basan en rebalancear el árbol cuando la inserción o eliminación de un registro va a producir que pase a dejar de ser balanceado. El coste es de $C \log N$, con C dependiente de las probabilidades de búsqueda/inserción de elementos a distintas alturas del árbol.

Existen otras alternativas como el *árbol de pesos balanceado*, que permite una diferencia de alturas entre ramas de hasta cuatro unidades, y el *árbol 2-3*, en el que puede haber hasta 3 nodos descendientes de uno superior. En ambos casos las reorganizaciones para mantener el balanceado son menos costosas por producirse cada más tiempo.

A.3.4 Árboles multicamino

Se utilizan básicamente para almacenamiento en dispositivos secundarios (discos duros, cintas, etc.). Consisten básicamente en árboles cuyos nodos pueden tener más de dos nodos hijo. Así, un árbol se denomina de orden m si cada nodo del árbol puede tener hasta m nodos hijo.

A.4 Búsqueda digital

Aquí aparecen los denominados *Trie* cuyo nombre viene del inglés *“information retrieval”*. Se trata de un árbol M-ario cuyos nodos son vectores *M-place* con componentes correspondientes a dígitos o caracteres. Cada nodo en nivel L representa el conjunto de todas las claves que empiezan con cierta secuencia de L dígitos o caracteres. El nodo especifica una rama *M-way* dependiendo del $(L+1)$ dígito o carácter.

El problema de los “Trie” es que ocupan bastante memoria, pero se puede reducir a costa de mayor coste computacional (listas enlazadas). Como en el esquema mostrado en la figura A.4, ahora la búsqueda consiste en un recorrido por comparación de igualdad/desigualdad, en lugar del mayor/menor de los árboles binarios, sobre cada elemento que conforma la cadena de las claves recorridas (por ejemplo, comparación bit a bit en series de números). De esta forma en media se hacen $log_2 N$ comparaciones. Como ejemplo, en la figura A.4 se almacenan las cadenas 1-2-5-10, 1-2-5-11, 1-2-6-12, 1-2-6-13, 1-3-7, 1-3-8 y 1-4-9.

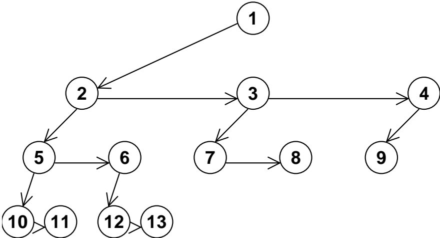


Figura A.4: Ejemplo de búsqueda digital

A.5 Búsqueda por hashing

Un algoritmo de *hash* es una función que mapea las claves en una posición única de una tabla o permite que distintas claves den como resultado la misma posición de la tabla que es lo que se denomina colisión.

El problema es conocer a priori los valores a almacenar. Si K es la clave y M el tamaño de la tabla de hash, las funciones típicas de hash $h(K)$ con $(0 \leq h(K) < M)$ son las siguientes:

- Basada en la división: $h(K) = K \, mod \, M$
- Basada en la multiplicación: $h(K) = \left\lfloor M \left((\frac{A}{w} K) \, mod \, 1 \right) \right\rfloor$ donde w es el tamaño de palabra del ordenador y A es un número primo relativo a w.

Existen diferentes alternativas para solucionar las colisiones:

- Una forma de resolución de colisiones es el llamado encadenado que consiste en colocar una lista enlazada en las posiciones con colisión.

- Otra forma consiste en almacenar en la misma tabla de hash las claves y las colisiones en posiciones adyacentes.
- Otra posibilidad consiste en almacenar las colisiones en la misma tabla pero en orden dependiente de una función relacionada con K.
- Finalmente, se puede utilizar un segundo hash para acceder a las colisiones del primer hash.

A.6 Resumen

No existe un algoritmo genérico de búsqueda que se demuestre claramente superior a los demás, ya que depende de las condiciones que uno u otro sea mejor. En la tabla A.1 se presenta la complejidad y costes de inserción/borrado para diferentes algoritmos.

Algoritmos	*Coste Búsqueda*	*Coste Insercion/Borrado*
Secuencial	$(N+1)/2$	Medio. Redimensionamiento del array
Binaria	$\log_2 N$	Medio-Alto. Redimensionar y reordenar el array
Árbol Binario	$\sqrt{N}$	Bajo
Digital	$\log_2 N$	Bajo
Hashing	1 sin colisiones	Bajo sin colisiones. Alto con colisiones, recalcular función de hash

Tabla A.1: Comparativa de algoritmos de búsqueda

De la tabla se puede deducir que los mecanismos de hash son los más eficientes si el conjunto de registros es estático. Sin embargo, cuando se necesita el poder añadir nuevos registros de manera dinámica, los árboles binarios y digitales se convierten en una mejor opción.

Para el caso específico de algoritmos de filtrado para sistemas de monitorización, habrá que usar una combinación de ellos según estemos tratando con datos estáticos o dinámicos.

Apéndice B

Consideraciones sobre la implementación de PAM-Tree

B.1 Introducción

En el presente apéndice vamos a dar una posible implementación del algoritmo de filtrado, cumpliendo la especificación del capítulo 2 pero ahora haciendo que los procesos de inserción, borrado, consulta y procesado de paquetes no se puedan intercalar.

B.2 Consideraciones

A la hora de implementar el nodo subfiltro, se puede mejorar la reutilización de filtrado si se divide el nodo subfiltro en dos partes: una primera que obtenga el valor intermedio a partir de *StartBit*, *StopBit* y *Function*, y una segunda parte para el *Value* y *ComparisonType*, de forma que si varios subfiltros tienen igual primera parte ésta se pueda reutilizar difiriendo sólo en la segunda. Así, de esta primera parte colgarán tantas segundas partes como combinaciones *Value* - *ComparisonType* existan entre los subfiltros con esa primera parte. Esta descomposición es muy útil porque en la definición de filtros en la práctica se suele comparar un campo contra muchos valores posibles, por ejemplo, para direcciones IP o puertos, matrices de tráfico, etc. Con esta alternativa se mejora aún más la eficiencia del subsistema de filtrado.

