Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

Doctorado en Tecnologías de las Comunicaciones

TESIS DOCTORAL

CONTRIBUCIÓN AL DISEÑO DE ARQUITECTURAS DISTRIBUIDAS DE NODOS DE RED PROGRAMABLE

Autor: **MANUEL URUEÑA PASCUAL**
Licenciado en Informática

Director: **DAVID LARRABEITI LÓPEZ**
Doctor Ingeniero de Telecomunicación

Mayo de 2005

A Raquel

Resumen

Hoy en día, los nodos de red que forman Internet son complejos sistemas *hardware/software* que soportan un gran número de protocolos, servicios de red, o funcionalidades avanzadas como *firewall* o NAT. Sin embargo el proceso para añadir un nuevo protocolo o servicio es extremadamente largo y costoso, debido a múltiples causas, pero especialmente a que los *routers* siguen siendo sistemas propietarios, integrados verticalmente por los fabricantes.

En este sentido, la investigación en redes programables intenta simplificar el desarrollo y el despliegue de los servicios de red mediante la definición de interfaces abiertos entre todos los elementos que forman el *router*. Sin embargo hasta que los primeros diseños de nodos de red totalmente programables lleguen a comercializarse, es necesario aportar soluciones a corto y medio plazo que permitan ampliar las capacidades y servicios de los *routers* de alto rendimiento actuales.

Esta tesis presenta una arquitectura de nodo de red programable de transición y bajo coste, denominada *Simple Assistant-Router Architecture* (SARA), que permite extender las capacidades de un *router* comercial delegando el procesamiento avanzado de los paquetes a un *cluster* de "asistentes", lo que simplifica el desarrollo y despliegue dinámico de los nuevos servicios de red.

Un aspecto fundamental de esta arquitectura distribuida es la definición de mecanismos de coordinación de los asistentes entre sí y con el *router* legado. Para ello se propone la utilización del *Router-Assistant Protocol* (RAP), un protocolo de control que permite a los asistentes configurar el plano de datos del *router*, recibir eventos, así como desviar paquetes de señalización y flujos de datos para su procesamiento en los asistentes.

Dada la heterogeneidad de los requisitos de las aplicaciones de red es necesario proporcionar varios mecanismos para asegurar un reparto de carga efectivo en el *cluster* de asistentes. Esta Tesis Doctoral propone dos algoritmos de *Fast Robust Hashing* que permiten la asignación equitativa y persistente de flujos a asistentes, mejorando el rendimiento de las técnicas de *Robust Hashing* actuales, por lo que son lo suficientemente eficientes como para ser implementados en el plano de datos de un *router* comercial. Además, este trabajo especifica el *eXtensible Service Discovery Framework (XSDF)*, un marco de trabajo sencillo y escalable, que integra en un único proceso el descubrimiento de servicios y el reparto de carga entre servidores desacoplados.

Abstract

Nowadays, the network nodes that build Internet are complex hardware/ software systems, that support many signalling protocols, network services, and complex functionalities such as firewalling or NAT. However adding a new capability is a long, complex and costly process, due to many causes, but specially because routers are still proprietary systems, vertically integrated by the vendors.

In this sense, the research in programmable networks tries to simplify the development and deployment of network services by specifying open interfaces among all the elements that make up a router. However, before the first programmable network nodes start being deployed, it is necessary to provide short and medium term solutions that allow current high-performance routers to add advanced capabilities and new network services.

This PhD. Thesis presents a low-cost transition architecture for programmable network nodes named Simple Assistant-Router Architecture (SARA), that allows a commercial router to easily extend its capabilities by delegating the advanced packet processing to a cluster of "assistants", which greatly simplifies the development and dynamic deployment of new network services.

A key aspect of this distributed architecture is the need of several coordination mechanisms between the router and the assistants, and among assistant themselves. Therefore, the Router-Assistant Protocol (RAP) has been proposed, which is a control protocol based on ForCES, that allows assistants to configure the router's data plane, to notify events, as well as to divert signalling packets and data flows to the assistants.

As network application requirements could be very heterogeneous, it is necessary to provide several mechanisms in order to load-balance the assistant cluster. Thus, this Thesis presents two novel Fast Robust Hashing algorithms that provides a permanent and fair mapping of flows to assistants, and improves existing Robust Hash techniques as it is efficient enough to be implemented in the data plane of a commercial router. Moreover this research work also defines the eXtensible Service Discovery Framework (XSDF), which integrates in a single process: scalable service location, and load-sharing among lightly-coupled servers.

Índice general

Índice de figuras

Capítulo 1

Introducción y Objetivos

1.1. Introducción

El principio extremo a extremo es uno de los pilares básicos sobre los que se diseñaron las redes IP, y por extensión Internet. Según este principio, la mayoría de funcionalidades de un sistema distribuido debería residir en los sistemas finales, mientras que los sistemas intermedios deberían limitarse a realizar funcionalidades básicas, esencialmente al encaminamiento de paquetes, u otras tareas en las que la mejora del rendimiento así lo justifique. A pesar de que este razonamiento está ampliamente aceptado, actualmente existe una clara tendencia a incrementar el procesamiento que realizan los nodos de red, de modo que los *firewalls*, NAT, *proxies* web o balanceadores de carga son cada vez más populares entre los operadores de red.

Sin embargo, esta necesidad de desarrollar nuevos servicios de red, tales como la calidad de servicio, entornos de AAA o los citados anteriormente, choca con la arquitectura de los nodos de red actuales. Estos sistemas están integrados verticalmente, esto es, el fabricante distribuye tanto el *hardware* que se encarga de las comunicaciones, como el *software* que implementa el plano de control. Esta dependencia del fabricante limita la interoperabilidad y dificulta el desarrollo de nuevos servicios, puesto que para que puedan ser desplegados deben pasar por un largo y arduo proceso de estandarización y pruebas antes de ser soportados por el fabricante. Esta es una de las causas por las que mecanismos relativamente recientes, como IPv6 o Diffserv, todavía no han podido ser implantados a gran escala.