En la implementación del algoritmo de filtrado PAM-Tree se ha tenido en cuenta esta optimización del nodo subfiltro. Así, el *nodo subfiltro* se convierte en la unión de una estructura *MasterSubfilter* con una estructura *OutputSubfilter*. La primera lleva la definición de los bits del campo y la función para extraer su valor intermedio, mientras que la segunda contiene el valor con el que se ha de comparar el valor intermedio y el tipo de comparación a realizar. Se divide este nodo en dos partes para realizar una implementación más eficiente y sea suficiente la ejecución de un *MasterSubfilter* cuando existen varios *OutputSubfilters* que se refieren al mismo resultado intermedio. Esto es muy útil, porque en el funcionamiento habitual de un algoritmo de filtrado existen gran cantidad de filtros de este tipo. Por ejemplo, si se quiere una matriz de tráfico hará falta un *MasterSubfilter* único por dirección IP origen y dirección IP destino,

mientras que para almacenar las direcciones IP encontradas se utilizarán los *OutputSubfilter*, de los que habrá tantos como direcciones IP origen o destino. La estructura *Parameter* se corresponde directamente con el *nodo parámetro* del PAM-Tree.

Se utiliza una estructura de datos genérica *Map* que hace referencia a la estructura de datos necesaria para llevar cuenta de varios objetos del mismo tipo con una búsqueda, inserción y borrado eficientes. Proveerá métodos *Insert* para insertar un elemento, *Remove* para eliminar un elemento y *Find* para devolver el elemento que verifique una clave (por ejemplo de cierto *Value* en *OutputSubfilter* o de cierto *ID* en *Params*). Esta estructura se ha implementado en las herramientas comentadas en el capítulo 4 con una combinación de tablas hash y búsqueda lineal.

En la figura B.1 se muestran los elementos del árbol cuyos campos se detallan a continuación.

MasterSubfilter

- **int StartBit**: offset del bit inicial desde el comienzo de la cabecera del protocolo actual sobre el que se aplica el subfiltro.
- **int StopBit**: offset del bit final desde el comienzo de la cabecera del protocolo actual sobre el que se aplica el subfiltro.
- **bitset *Function**: función que produce la salida intermedia sobre la que realizar el testeo. Normalmente devolverá los bits entre *StartBit* y *StopBit* (*ValueFunction*), pero en otras ocasiones podrá realizar un procesado específico como *HeaderLengthFunction* que devuelve la longitud de la cabecera del protocolo actual o *FirstEncapsulationFunction* que devuelve un identificador para discernir el encapsulado de nivel de enlace cuando se pueden encontrar varios basándose en la propia cabecera del mismo nivel (ya que no existe ningún nivel por debajo que nos lo identifique).
- **Map OutputSubfilter MapOutputSubfilter**: estructura de acceso a los Output-Subilters que contienen los posibles valores del resultado intermedio en que se está interesado.
- **int Used**: contabiliza el número de parámetros que están usando este MasterSubfilter.

OutputSubfilter

- **int ComparisonType**: identifica el tipo de test que se va a realizar sobre el resultado intermedio que puede ser una de las operaciones siguientes: EQUAL (=), NOTEQUAL (! =), SMALLER (<), BIGGER (>) y DISCOVER (?).
- **bitset Value**: valor que comparado con el resultado intermedio del nodo MasterSubfilter al que está asociado, servirá para verificar la condición del subfiltro.
- **Map Parameter MapParameter**: estructura de acceso a los parámetros asociados que se actualizan si se verifica el test.

- **Map MasterSubfilter MapMasterSubfilter**: estructura de acceso a los subfiltros que son nodos hijos del nodo actual en el árbol. Se bajará por ellos si se verifica el test actual.
- **int Used**: contabiliza el número de parámetros que están usando este OutputSubfilter.

Parameter

- **unsigned long long Counter**: lleva cuenta del parámetro monitorizado.
- **int *UpdateFunction**: función que devuelve el valor a incrementar el contador según sea de bits, paquetes, autodescubrimiento, etc.
- **int Used**: contabiliza el número de parámetros que están usando este Parameter.
- **Param *RParam**: acceso a la entrada correspondiente Param.