Ante esta situación, se inició la investigación en redes programables, bajo la cual surgieron dos líneas de investigación paralelas e interrelacionadas: las **redes activas** y la investigación en **interfaces abiertas**. La primera reconocía explícitamente la necesidad de realizar más procesamiento en los nodos intermedios de red y su objetivo era reducir al mínimo el tiempo de implantación de nuevos protocolos y servicios, definiendo una infraestructura de red común en la que los sistemas finales fuesen capaces de inyectar programas en la red y de esta forma adaptar el comportamiento de la misma. Por otro lado los organismos de estandarización han comenzado a definir interfaces estándar y abiertas entre los diferentes elementos y capas que forman un nodo de red. De esta forma se pretende fomentar la interoperabilidad entre fabricantes y simplificar el desarrollo de nuevos protocolos, que de este modo podrían adaptarse rápidamente a los sistemas de diferentes fabricantes.

En ambos casos se aboga por el diseño de nodos de red programables con capacidades extendidas y extensibles. Este planteamiento destaca frente a los diseños *hardware* de los *routers* de alto rendimiento actuales, que están formados por múltiples procesadores de propósito específico (muchas veces ASIC[1]), optimizados para encaminar paquetes a altas velocidades (el denominado *fast-path*), pero que tan sólo disponen de un procesador central, que normalmente no es demasiado potente, para tratar los casos especiales (*slow-path*), además del resto de las tareas de gestión y protocolos de encaminamiento del *router*.

Aunque las próximas generaciones de *routers* dispondrán de muchas funcionalidades adicionales y estarán diseñados para permitir un procesamiento por paquete más sofisticado y flexible, la gran mayoría de los *routers* desplegados actualmente posee una arquitectura optimizada y diseñada casi en exclusiva para el reenvío de datagramas.

Obviamente estas arquitecturas legadas no son la plataforma idónea para desplegar nuevos protocolos o servicios de red con grandes requisitos de computación, que requieran operaciones no disponibles en los ASIC del *fast-path* o que excedan la exigua capacidad de la CPU principal del *router*. Por esta razón es necesario diseñar arquitecturas de nodos de red programable que permitan a los *routers* de altas prestaciones tradicionales realizar procesamiento avanzado y especializado, más allá del reenvío de paquetes puro, de manera escalable.

[1] Applicacion-Specific Integrated Circuit, en inglés

1.2. Objetivos

Esta tesis doctoral propone una nueva arquitectura de nodo de red programable con carácter distribuido, es decir, constituido por varios componentes, que están poco acoplados para facilitar su implementación, escalabilidad y extensibilidad. En particular, esta propuesta se centra en ampliar las capacidades de los *routers* de altas prestaciones actuales delegando el procesamiento especial a elementos externos, denominados *asistentes*, que se basan en CPUs de propósito general de altas prestaciones. De este modo, todo el procesamiento que requiera una elevada capacidad de cómputo o de almacenamiento puede ser desviada por el *router* a un asistente, y de esta forma la CPU principal del *router* se limitará a las tareas esenciales de gestión del mismo.

Una característica muy importante de esta arquitectura es la capacidad del *router* para desviar de manera "transparente", esto es, sin la colaboración de los sistemas finales, paquetes de señalización y flujos de datos a los asistentes. De este modo se simplifica el despliegue de los servicios de red puesto que, a diferencia de otros mecanismos como las redes superpuestas[2], los sistemas finales no necesitan conocer a priori qué nodos ofrecen el nuevo servicio de red.

Esta arquitectura de nodo de red distribuido presenta importantes retos a la hora de conseguir un comportamiento coordinado entre el *router* y el asistente, pero que mantenga la independencia entre ambas entidades y que no requiera cambios significativos en el *router*. Para ello se proponen varios mecanismos de comunicación *router*-asistente, incluyendo un protocolo de control que permite a un asistente configurar el plano de datos del *router*.

El desarrollo del **protocolo de control router-asistente** puede encuadrarse dentro del campo de las redes programables, puesto que define una interfaz abierta entre el plano de control, que residirá parcialmente en el asistente, y el plano de datos del *router*. Para ello es necesario definir un modelo abstracto de los recursos del *router* e identificar las funcionalidades comunes. En este sentido, esta investigación se ha basado en la realizada dentro de foros como el *Network Processing Forum* (NPF) o el grupo de trabajo Forwarding and Control Element Separation (ForCES) del IETF. En ambos casos se trata de grupos de investigación activos, formados por un gran número de empresas y centros de investigación, lo que puede dar una idea del interés que despierta hoy en día esta línea de trabajo. De esta manera, otro objetivo de esta tesis es investigar la

[2] overlay networks, en inglés

relación entre la arquitectura propuesta y ForCES, identificando los elementos originales que contribuyan al desarrollo de ForCES.

Otro requisito muy importante de esta arquitectura es que pueda extenderse a varios asistentes, lo que permitiría ofrecer tolerancia a fallos y una alta escalabilidad de procesamiento. Además, la utilización de un ***cluster* de asistentes** permitiría el incremento paulatino de la capacidad del nodo de red en función de los nuevos servicios desplegados. Sin embargo, para poder disfrutar de estas ventajas es indispensable la utilización de diferentes mecanismos de reparto de carga efectivos a altas velocidades.

La posibilidad de emplear diferentes políticas de selección es esencial, puesto que el objetivo final es soportar cualquier tipo de servicio de red, y estos pueden tener unos requisitos de reparto de carga muy diferentes entre sí. Por ejemplo, los servicios con sesiones de trabajo largas seleccionarán un servidor cada bastante tiempo, mientras que otro tipo de servicios puede requerir la elección del mejor servidor por cada paquete recibido. Por tanto esta tesis doctoral estudia qué políticas de selección son las más adecuadas para diferentes tipos de servicio de red, y propone una **nueva política de selección** que permite desviar flujos de datos a los diferentes asistentes a velocidad de línea.