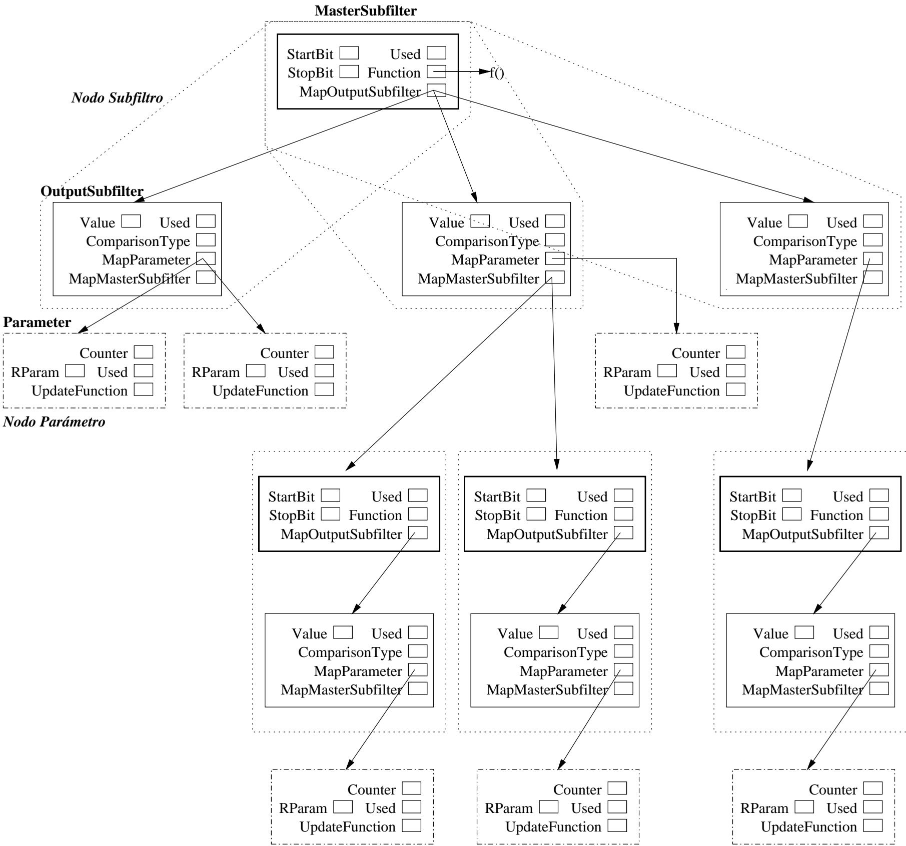


Figura B.1: Campos de los nodos de PAM-Tree

La figura B.2 es una representación esquemática de la estructura presentada en la figura B.1. En ella se distinguen los elementos fundamentales nodo subfiltro y nodo parámetro sin entrar en detalles de implementación. En la misma figura se presentan las variables globales de PAM-Tree y su relación con la estructura en árbol.

Variables globales

- **Map MasterSubfilter Root**: estructura Map que apunta a los nodos raíces del árbol.
- **Map Param Params**: estructura Map que apunta a los parámetros establecidos en el PAM-Tree, siendo una estructura de acceso rápido a los contadores asociados a cada nodo parámetro a partir de su identificador.
- **int Offset**: para el recorrido del árbol, guarda el offset del nivel de protocolo que se está procesando en la actualidad. Será un valor en bytes que permitirá que los valores de *StartBit* y *StopBit* de las estructuras MasterSubfilter del árbol se puedan definir en relación a este offset, es decir, desde el comienzo de la cabecera del nivel de protocolo correspondiente.

Param

- **ID**: identificador del parámetro.
- **Q**: definición completa del filtro asociado.
- **U**: campo auxiliar que permite a la función *UpdateFunction()* de un nodo parámetro comprobar si se está contabilizando cada paquete una sola vez en el caso de que se verifiquen varios componentes OR.
- **Map Counter Counters**: acceso directo a los contadores de los parámetros del árbol entre los que se hace operación OR.
- **Map Counter LastValues**: valor de los contadores en la consulta anterior para poder calcular las diferencias.

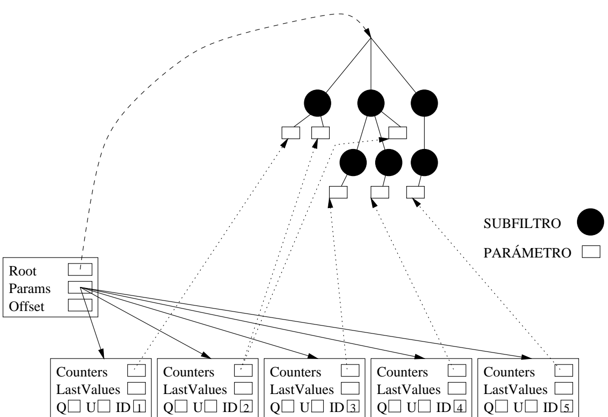


Figura B.2: Estructura esquemática del PAM-Tree de la figura B.1

B.3 Inserción, borrado, consulta y actualización de parámetros

Operación de inserción de parámetro

SetParam(Q)

```
{
id ← NewID();
FOR  all  F_{OR_j} ∈ Q.F,
    mapmastersubfilter ← Roots;
    FOR  all  F_i ∈ Q.F.F_{OR_j},
        mastersubfilter ← InsertMasterSubfilter(mapmastersubfilter, F_i);
        outputsubfilter ← InsertOutputSubfilter(mastersubfilter.MapOutputSubfilter, F_i);
        mapmastersubfilter ← outputsubfilter.MapMasterSubfilter;
    END
    parameter ← InsertParameter(outputsubfilter.MapParameter, Q.UpdateFunction);
    InsertParam(id, parameter, Q);
END
SEND_{entorno}(AnswerSetParam(id));
}
```

Operación de borrado de parámetro

DelParam(ID)

```
{
param ← FindParams(ID);
F ← param.Q.F;
FOR  all  F_{OR_j} ∈ F,
    mapmastersubfilter ← Roots;
    FOR  all  F_i ∈ F.F_{OR_j},
        mapoutputsubfilter ← RemoveMasterSubfilter(mapmastersubfilter, F_i);
        {mapmastersubfilter, mapparameter} ← RemoveOutputSubfilter(mapoutputsubfilter, F_i);
    END
    RemoveParameter(mapparameter, param.Q.UpdateFunction);
    RemoveParam(ID);
END
}
```

Operación de consulta de parámetro

PollParam(ID)

```
{
param ← FindParam(ID);
total ← 0;
FOR  i = 1 : length(param.Counters),
    total ← total + (param.Counters[i] − param.LastValues[i]);
    param.LastValues[i] ← param.Counters[i];
END
SEND_{entorno}(AnswerPollParam(ID, total));
}
```

Operación de actualización de parámetros

Packet(P)

```
{
Offset ← 0;
ProcessPacket(Roots, P);
}
```

Funciones utilizadas

MasterSubfilter NewMasterSubfilter(E,Function)
{
$new\, mastersubfilter;$
$mastersubfilter.StartBit = E.StartBit;$
$mastersubfilter.StopBit = E.StoptBit;$
$mastersubfilter.Function = Function;$
$mastersubfilter.MapOutputSubfilter = \{\};$
$mastersubfilter.Used = 1;$
$return \quad mastersubfilter;$
}

OutputSubfilter NewOutputSubfilter(Value,ComparisonType)
{
$new\, outputsubfilter;$
$outputsubfilter.Value = Value;$
$outputsubfilter.ComparisonType = ComparisonType;$
$outputsubfilter.MapParameter = \{\};$
$outputsubfilter.MapMasterSubfilter = \{\};$
$outputsubfilter.Used = 1;$
$return \quad outputsubfilter;$
}

Parameter NewParameter(UpdateFunction)
{
$new\, parameter;$
$parameter.Counter = 0;$
$parameter.UpdateFunction = UpdateFunction;$
$parameter.Used = 1;$
$parameter.RParam = \emptyset;$
$return \quad parameter;$
}