Aunque existe una extensa literatura sobre la aplicación de *clusters* de servidores en sistemas multiprocesadores o la utilización de balanceadores de carga en granjas de servidores web, los algoritmos de reparto de carga están especializados en dominios de aplicación particulares. En este caso, sin embargo, no es posible definir un único mecanismo de reparto de carga, puesto que un nodo de red programable debe ser capaz de soportar diferentes tipos de protocolos o de servicios de red, cada uno con sus propios requisitos de procesamiento y/o persistencia de estado y, por tanto, con necesidades de reparto de carga diferentes.

Esta es la razón por la que se propone un **nuevo mecanismo de reparto de carga extensible** para granjas de servidores genéricos. Sin embargo, dado el interés que hoy en día despierta la utilización de *clusters* de servidores desacoplados, el ámbito de aplicación de este mecanismo no está restringido a la arquitectura de nodo de red programable descrita, sino que su diseño le permite adaptarse a un contexto mucho más amplio, abarcando otros dominios de aplicación.

Adicionalmente, este trabajo propone la integración de los procesos de re-

parto de carga y de descubrimiento de servicios ya que, desde el punto de vista del autor, el reparto de carga puede considerarse simplemente como el último paso del proceso de localización de servicios. Además, aunque hoy en día los mecanismos de reparto de carga parten de la situación en la que el cliente conoce a priori la localización del servicio, es preferible emplear en su lugar un protocolo de **descubrimiento de servicios** que sea mucho más flexible y dinámico que el DNS.

La integración del reparto de carga dentro del proceso de descubrimiento de servicios es especialmente beneficiosa para su aplicación en granjas o *clusters* de servidores, puesto que evitaría la utilización de elementos intermedios, como los balanceadores de carga, que pueden limitar la escalabilidad del *cluster*. Además, la utilización de un protocolo de descubrimiento de servicios dinámico simplifica en gran medida la gestión de la granja de asistentes. Idealmente bastaría con arrancar una nueva aplicación de red o añadir un nuevo asistente, para que este se integrase en el *cluster* y, una vez que fuese detectado, el *router* sería capaz de utilizar los recursos adicionales inmediatamente. Todo ello sin necesidad de configuración del administrador, lo cual es especialmente útil para adaptarse rápidamente a incrementos súbitos del tráfico, que hoy en día son bastante habituales en Internet.

En resumen, este trabajo de investigación propone una arquitectura de nodo programable formado por un *router* legado y uno o más asistentes externos en cluster, a los que se desvían los flujos que requieren procesamiento adicional dentro de la red. Para que la cooperación entre el *router* y los asistentes sea realmente efectiva será necesario especificar algún tipo de protocolo de control *router*-asistente, así como emplear diferentes mecanismos de reparto de carga. Adicionalmente se analizará la relación entre los protocolos de reparto de carga y los de descubrimiento de servicios con el objetivo de integrar ambos mecanismos en un único proceso.

1.3. Organización de la Tesis

La memoria de la tesis se ha dividido en 7 capítulos. A continuación se ofrece un breve resumen de sus contenidos.

El capítulo 2 presenta el estado del arte de las áreas de investigación a las que pertenece esta tesis doctoral. En particular se evalúan las arquitecturas mo-

dernas de *routers* de altas prestaciones, que son la base y motivación principal de esta investigación. Posteriormente se presentan los trabajos más relevantes en el ámbito de las redes programables, incluyendo sus dos líneas de investigación predecesoras fundamentales: las redes activas y las interfaces abiertas. Además se hace especial énfasis en las arquitecturas y protocolos de nodos de red distribuidos, tales como GSMP o ForCES. Por último se analizan los principales mecanismos de reparto de carga y descubrimiento de servicios, centrándose en la comparación entre SLP y Rserpool, ya que estos son la base del mecanismo integrado de reparto de carga y descubrimiento de servicio propuesto por esta tesis.

El capítulo 3 explica detalladamente la arquitectura *Simple Assistant-Router Architecture* (SARA), que permite construir un nodo de red programable con despliegue dinámico de servicios a partir de un *router* de altas prestaciones, pero que por sí mismo no es lo suficientemente flexible para soportar los nuevos servicios que necesitan los operadores de red. Una vez definidos los principios de diseño, se analizan las características fundamentales de la arquitectura, tales como el desvío transparente de flujos o la utilización de múltiples asistentes.

El capítulo 4 se centra en evaluar diferentes mecanismos de cooperación entre el *router* y los asistentes, basándose en los requisitos de los diferentes tipos de servicios de red. En particular propone un nuevo protocolo denominado *Router-Assistant Protocol* (RAP), que permite acceder a todas las funcionalidades del plano de datos de un *router* desde un asistente externo, empleando el modelo abstracto de *router* definido por ForCES. Para ello ha sido necesario desarrollar partes de la arquitectura que aún no han sido definidas por el grupo de trabajo ForCES. Además se detallan los diferentes mecanismos de reparto de carga que pueden aplicarse en el *cluster* de asistentes, así como el uso de RAP para intercambiar paquetes de datos entre el *router* y los asistentes, incluyendo la meta-información necesaria para su procesamiento. El capítulo finaliza con tres ejemplos de uso que permiten ilustrar el funcionamiento de los mecanismos de cooperación *router*-asistente.

Ante la necesidad de mecanismos de reparto de carga de alta velocidad para el plano de datos de los *routers*, el capítulo 5 analiza diferentes algoritmos de *hashing* y propone un nueva familia de técnicas, denominada *Fast Robust Hashing*, que intenta suplir las carencias detectadas en los mecanismos actuales.

El capítulo 6 analiza el problema general del reparto de carga, y analiza su relación con el descubrimiento dinámico de servicios. A partir de dichas ob-

servaciones se presenta el *eXtensible Service Discovery Framework* (XSDF), un marco para el descubrimiento de servicios, que integra el reparto de carga entre servidores replicados como un paso más del descubrimiento de servicios, y que puede aplicarse a un gran número de escenarios.

Por último, el capítulo 7 resume las aportaciones de esta tesis y ofrece la principales conclusiones de la investigación realizada, así como las posibles líneas de trabajo abiertas a raíz de la misma.

Capítulo 2

Estado del arte

Este capítulo analizará los trabajos más relevantes relacionados con los objetivos de investigación que se desarrollan en esta tesis doctoral. En particular, se estudiarán temas como las arquitecturas de los *routers* actuales, la investigación en redes programables, el reparto de carga en *clusters* de servidores desacoplados, y por último mecanismos de descubrimiento de servicios.

2.1. Arquitectura de los *routers*

Los encaminadores, o *routers* en inglés[1], son la columna vertebral de las redes de paquetes IP y por extensión de Internet. Por tanto las, funcionalidades ofrecidas por este tipo de redes dependen esencialmente de las características y del rendimiento de estos dispositivos.

Uno de los principios básicos del diseño de las redes IP es el principio extremo a extremo [211], según el cual la mayor parte de la funcionalidades de la comunicación, por ejemplo la fiabilidad, debería residir en los sistemas finales, mientras que la red debería ofrecer simplemente un servicio de transporte de datos no fiable entre los extremos de la comunicación. Aunque los *routers* reales implementan numerosas funcionalidades [22], bajo este modelo informal y extremadamente simplificado, podría considerarse que los *routers* realizan esencialmente dos funciones:

[1] A lo largo de este trabajo se emplearán ambos términos indistintamente

- El reenvío[2] de paquetes IP entre sus interfaces, basado en la dirección destino. Por cada datagrama IP se consulta la tabla de rutas que indica, dada una dirección IP, cual es la ruta más adecuada para alcanzar la subred más específica a la que pertenece dicha dirección.

- La ejecución de algún tipo de protocolo de encaminamiento[3] con otros *routers* para conocer la topología actual de la red y de esta forma mantener la información de la tabla de rutas actualizada.

La función de encaminamiento pertenece al plano de control mientras que el plano de datos se encarga del reenvío de los datagramas, aunque a veces también se emplea el término "encaminamiento de paquetes". A diferencia de otras tecnologías de conmutación [163] donde la decisión de encaminamiento se realiza una vez por conexión, la conmutación de paquetes requiere que se consulte la tabla de rutas por cada datagrama IP reenviado. A este proceso se le denomina *route lookup* y en buena medida es el responsable del rendimiento global del *router*.

El proceso de *route lookup* consiste en encontrar la ruta más adecuada para alcanzar la dirección IP destino. Este proceso de búsqueda es relativamente laborioso ya que la tabla de rutas no tiene una entrada por cada dirección IP posible, sino que cada entrada indica cómo alcanzar a una subred. Además, las subredes pueden contenerse unas a otras y en ese caso debe escogerse la entrada más específica, es decir, aquella que comparta el prefijo de mayor longitud[4] con la dirección IP buscada.

Inicialmente, el rendimiento de las redes IP estaba limitado por el ancho de banda de los enlaces, de forma que los *routers* podían implementar el proceso reenvío de paquetes en *software* y por tanto era posible emplear, con ciertas modificaciones [162], las estaciones de trabajo de la época.

En la figura 2.1 se muestra un esquema simplificado del plano de datos[5] de un *router software*. Cuando se recibe un paquete por una de las interfaces del *router*, se busca en la tabla de rutas la información de encaminamiento (interfaz de salida y siguiente salto) más específica para la dirección IP destino del paquete (en este caso el prefijo `01*` para la dirección destino `011`) y el paquete se encola en la interfaz de salida seleccionada, para que pueda ser transmitido.

[2] Forwarding, en inglés
[3] Routing, en inglés
[4] Longest Prefix Match, en inglés
[5] Data path, en inglés

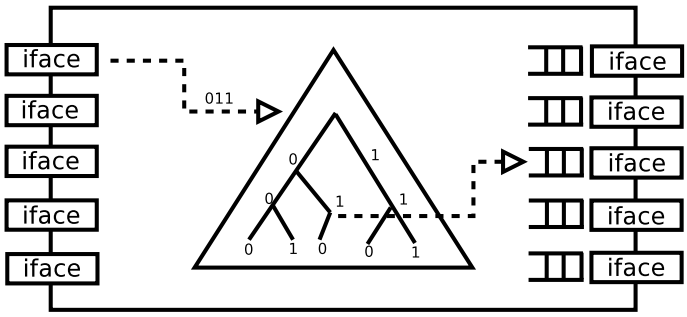


Figura 2.1: Esquema funcional de un *router software*

Conforme aumentaba el ancho de banda de las interfaces de red, el *route lookup* comenzó a convertirse en un cuello de botella, y aunque se han propuesto una gran número de algoritmos [41, 208], estructuras de datos (e.g. Tries), o caches para reducir el número de accesos a memoria, y de esta forma acelerar el proceso de reenvío, los *routers* de altas prestaciones comenzaron a emplear arquitecturas *hardware* especializadas en el reenvío de paquetes IP.

Una de las alternativas para acelerar el reenvío de paquetes consiste en implementar el proceso de *route lookup* en *hardware*. Para ello pueden diseñarse ASICs especializadas [131] o emplear memorias TCAM[6] que permiten comparar una dirección IP con múltiples rutas simultáneamente. En ciertos casos los circuitos *hardware* no son capaces de realizar todas la posibles etapas del reenvío de datagramas (por ejemplo el tratamiento de las opciones IP) y tan sólo implementan la secuencia de etapas más habituales (el denominado *fast-path*). En estos casos, los paquetes que requieren un procesamiento especial son tratados por el procesador central, lo que usualmente lleva parejo una penalización considerable en el rendimiento del *router* (*slow-path*).

Sin embargo los avances en transmisión óptica han disparado el ancho de banda disponible por interfaz y hoy en día es necesario emplear arquitecturas con múltiples procesadores para paralelizar el reenvío de paquetes [132, 47].

Típicamente en un *router* multi-procesador todas las operaciones del plano de control, tales como las tareas de gestión o los protocolos de encaminamiento, se ejecutan en un procesador central de propósito general, mientras que el plano de datos se distribuye entre varios procesadores de propósito específico, interconectados por una o varias matrices de conmutación[7] (e.g. buses de *backplane*, redes de Clos, etc) [41].