Param NewParam(ID,counter,Q)
{
$new\, param;$
$param.Id = ID;$
$param.Q = Q;$
$param.Counters = counter;$
$param.LastValues = \emptyset;$
$return \quad param;$
}

FreeMasterSubfilter(mastersubfilter)
/* Libera memoria para un objeto de tipo MasterSubfilter */

FreeOutputSubfilter(outputsubfilter)
/* Libera memoria para un objeto de tipo OutputSubfilter */

FreeParameter(parameter)
/* Libera memoria para un objeto de tipo Parameter */

FreeParam(param)
/* Libera memoria para un objeto de tipo Param */

MasterSubfilter InsertMasterSubfilter(MapMasterSubfilter,$\mathbf{F_i}$)

```
{
mastersubfilter ← FindMasterSubfilter(MapMasterSubfilter, F_i);
IF (mastersubfilter = NULL) THEN
    mastersubfilter ← NewMasterSubfilter(F_i.E, F_i.Function);
    MapMasterSubfilter = MapMasterSubfilter ∪ {mastersubfilter};
ELSE
    mastersubfilter.Used ← mastersubfilter.Used + 1;
END
return mastersubfilter
}
```

OutputSubfilter InsertOutputSubfilter(MapOutputSubfilter,$\mathbf{F_i}$)

```
{
outputsubfilter ← FindOutputSubfilter(MapOutputSubfilter, F_i);
IF (outputsubfilter = NULL) THEN
    outputsubfilter ← NewOutputSubfilter(F_i.Value, F_i.ComparisonType);
    MapOutputSubfilter = MapOutputSubfilter ∪ {outputsubfilter};
ELSE
    outputsubfilter.Used ← outputsubfilter.Used + 1;
END
return outputsubfilter;
}
```

Parameter InsertParameter(MapParameter,UpdateFunction)

```
{
parameter ← FindParameter(MapParameter, UpdateFunction);
IF (parameter = NULL) THEN
    parameter ← NewParameter(UpdateFunction);
    MapParameter = MapParameter ∪ {parameter};
ELSE
    parameter.Used ← parameter.Used + 1;
END
return parameter;
}
```

Param InsertParam(ID,parameter,Q)

```
{
param ← FindParam(ID);
IF (param = NULL) THEN
    param ← NewParam(ID, parameter.Counter, Q);
    Params ← Params ∪ {param};
    parameter.RParam ← parameter.RParam ∪ {param};
ELSE
    param.Counter ← param.Counter ∪ {parameter.Counter};
END
return param;
}
```

MasterSubfilter FindMasterSubfilter(MapMasterSubfilter,$\mathbf{F_i}$)

```
{
return{m ∈ MapMasterSubfilter : m.StartBit = F_i.StartBit ∧
∧ m.StopBit = F_i.StopBit ∧ m.Function = F_i.Function};
}
```

OutputSubfilter FindOutputSubfilter(MapOutputSubfilter,$\mathbf{F_i}$)

```
{
return{m ∈ MapOutputSubfilter : m.Value = F_i.Value ∧ m.ComparisonType = F_i.ComparisonType};
}
```

Parameter FindParameter(MapParameter,UpdateFunction)

```
{
return{m ∈ MapParameter :  m.UpdateFunction = UpdateFunction};
}
```

Param FindParam(ID)

```
{
return{m ∈ Params :  m.Id = ID};
}
```

MapOutputSubfilter RemoveMasterSubfilter(MapMasterSubfilter,F_i)

```
{
mastersubfilter ← FindMasterSubfilter(MapMasterSubfilter, F_i);
mapoutputsubfilter ← mastersubfilter.MapOutputSubfilter;
IF  (mastersubfilter.Used = 1)  THEN
      MapMasterSubfilter ← MapMasterSubfilter − {mastersubfilter};
      FreeMasterSubfilter(mastersubfilter);
ELSE
      mastersubfilter.Used ← mastersubfilter.Used − 1;
END
return  mapoutputsubfilter;
}
```

{MapMasterSubfilter,MapParameter} RemoveOutputSubfilter(MapOutputSubfilter,F_i)

```
{
outputsubfilter ← FindOutputSubfilter(MapOutputSubfilter, F_i);
mapmastersubfilter ← outputsubfilter.MapMasterSubfilter;
mapparameter ← outputsubfilter.MapParameter;
IF  (outputsubfilter.Used = 1)  THEN
      MapOutputSubfilter ← MapOutputSubfilter − {outputsubfilter};
      FreeOutputSubfilter(outputsubfilter);
ELSE
      outputsubfilter.Used ← outputsubfilter.Used − 1;
END
return  {mapmastersubfilter, mapparameter};
}
```

RemoveParameter(MapParameter,UpdateFunction)

```
{
parameter ← FindParameter(MapParameter, UpdateFunction);
IF  (parameter.Used = 1)  THEN
      MapParameter ← MapParameter − {parameter};
      FreeParameter(parameter);
ELSE
      parameter.Used ← parameter.Used − 1;
END
}
```

RemoveParam(ID)

```
{
param ← FindParam(ID);
param.Counters ← {};
FreeParam(param);
}
```

ProcessPacket(MapMasterSubfilter,P)

```
{
FOR  all  mastersubfilter ∈ MapMasterSubfilter,
```

```
    result ← mastersubfilter.Function(mastersubfilter.StartBit, mastersubfilter.StopBit, P);
    FOR  all  outputsubfilter ∈ mastersubfilter.MapOutputSubfilter,
        IF  Comparison(outputsubfilter.Value, result, outputsubfilter.ComparisonType)  THEN
            ProcessPacket(outputsubfilter.MapMasterSubfilter, P);
            IncrementCounters(outputsubfilter.MapParameter, P);
        END
    END
END
}
```

IncrementCounters(MapParameter,P)

```
{
FOR  all  parameter ∈ MapParameter,
    parameter.Counter ← parameter.Counter + parameter.UpdateFunction(offset, P, parameter.RParam);
END
}
```

Boolean Comparison(value,result,ComparisonType)

```
/* Devuelve el resultado de aplicar la comparación ComparisonType entre value y result */
{
CASE  ComparisonType,
    EQUAL : return (result = value);
    NOTEQUAL : return (result! = value);
    SMALLER : return (result < value);
    BIGGER : return (result > value);
    DISCOVER : return (TRUE)
END
}
```

Apéndice C

Publicaciones y proyectos relacionados con la Tesis Doctoral

C.1 Publicaciones

- Eduardo Magaña, Javier Aracil y Jesús Villadangos. PROMIS: A Reliable Real-Time Network Management Tool for Wide Area Networks. *Proceedings of IEEE Euromicro 98*, Volumen II, pag.581-588. Vasteräs, Suecia, Agosto 1998.