[6] Ternary Content Addressable Memory, en inglés
[7] Switching Fabric, en inglés

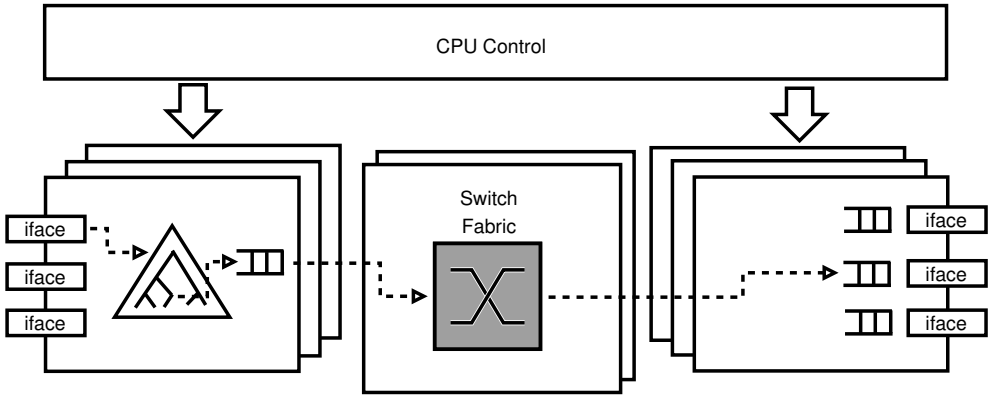


Figura 2.2: Esquema básico de un *router* multi-procesador

En este tipo de arquitecturas el *router* está compuesto por varias tarjetas procesadoras que contienen: varias interfaces de línea, un procesador que realiza el *route lookup* y la lógica necesaria para acceder a la red de interconexión. De este modo cada procesador de reenvío se encarga tan sólo de un subconjunto de todas las interfaces del *router*[8]. Por cada paquete que recibe por alguna de sus interfaces, busca en la tabla de rutas local la interfaz de salida y el siguiente salto. Si la interfaz también se encuentra dentro de la misma tarjeta procesadora, se reenvía localmente. En otro caso es necesario enviar el paquete a la tarjeta procesadora destino a través de la red de interconexión, tal como se muestra en la figura 2.2. Por último el paquete se encola en la interfaz de salida a la espera de poder ser transmitido. Este es el caso más general para sistemas con *buffering* a la entrada y a la salida, aunque otras arquitecturas presentan *buffering* exclusivamente a la entrada o a la salida.

En cuanto al plano de control, la CPU principal tiene que encargarse de las tareas de gestión (e.g. SNMP) y de ejecutar los diferentes protocolos de encaminamiento para mantener la tabla de rutas actualizada. Además debe distribuir una copia de la tabla de rutas, que se ha generado a partir de los protocolos de encaminamiento, a todas las tarjetas procesadoras.

Esta arquitectura permite reenviar simultáneamente tantos paquetes como tarjetas procesadoras tenga el *router*, a costa de complicar su diseño al requerir mecanismos de planificación de la red de interconexión interna, y de gestión de la memoria distribuida [32]. A diferencia de las soluciones anteriores donde se empleaba un esquema de memoria centralizada, en este tipo de arquitecturas multi-procesador cada tarjeta procesadora tiene sus propios bancos de memoria,

[8] Algunas arquitecturas [182] también permiten que el paquete se desvíe a otra tarjeta procesadora menos cargada.

de forma que muchas veces es necesario ir almacenando los paquetes en diferentes tarjetas y transferirlos a través de la red de interconexión (generalmente segmentados en células de tamaño fijo), aunque a veces es posible enviar tan sólo la cabecera y la meta-información asociada para evitar copias innecesarias [140].

Obviamente estas arquitecturas tan complejas tan sólo se emplean en los *routers* de gama alta, orientados a las redes de operador, y que necesitan encaminar millones de paquetes por segundo. Por el contrario en las redes de acceso, el ancho de banda de los enlaces suele ser bastante limitado y pueden emplearse arquitecturas de memoria compartida mucho más sencillas. Por tanto los *routers* pueden clasificarse en:

***Routers* Frontera** [9]. Están situados normalmente en el borde de las organizaciones para dar acceso a sedes remotas o al exterior, y suelen disponer de enlaces WAN con un ancho de banda medio o bajo. El reenvío de paquetes suele realizarse en *software* y en algunos casos con soporte *hardware* parcial.

***Routers* de Red Troncal** [10]. Pertenecen a los operadores de red y forman el núcleo de su infraestructura de red. Suelen tener enlaces ópticos con un gran ancho de banda, aunque normalmente están sobre-dimensionados. El reenvío de paquetes se implementa casi en su totalidad en el *hardware* del sistema.

A pesar de que esta clasificación puede basarse puramente en el rendimiento de encaminamiento de los *routers*, hoy en día esta diferenciación se refiere esencialmente a las funcionalidades adicionales de cada tipo de *router*. En los *routers* troncales es esencial mantener una tasa de reenvío elevada, y por tanto el procesamiento de los paquetes se reduce a lo mínimo imprescindible. En cualquier caso, este tipo de *routers* suele disponer de mecanismos de calidad de servicio como Diffserv, de control de congestión como *Random Early Detection* (RED) y *Explicit Congestion Notification*(ECN), o de ingeniería de tráfico, basada en típicamente *Multiprotocol Label Switching* (MPLS).

En el otro extremo se encuentran los *routers* frontera, donde el rendimiento de reenvío no es un requisito crítico sino que las organizaciones necesitan un gran número de funcionalidades suplementarias a la transmisión de paquetes pura.

[9] Edge Routers, en inglés
[10] Core Routers, en inglés

Es bastante habitual que estos equipos sean multi-protocolo (e.g. IP, IPX) y se utilicen para implementar políticas de seguridad perimetral. Las funcionalidades más habituales son: reglas de control de acceso (ACLs), *firewalls* con estado, filtros anti-*spoofing* [51], túneles IPSec cifrados para construir redes privadas virtuales (VPNs), traducción de direcciones (NAT), monitorización, priorización de paquetes para ofrecer QoS en los enlaces de acceso y un largo etcétera.

La tendencia actual consiste en añadir aún más servicios a los *routers* frontera y por tanto los vendedores se encuentran en la disyuntiva de elegir plataformas basadas en *software*, fáciles de programar y actualizar, o soluciones *hardware* que permiten realizar un gran número de operaciones por paquete. Así, algunos fabricantes [133] han comenzado a producir procesadoras específicas para realizar procesamiento avanzado (túneles cifrados, NAT...) por paquete, a altas velocidades. Además, estos ASIC especializados se montan en tarjetas procesadoras como si fueran una interfaz más, de modo que es posible ampliar las capacidades de los *routers* multi-procesador basados en *hardware*. Los paquetes se reenvían normalmente, pero antes de llegar a la interfaz de salida se desvían a la tarjeta de servicios para ser procesados.

Los procesadores de red[11] son una alternativa intermedia entre la versatilidad de los *routers software* y el rendimiento de las soluciones basadas es ASICs. Los *Network Processors* han sido diseñados por los principales fabricantes de procesadores (e.g. Intel [2] e IBM [8]) para aprovechar las ventajas de la economía de escala en el floreciente mercado de los conmutadores y *routers*. En cuanto a su programabilidad, los *Network Processors* son similares a las CPUs de propósito general tradicionales, pero están optimizados para el procesamiento de paquetes a altas velocidades. Disponen de juegos de instrucciones especializados, tienen unidades *hardware* para gestionar las colas de tx/rx, memorias de alta velocidad, co-procesadores [194] para realizar operaciones de *hash*, búsquedas en árbol o basadas en TCAMs, y múltiples micro-unidades de procesamiento [58] que son capaces de ejecutar múltiples hilos simultáneamente, para solapar el procesamiento de paquetes durante los accesos a memoria.

Justamente el acceso a memoria se está convirtiendo en un aspecto fundamental del rendimiento de los *routers*. Por ejemplo una interfaz 10 Gigabit Ethernet puede recibir hasta 15 millones de paquetes de tamaño mínimo por segundo, lo que implica que el *router* debería ser capaz de procesar cada paquete en menos de 66 nanosegundos [2]. Para ello es imprescindible paralelizar el

[11] Network Processors, en inglés

procesamiento de paquetes y utilizar memorias de baja latencia (SRAM) y de alta velocidad, con capacidades de *pipeline*, como la Rambus DRAM.

2.2. Redes Programables

Tal como se ha descrito en el apartado anterior, los *routers* actuales son sistemas *hardware* y *software* extremadamente complejos, optimizados para conseguir altas prestaciones, y que cada vez ofrecen más funcionalidades y soportan un mayor número de protocolos.

Sin embargo, aunque estas capacidades adicionales son cada vez más importantes para que los fabricantes de equipamiento puedan diferenciarse de la competencia, el desarrollo de nuevos servicios de red o el despliegue de nuevos protocolos es un proceso extremadamente complejo, largo y costoso. Tanto es así que se estima que protocolos como IPv6, que ya está estandarizado en su mayor parte, pueden tardar del orden de 10 años más antes de llegar a implantarse masivamente.

Este aspecto no ha cambiado desde los inicios de las redes de datos y se debe a numerosos factores: la incompatibilidad entre diferentes fabricantes, la integración vertical de los nodos de red, la dependencia de los fabricantes de equipamiento, etc. De hecho actualmente se considera que este es uno de los grandes problemas de las redes de datos y por tanto existe un gran interés en soluciones que consigan simplificar el desarrollo y despliegue de nuevos protocolos y servicios de red.

A partir de este problema ha surgido todo un área de investigación que aboga por incrementar la programabilidad de los nodos de red, para simplificar el despliegue de los nuevos protocolos y fomentar la interoperabilidad entre fabricantes de componentes *hardware* y *software* de *routers*. A todas estas iniciativas con este objetivo común se las conoce por el término genérico de "Redes Programables". Dentro de este amplio campo de investigación existen dos grandes líneas de trabajo, que aunque intentan resolver el mismo problema, lo enfocan desde puntos de vista diferentes:

Redes Activas. Esta línea de investigación ha explorado las posibilidades de un paradigma totalmente nuevo para las redes de datos, en el cual los nodos de red tradicionales se convierten en plataformas de computación abiertas. De esta forma los nodos de red, al ser totalmente programables,

permiten el despliegue dinámico de los servicios de red.

Interfaces Abiertas. Este tipo de propuestas también intentan incrementar la programabilidad de los nodos de red, pero se centran en la estandarización de interfaces y protocolos entre las diferentes capas y elementos que forman un *router*. De esta forma se intenta simplificar el desarrollo multi-plataforma y permitir un mayor grado de interoperabilidad entre fabricantes.

En las siguientes secciones se estudian en detalle ambas líneas de investigación, así como los principales trabajos dentro de cada una de ellas.

2.3. Redes Activas

La motivación principal de la investigación sobre las denominadas "Redes Activas" consiste en simplificar el despliegue de nuevos protocolos en las redes de comunicaciones. Para ello es necesario dotar a los nodos de red con capacidades adicionales:

- Una mayor capacidad de proceso para implementar nuevos protocolos y servicios de red.
- Mecanismos para la distribución y carga dinámica del código de los nuevos protocolos.

Sin duda, la propuesta para aumentar la capacidad de procesamiento en los nodos de red ha sido la característica más revolucionaria de las redes activas, puesto que permitiría, no sólo el transporte de datos como una red tradicional, sino además realizar un procesamiento de alto nivel sobre los mismos dentro de la red.

Por tanto una "Red Activa" [236] puede definirse como una red de paquetes para el transporte de datos, formada por sistemas finales, enlaces y nodos de red, que permite el despliegue rápido de nuevos protocolos y servicios de red, mediante la distribución y carga dinámica de código. Para ello la red está formada por "nodos activos" que permiten realizar un procesamiento personalizado sobre los paquetes de datos, más allá del encaminamiento de los mismos.