- Eduardo Magaña, Javier Aracil y Jesús Villadangos, Herramienta de Gestión en Tiempo Real de Redes de Área Extensa. *En actas del URSI'98, XIII Simposium Nacional*, pag.321-322. Pamplona, Septiembre 1998.

- Eduardo Magaña, J. Aracil, J. Villadangos y J.R. González de Mendívil. Reliable Network Management Tool through Internet. *Proceedings of Seventeenth IASTED International Conference on Applied Informatics*, pag.437-442. Innsbruck, Austria, Febrero 1999.

- Jose Javier Ruiz, Eduardo Magaña, Javier Aracil y Jesús Villadangos. Técnicas eficientes de filtrado y análisis de tráfico para la monitorización continua de redes de comunicaciones. *En actas de las II Jornadas de Ingeniería Telemática JITEL'99*, pag.215-223. Madrid, Septiembre 1999.

C.2 Trabajos pendientes de revisión

- Eduardo Magaña, Javier Aracil y Jesús Villadangos. A protocol-adaptive monitoring tree for efficient design of traffic monitoring probes. *Submitted to Computer Networks (Elsevier).*

C.3 Patente

Título:	**Sistema de monitorización de redes de comunicaciones empleando un método jerárquico de análisis de tráfico y almacenamiento de medidas de tráfico**
Inventores:	E Magaña, J. Villadangos, J. Aracil, J.J. Ruiz
Nº de solicitud:	P9901926
País de prioridad:	España
Fecha de prioridad:	17-9-1999
Entidad titular:	Universidad Pública de Navarra
Empresa/s que la están explotando:	VW-Navarra S.A., OPNATEL, RETENA

C.4 Proyectos

Título del proyecto:	**Sistema PROMIS de monitorización de redes de comunicaciones de área extensa**
Descripción:	Diseño, desarrollo e implantación de un sistema de monitorización continua de las redes de datos de Volkswagen Navarra S.A. mediante sondas sobre sistemas Linux distribuidas y una consola central (visual C++) para acceso a información de monitorización en tiempo real, capturas, informes periódicos y alarmas.
Tipo de contrato:	Contrato universidad-empresa
Entidad financiadora:	Volkswagen S. A.
Entidades participantes:	Volkswagen, Univ. Pública de Navarra
Duración:	Febrero 1999 - Diciembre 2001
Investigador responsable:	Javier Aracil y Jesús Villadangos, Univ. Pública de Navarra
Puesto:	Investigador

Título del proyecto:	**Migración de la red HFC de Retena a red multiservicio**
Descripción:	Dos líneas de trabajo. Por una parte un sistema de monitorización de las cabeceras de cablemódem CISCO con soporte Ethernet-Netflow, con sondas Linux distribuidas en cada cabecera y consola Java. Por otra parte, un servicio de vídeo bajo demanda MPEG-1 con el desarrollo de un servidor de vídeo y un cliente reproductor, con acceso integrado vía web.
Tipo de contrato:	Contrato Universidad/Empresa
Entidad Financiadora:	Redes de Telecomunicaciones de Navarra S. A.
Duración:	Mayo 1999 - Mayo 2001
Investigador responsable:	Javier Aracil y Jesús Villadangos, Univ. Pública de Navarra
Puesto:	Investigador, coordinador

Título del proyecto:	**Migración de redes HFC a redes multiservicio (Proyecto TIC 2FD97-0960-C05-04)**
Descripción:	Ayuda europea correspondiente al proyecto anterior.
Tipo de contrato:	Plan Nacional de I+D/FEDER
Entidad Financiadora:	Comisión Interministerial de Ciencia y Tecnología y Fondos FEDER Unión Europea.
Duración:	Mayo 1999 - Mayo 2001
Investigador responsable:	Javier Aracil Rico, Univ. Pública de Navarra
Puesto:	Investigador, coordinador

Título del proyecto:	**Sistema de monitorización OPMIS para redes de área extensa**
Descripción:	Implementación de un sistema de monitorización de redes de área local remotas unidas por RDSI con la red corporativa del Gobierno de Navarra con un interfaz web, y soporte de monitorización de Voz sobre IP para la red troncal ATM.
Tipo de contrato:	Contrato Universidad/Empresa
Entidad Financiadora:	OPNATEL, Obras Públicas y Telecomunicaciones de Navarra S. A.
Duración:	Septiembre 1999 - Marzo 2001
Investigador responsable:	Javier Aracil y Eduardo Magaña, Univ. Pública de Navarra
Puesto:	Investigador principal

Título del proyecto:	**ESTIGMA. Equipos y Soluciones de Transmisión para Internet2: Gestión, Monitorización y Acceso**
Tipo de contrato:	Contrato Universidad/Empresa + Proyectos de Investigación Científica y Desarrollo Tecnológico (Modalidad P4) Plan Nacional de I+D
Entidad Financiadora:	Intelnet y Plan Nacional de I+D
Duración:	Junio 2001 - Junio 2003
Investigador responsable:	Javier Aracil, Univ. Pública de Navarra
Puesto:	Investigador

C.5 Informes técnicos

- Eduardo Magaña Lizarrondo. Análisis de Aplicaciones en la Red de Chapa de Volkswagen Navarra. /VW/TR/001/4/F, 9/8/1997. Informe Técnico - Volkswagen Navarra S.A.