Existe un gran número de propuestas de redes activas [235, 36, 196] pero todas son variaciones de esta definición general en las que se han tomado unas

decisiones de diseño determinadas, referidas a la capacidad de computación de los nodos activos y a la flexibilidad de los mecanismos de distribución de código empleados.

Por ejemplo, las primeras investigaciones se centraron en desarrollar mecanismos para permitir que fueran los propios usuarios finales los que pudiesen inyectar nuevos servicios en la red y definir protocolos para encaminar sus paquetes. Para ello se empleaban los denominados "paquetes activos" que además de transportar los datos de aplicación como las PDU tradicionales, indicaban a los nodos activos como debían ser procesados y encaminados. En el extremo de la flexibilidad total algunos autores proponían que los paquetes, denominados "cápsulas", contuviesen código móvil con las rutinas necesarias para su encaminamiento, de modo que los nodos activos se limitarían a ejecutar dicho código al recibir una cápsula. Algunos ejemplos de esta aproximación son los Smart packets de BBN [216] o la opción Active IP [255].

Obviamente, la utilización de código móvil conlleva importantes problemas de seguridad [6]. Un administrador de red no puede permitir que cualquier usuario de Internet pueda ejecutar cualquier programa que desee en los *routers* de su red (e.g. un bucle infinito). Por tanto es indispensable comprobar que el código móvil cumple diversas propiedades antes de ejecutarlo. La investigación en este campo ha sido muy intensa y ha llevado al estudio de comprobaciones dinámicas de código [217], limitar el uso de recursos o primitivas [5], y lenguajes intrínsecamente seguros [109], sin bucles ni funciones recursivas. En cualquier caso la flexibilidad total en la distribución de código requiere sacrificar cierto grado de procesamiento de los paquetes activos (funcionalidades o rendimiento) para obtener un nivel de seguridad aceptable.

En el otro extremo, algunos autores consideran que tanta flexibilidad es excesiva y/o peligrosa y que en su lugar debería darse prioridad al procesamiento sin restricciones de los paquetes. En ese caso el código activo sería supervisado e instalado dinámicamente por los administradores de red en los nodos activos (e.g. desde servidores de código fiables), y por tanto los paquetes activos sólo indican qué servicio del proveedor de red desean emplear para ser procesados [28]. DAN [67], Switchware [5], y la arquitectura SARA [148], propuesta en esta tesis, optan por esta alternativa de diseño.

Dado el gran número de soluciones propuestas, el grupo de trabajo de DARPA sobre redes activas diseñó un modelo de referencia. Según dicha arquitectura (figura 2.3) un nodo activo está formado por tres niveles:

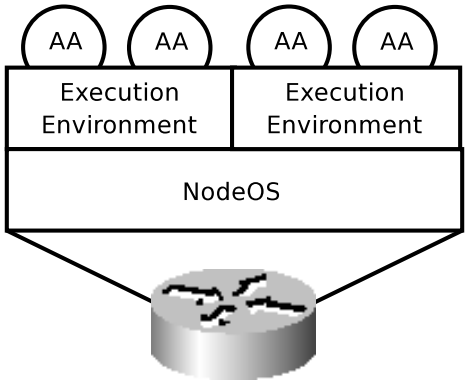


Figura 2.3: Arquitectura de un Nodo de Red Activo

1. Las Aplicaciones Activas[12] (AA) implementarían los servicios de red o protocolos que ofrece el nodo activo, esto es, son las entidades encargadas de procesar y encaminar en última instancia los paquetes activos que llegan al nodo.

2. El Entorno de Ejecución[13] (EE) gestiona el ciclo de vida de las Aplicaciones Activas, es decir, la carga dinámica de su código, y proporcionan a las AAs una interfaz de programación (API) que les permite utilizar las diversas funcionalidades del nodo, o acceder a la configuración y al estado del mismo.

3. Por último, el NodeOS [193] es el sistema operativo del nodo de red y, como tal, proporciona una abstracción del *hardware* subyacente, y gestiona sus recursos físicos, tales como la CPU y la memoria, multiplexándolos entre los diferentes Entornos de Ejecución, en el caso de existir más de uno.

La decisión de soportar múltiples Entornos de Ejecución en cada nodo activo se debe al amplio abanico de soluciones propuestas por la comunidad investigadora, estando muchas de de ellas especializadas en algún subtipo de aplicación activa, como por ejemplo la gestión de red [216, 156, 200]. Evidentemente la cohabitación de Entornos de Ejecución requiere que el NodeOS sea capaz, no sólo de repartir los diferentes recursos físicos entre ellos, sino también de entregar los paquetes activos recibidos al EE adecuado. Para ello también se definió el *Active Network Encapsulation Protocol*(ANEP) [7] que fue la base de la red de investigación *Active Network Backbone*(ABONE) [1].

[12] Active Applications, en inglés
[13] Execution Environment, en inglés

Al igual que el MBONE sirve como plataforma de pruebas para el tráfico *multicast*, la red ABONE ha sido la plataforma de investigación por excelencia de la comunidad de redes activas. Como el resto de redes de investigación, la ABONE es una red virtual, que emplea túneles UDP para comunicar los nodos activos entre sí.

Esta decisión de crear una red activa superpuesta a Internet ha tenido bastante importancia en la evolución de las redes activas. En dicha red, los nodos activos se conocen entre sí (al menos los nodos adyacentes) y se envían los paquetes activos directamente a sus direcciones IP. Por tanto ABONE está totalmente desligada de la red IP subyacente, que tan sólo se emplea como mecanismo de transporte. Si bien este modo de funcionamiento es razonable, y ha sido utilizado en repetidas ocasiones por otras redes de investigación, tiene una consecuencia clave: de popularizarse, todos los *routers* de Internet deberían ser nodos activos para poder aplicar los mecanismos de comunicación directa empleados en la ABONE.