- Eduardo Magaña Lizarrondo. Análisis de Tráfico de la Ethernet Azul de la Nave de Chapa de VW Navarra. /VW/TR/002/2/F, 9/8/1997. Informe Técnico - Volkswagen Navarra S.A.

- Eduardo Magaña Lizarrondo. Análisis de Tráfico de la Ethernet Roja de la Nave de Chapa de VW Navarra. /VW/TR/003/1/F, 11/8/1997. Informe Técnico - Volkswagen Navarra S.A.

- Jesús Villadangos Alonso y Eduardo Magaña Lizarrondo. Analizador de redes distribuido PROMIS para la red de comunicaciones de VW-Navarra, S.A. Especificación de requisitos. Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A. PROMIS/TR/001. Septiembre 1998.

- Eduardo Magaña Lizarrondo y Jesús Villadangos Alonso. Análisis de tráfico de la red SINEC H1 de VW-Navarra, S.A. Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A. PROMIS/TR/002. Septiembre 1998.

- Eduardo Magaña Lizarrondo y Manuel Prieto Miguez. Sistema de monitorización MONET: Manual de la Sonda. Ref: /FEDER/TR/011. Fecha: Octubre 2000.

- Eduardo Magaña Lizarrondo y Manuel Prieto Miguez. Sistema de monitorización MONET: Manual de la Consola. Ref: /FEDER/TR/012. Fecha: Octubre 2000.

- Iñaki Astrain Escola, Eduardo Magaña Lizarrondo y Jesús Villadangos Alonso. Informe final de implantación del sistema de monitorización PROMIS en la red de Volkswagen Navarra S.A.. Ref: /PROMIS/TR/004. Fecha: Octubre de 2000, Estado: Final, Accesibilidad: Proyecto.

- Edurne Izkue Mendi y Eduardo Magaña Lizarrondo. Informe sondas-MONET de las semanas: 21-28 de Noviembre de 2000 y 28 de Noviembre-5 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Ref: /FEDER/TR/013. Fecha: 11 Diciembre 2000.

- Edurne Izkue Mendi y Eduardo Magaña Lizarrondo. Informe sondas-MONET de las semanas: 5-12 y 12-19 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Ref: /FEDER/TR/014. Fecha: 26 Diciembre 2000.

- Edurne Izkue Mendi y Eduardo Magaña Lizarrondo. Informe sondas-MONET de la semana 18-25 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Ref: /FEDER/TR/015. Fecha: 2 Enero 2001.

- Edurne Izkue Mendi y Eduardo Magaña Lizarrondo. Informe sondas-MONET de la semana: 25 de Diciembre de 2000 a 1 de Enero de 2001, Tráfico de la cabecera con sonda Ethernet y Netflow. Ref: /FEDER/TR/016. Fecha: 10 Enero 2001.

Bibliografía

[1] W. Stallings. *Local & Metropolitan Area Networks.* Prentice Hall, 5 ed., 1997.

[2] F. Kuo, W. Effelsberg, and J.J. Garcia-Luna-Aceves. *Multimedia Communications, Protocols and Applications.* Prentice Hall, 1998.

[3] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service, RFC 2212. Internet Engineering Task Force, September 1997.

[4] J. Bellamy. *Digital Telephony.* Wiley Series in Telecommunications. John Wiley & Sons, 2 ed., 1991.

[5] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, February 1997.

[6] V. Paxson and S. Floyd. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[7] V. Paxson and S. Floyd. Why We Don't Know How To Simulate the Internet. In *Proceedings of Winter Simulation Conference*, Atlanta, Georgia, December 1997.

[8] U. Black. *Computer Networks: Protocols, Standards, and Interfaces.* Prentice-Hall, 1993.

[9] E. Wilson. *Network Monitoring and Analysis: a protocol approach to troubleshooting.* Prentice-Hall, 2000.

[10] M. Naugle. *Network Protocols.* McGraw-Hill Series on Computer Communications, 1999.

[11] G. R. Ryan. The New Generation of Network Monitoring Systems. Technical report, ATG's Communications & Networking Technology Guide Series, 1997.

[12] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[13] Sun Microsystems. Network Interface Tap. SunOS 4.0 Documentation.

[14] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of Winter USENIX*, pages 259–269, San Diego, CA, January 1993.

[15] V. Jacobson, C. Leres, and S. McCanne. The tcpdump manual. Berkeley University, June 1997.

[16] S. McCanne, C. Leres, and V. Jacobson. Libpcap 0.4. ftp://ftp.ee.lbl.gov/libpcap.tar.Z, June 97.

[17] Libpcap y tcpdump versiones actualizadas. http://www.tcpdump.org/.

[18] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, Affordable, High Performance Statistics Collection. In *Proceedings of LISA*, pages 97–112, Chicago, September 1996.

[19] CAIDA: Cooperative Association for Internet Data Analysis. http://www.caida.org/.

[20] R. T. Braden. A pseudo-machine for packet monitoring and statistics. In *Proceedings of SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–209, Standford, CA, USA, August 1988.

[21] MRTG: Multi Router Traffic Grapher. http://ee-staff.ethz.ch/ oetiker/webtools/mrtg/mrtg.html, 1996.

[22] RRDtool: Round Robin Database Tool. http://www.caida.org/tools/utilities/rrdtool, 1999.

[23] NAM: Network Animator. http://www.isi.edu/nsnam/nam/, 1990.

[24] NS-2, The Network Simulator. http://www.isi.edu/nsnam/ns, 1989.

[25] T. Kitayama, A. Miyoshi, T. Saito, and H. Tokuda. Real-Time Communication in Distributed Environment - Real-Time Packet Filter Approach -. In *Proceedings of IEEE International Workshop on Real-Time Computing Systems and Applications (RTCSA)*, 1997.

[26] V. Paxson. On calibrating measurements of packet transit times. In *Proceedings of the ACM SIGMETRICS Intl. Conference on Measurement and Modelling of Computer Systems*, June 1998.

[27] J. W. Hong, S. Park, and Y. Kang. Enterprise Network Traffic Monitoring, Analysis and Reporting Using Web Technology. *Journal of Network and Systems Management*, 1999.

[28] M. Thottan and C. Ji. Proactive Anomaly Detection Using Distributed Intelligent Agents. *IEEE Network*, Sep/Oct 1998.

[29] Analizadores de Protocolos Comerciales.
http://www.netkb.com/analpric.htm.

[30] J. Case, M. Fedor, M. Simple, Schoffstall, and J. Davin. A Simple Network Management Protocol. Internet Standard STD0015-RFC1157, Internet Engineering Task Force, May 1990.

[31] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets: MIB-II. Internet Standard STD0017-RFC1213, Internet Engineering Task Force, March 1991.

[32] HP Openview. http://www.openview.hp.com/.

[33] IBM Netview/AIX. http://www.tivoli.com/products/index/netview/.

[34] Sun Solstice. http://www.sun.com/solstice/system.mgmt.html.

[35] Cabletron Spectrum.
http://www.aprisma.com/solutions/spectrum_enterprise/spectrum_enterprise.html.

[36] Castle Rock. http://www.castlerock.com/.

[37] Tkined/Scotty Network Management. http://wwwhome.cs.utwente.nl/ schoenw/scotty/, 1993.

[38] S. Waldbusser. Remote Network Monitoring Management Information Base Version 2 using SMI v2. Request For Comments 2021, Standards Track, Internet Engineering Task Force, January 1997.

[39] Distributed sniffer de network general.
http://www.sniffer.com/products/distributed/default.asp?A=1.

[40] NetScout de Acterna. http://www.acterna.com/products/netscout/index.html.

[41] Tivoli® Distributed Monitoring. http://www.tivoli.com/products/index/distmon/.

[42] Distributed Protocol Inspector de Fluke.
http://www.flukenetworks.com/lan/protocol+inspector/features/hardware+analyzers.htm.

[43] J. C. Mogul. Efficient Use of Workstations for Passive Monitoring of Local Area Networks. In *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, PA, September 1990.

[44] E. Magaña. BSD Packet Filter (BPF) como Herramienta de Monitorización de Red. Technical Report 99_TR_BPF, Área de Telemática, Depto. de Automática y Computación, Universidad Pública de Navarra, Junio 1999.

[45] D. Newman, T. Giorgis, and B. Melson. Probing RMON 2. Data Communications, May 1998.

[46] W. Stallings. *SNMP, SNMPv2, SNMPv3 and RMON 1 and 2*. Addison-Wesley, 3rd ed, 1999.

[47] S. McCanne. The Berkeley Packet Filter man page. ftp://ftp.ee.lbl.gov.

[48] M. Yuhara, B.N. Bershad, C. Maeda, and J.E.B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of Winter USENIX*, pages 153–165, San Francisco, CA, January 1994.

[49] M. L. Bailey, B. Gopal, M. A. Pagels, and L. L. Peterson. PathFinder: A Pattern-Based Packet Clasiffier. In *Proceedings of Symposium on Operating Systems Design and Implementation, Usenix Association*, November 1994.

[50] D. R. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols*, August 1996.

[51] D. Engler. VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[52] A.Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Harvard University, Cambridge, Massachusetts, September 1999.

[53] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficiente method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–25, 1989.

[54] J. R. B. Cockett and J. A.Herrera. Decision Tree Reduction. *Journal of the Association for Computing Machinery*, 37(4):815–842, October 1990.

[55] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An Imperative Language that Supports Declarative Programming. *ACM Trans. on Programming Languages and Systems*, 20(5):1014–1066, 1998.

[56] H. H. Tzeng and T. Przygienda. On Fast Address-Lookup Algorithms. *IEEE Journal on Selected Areas in Communications*, 17(6):1067–1082, June 1999.

[57] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (CIDR): An address assignment and aggregation strategy. RFC 1519. Internet Engineering Task Force, September 1993.

[58] R. Jain. Characteristics of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes. *Computer Networks and ISDN Systems*, 18:243–254, 1989/1990.

[59] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next-Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, February 2000.

[60] C. Labovitz, F. Jahanian, S. Johnson, R. Malan, S. R. Harris, J. Wan, M. Agrawal, and D. Zhu. Internet Performance Measurement and Analysis (IPMA) statistics. http://nic.merit.edu/ipma, 1998.

[61] Henry C. B. Chan, Hussein M. Alnuweiri, and Victor C. M. Leung. A Framework for Optimizing the Cost and Performance of Next-Generation IP Routers. *IEEE Journal on Selected Areas in Communications*, 17(6), June 1999.

[62] D. E. Knuth. *The Art of Computer Programming Vol.3: Sorting and Searching*. Addison-Wesley, 1973.

[63] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Volume 2: The Implementation*. Addison-Wesley Professional Computing Series, 1997.

[64] K. Sklower. A Tree-Based Packet Routing Table for Berkeley Unix. In *Proceedings of Winter USENIX*, pages 93–99, Dallas, TX, 1991.

[65] W. Doeringer, G. Karjoth, and M.Nassehi. Routing on Longest-Matching Prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[66] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 25–35, Cannes, France, 1997.

[67] S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE Journal on Selected Areas in Communications*, 17(6), June 1999.

[68] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 3–14, Cannes, France, 1997.

[69] T. Hayashi and T. Miyazaki. High-Speed Table Lookup Engine for IPv6 Longest Prefix Match. In *Proceedings of GLOBECOM*, pages 1576–1581, Río de Janeiro, 1999.

[70] B. Talbot. T. Sherwood and B. Lin. IP Caching for Terabit Speed Routers. In *Proceedings of GLOBECOM*, Río de Janeiro, 1999.

[71] Internet Engineering Task Force (IETF). http://www.ietf.org.

[72] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus. Requirements for Traffic Engineering Over MPLS. RFC2702. Internet Engineering Task Force, September 1999.

[73] J. Ryan. Multiprotocol Label Switching (MPLS). ATG's Communications & Networking Technology Guide Series, October 1998.

[74] X. Xiao and L. M. Ni. Internet QOS: A Big Picture. *IEEE Network*, pages 8–18, Mar/Apr 1999.

[75] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals.* Addison-Wesley, 2nd ed., 1999.

[76] R. T. Braden and A. L. DeSchon. NNStat: Internet Statistics Collection Package–Introduction and Users Guide. Technical Report ISI/RR-88-206, USC/Information Sciences Institute, August 1988.

[77] Nancy A. Lynch and Mark R. Tuttle. An Introduction to Input/Output Automata. CWI Quarterly, September 1989.

[78] A. V. Aho and J. D. Ullman. *Foundations of Computer Science.* Computer Science Press, 1992.

[79] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1989.

[80] J. P. Barthélemy and A. Guénoche. *Trees and Proximity Representations.* Wiley-Interscience, 1991.

[81] J. Postel. Internet Protocol (IP) - RFC791. Internet Engineering Task Force, September 1981.

[82] The Ethernet, a Local Area Network: Data Link Layer and Physical Layer Specifications. Digital Equipment Corporation, 1980.

[83] S. M. Ross. *Introduction to Probability Models.* Academic Press, 6ª edición, 1997.

[84] L. Kleinrock. *Queueing systems.* Wiley-Interscience, 1976.

[85] S. M. Ross. *Introduction to Probability Models.* Academic Press Limited, sixth edition, 1997.

[86] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers.* John Wiley and Sons, second edition, 1999.

[87] G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes.* Oxford Science Publications, second edition, 1994.

[88] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 1950.

[89] J. Schulist. Linux Socket Filter. Documentación del kernel Linux 2.2.14, filter.txt.

[90] W. Richard Stevens. *UNIX Network Programming Networking APIs: Sockets and XTI*, volume 1, 2ª ed. Prentice Hall, 1998.

[91] A. Cockcroft. How busy is the CPU, really? http://www.sunworld.com/unixinsideronline/swol-06-1998/swol-06-perf_p.html.

[92] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/.

[93] Linux Trace Toolkit. http://www.opersys.com/LTT/.

[94] FreeBSD. http://www.freebsd.org/.

[95] E. Magaña, J. Villadangos, J. Aracil, and J. Ruiz. Sistema de monitorización de redes de comunicaciones empleando un método jerárquico de análisis de tráfico y almacenamiento de medidas de tráfico. Patente española P9901926. Entidad titular: Universidad Pública de Navarra. Estado: revisión del estado del arte. Empresas que la están explotando: VW-Navarra S.A., OPNATEL, RETENA, Septiembre 1999.

[96] E. Magaña. Análisis de Aplicaciones en la Red de Chapa de Volkswagen Navarra. Technical Report /VW/TR/001/4/F, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Agosto 1997.

[97] E. Magaña. Análisis de Tráfico de la Ethernet Azul de la Nave de Chapa de VW Navarra. Technical Report /VW/TR/002/2/F, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Agosto 1997.

[98] E. Magaña. Análisis de Tráfico de la Ethernet Roja de la Nave de Chapa de VW Navarra. Technical Report /VW/TR/003/1/F, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Agosto 1997.

[99] J. Villadangos and E. Magaña. Analizador de redes distribuido PROMIS para la red de comunicaciones de VW-Navarra, S.A. Especificación de requisitos. Technical Report /PROMIS/TR/001, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Septiembre 1998.

[100] E. Magaña and J. Villadangos. Análisis de tráfico de la red SINEC H1 de VW-Navarra, S.A. Technical Report /PROMIS/TR/002, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Septiembre 1998.

[101] E. Magaña and I. Astrain. Informe final de implantación del sistema de monitorización PROMIS en la red de Volkswagen Navarra S.A. Technical Report /PROMIS/TR/004/1/F, Universidad Pública de Navarra, Dpto. de Automática y Computación - Volkswagen Navarra S.A., Octubre 2000.

[102] E. Magaña and M. Prieto. Sistema de monitorización MONET: Manual de la Sonda. Technical Report /FEDER/TR/011, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Octubre 2000.

[103] Netflow Services and Applications. Cisco Systems White Paper, http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm, June 2000.

[104] E. Magaña and M. Prieto. Sistema de monitorización MONET: Manual de la Consola. Technical Report /FEDER/TR/012, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Octubre 2000.

[105] Cisco uBR7246 - Universal Broadband Router, howpublished =Cisco Systems Product Overview, http://www.cisco.com/univercd/cc/td/doc/pcat/ub7246.htm, June 2000.

[106] E. Izkue and E. Magaña. Informe sondas-MONET de las semanas: 21-28 de Noviembre de 2000 y 28 de Noviembre-5 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Technical Report /FEDER/TR/013, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Diciembre 2000.

[107] E. Izkue and E. Magaña. Informe sondas-MONET de las semanas: 5-12 y 12-19 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Technical Report /FEDER/TR/014, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Diciembre 2000.

[108] E. Izkue and E. Magaña. Informe sondas-MONET de la semana 18-25 de Diciembre de 2000, Tráfico de la cabecera con sonda Ethernet y Netflow. Technical Report /FEDER/TR/015, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Enero 2001.

[109] E. Izkue and E. Magaña. Informe sondas-MONET de la semana: 25 de Diciembre de 2000 a 1 de Enero de 2001, Tráfico de la cabecera con sonda Ethernet y Netflow. Technical Report /FEDER/TR/016, Universidad Pública de Navarra, Dpto. de Automática y Computación - RETENA, Enero 2001.

[110] J. Aracil. Características del tráfico en la Internet e implicaciones para el análisis y dimensionamiento de redes de ordenadores. *NOVATICA*, (124), Nov./Dic. 1996.

[111] V. Paxson. Fast, Approximate Synthesis of Fractional Gaussioan Noise for Generating Self-Similar Network Traffic. *Computer Communications Review*, 27(5):5–18, October 1997.

[112] Motorola. MPC8260 : PowerQUICC II Integrated Communications Processor. http://e-www.motorola.com/webapp/sps/prod_cat/prod_summary.jsp?code=MPC8260&catId=M98657, 2000.

[113] Eric J. Hunter. *Classification made simple.* Aldershot (England), 1988.

[114] I. Balderjahn, R. Mathar, and M. Schader. *Classification, data analysis and data highways.* Springer-Verlag, 1998.