Obviamente este requisito complicaría enormemente la convergencia gradual entre Internet y las redes activas. Sin embargo, las investigaciones para la integración del paradigma de las redes activas en Internet han sido relativamente tardías. Estos grupos de investigación también proponen la utilización de los paquetes activos pero buscan la transparencia de los nodos activos, esto es, que las aplicaciones finales no tengan que conocer al primer nodo activo de la red (a diferencia de ABONE) sino que envíen los paquetes directamente al destino, como cualquier otro paquete IP. El hecho de que cada nodo activo no necesita conocer a los otros nodos activos adyacentes simplifica la instalación de nodos activos en Internet, y evita la necesidad de desplegar nuevos protocolos de encaminamiento entre nodos activos.

Para ser transparentes, los *routers* activos deben ser capaces de diferenciar los paquetes activos entre el flujo normal de paquetes IP que reenvía. Para ello se han propuesto dos técnicas: la utilización de la opción *Router Alert*[14] de la cabecera IP [135, 188] de los paquetes activos [216, 252], o la aplicación de reglas de filtrado sobre todos los paquetes IP reenviados para localizar a los paquetes activos y entregarlos al Entorno de Ejecución adecuado [81].

[14] Esta opción se emplea en protocolos de señalización como RSVP

2.3.1. Principales grupos de investigación

Inicialmente, la investigación de las redes activas fue financiada por DARPA. Estos primeros grupos de trabajo se centraron en la parte de distribución de código móvil y definieron mecanismos como ANEP o la arquitectura de los nodos activos que tuvieron mucha influencia sobre las investigaciones posteriores. Los trabajos más significativos fueron:

- ANTS [254, 253] fue uno de los primeros entornos de redes activas y quizá uno de los más difundidos. Desarrollado por el Instituto Tecnológico de Massachusetts (MIT), empleaba las características de carga dinámica y código móvil de Java. Sin embargo no utilizaba el enfoque integrado de otras propuestas, ya que los paquetes activos no transportaban el código directamente, sino que este se descargaba del nodo activo anterior, y así recursivamente hasta llegar al usuario final.
- La arquitectura SwitchWare [5] de la Universidad de Pennsylvania mezclaba las propuestas de código móvil enviado en los propios paquetes y código estático instalado en los nodos de red. El código activo empleaba PLAN [109], un lenguaje funcional muy limitado, así que para suplir sus carencias podían emplearse los denominados *Switchlets*, funcionalidades más complejas desarrolladas con lenguaje de programación de alto nivel, y que estaban pre-instaladas en los nodos activos.
- El Instituto Tecnológico de Georgia centró su investigación en la composición de servicios de red a partir de elementos básicos. Como resultado desarrolló el Entorno de Ejecución CANEs [156] y el NodeOS Bowman [157].

Además de DARPA, otras instituciones han seguido subvencionado investigaciones de este tipo, aunque el termino "red activa" ha dejado de ser un sinónimo de código móvil y se ha integrado en otras áreas de investigación como las redes superpuestas, los *proxies*, interfaces abiertas y los agentes móviles. Algunos proyectos recientes que investigan estos sistemas híbridos son:

- El sistema *Application Level Active Networking* (ALAN) [90], desarrollado dentro del proyecto europeo ANDROID, no aplica el concepto de *routers* activos, sino que el procesamiento se realiza en servidores *proxy* localizados en puntos estratégicos de la red. Dichos *proxies* son capaces de descargar

proxylets para realizar transformaciones sobre los flujos de datos de las aplicaciones que los utilizan.

- El grupo de trabajo *Open Pluggable Edge Services* (OPES) del IETF ha definido una arquitectura [19] para permitir que *proxies* dentro de la red analicen y transformen el flujo de datos de acuerdo a unas políticas configurables. Esas transformaciones pueden realizarse directamente en el *proxy*, o delegarlas [207] a servidores externos especializados. El filtrado de contenidos o la traducción de páginas web son buenos ejemplos [18] de servicios OPES.

- El principal objetivo del proyecto europeo FAIN [91] ha sido desarrollar una arquitectura de red abierta y programable, basada en la integración de tecnologías como las redes activas, los sistemas de objetos distribuidos y los agentes móviles. Los resultados de dicho proyecto [81] incluyen, además del desarrollo de un NodeOS con soporte para diversos Entornos de Ejecución, la definición de un nuevo modelo de gestión de red, y el estudio de diferentes modelos de negocio para las redes activas.

Tal como veremos en la siguiente sección, las propuestas de redes activas han cubierto muchos ámbitos y aplicaciones, por lo que realizar un estudio pormenorizado de cada uno de ellos queda fuera del ámbito de este trabajo. Sin embargo merece la pena mencionar algunos proyectos muy relacionados con esta tesis doctoral, y cuyo objetivo se ha centrado en sacar a las redes activas del ámbito de la investigación, y trabajar para su integración en *routers* de alto rendimiento. Los principales trabajos en este sentido han sido:

- La arquitectura PRONTO [111] ha sido desarrollada por los laboratorios de AT&T, para el desarrollo de nuevos servicios de red sobre nodos de alto rendimiento orientados a flujos, como conmutadores ATM o *routers* MPLS. Su principal aportación ha sido la definición de una interfaz eficiente en ancho de banda para suscribirse a los flujos que atraviesan el nodo de red y recibir sólo la parte necesaria de los datos.

- Nortel Networks ha definido el *Java Forwarding* API para acceder a las funcionalidades del plano de datos de sus *routers*. A partir de dicho API se ha desarrollado el Entorno de Ejecución ORE [126], que funciona sobre una de sus plataformas comerciales, la familia de *Accelar Gigabit Router-Switches*.

- ANN [66] en cambio es una arquitectura de *router* multi-procesador basada en *hardware* y diseñada específicamente para crear un nodo activo de altas prestaciones. El *router* está formado por una *switching fabric* ATM y múltiples tarjetas procesadoras denominadas *Active Network Processing Engines* (APNEs), que están compuestas por una CPU de propósito general, memoria DRAM y una FPGA, aunando de esta forma capacidad de proceso y flexibilidad.

2.3.2. Aplicaciones Activas

Los proyectos de redes activas han tenido una escasa aceptación más allá del entorno de investigación. Una de causas que se citan más a menudo es la ausencia de una aplicación realmente atractiva y novedosa que justificase el desarrollo de productos comerciales basados en este nuevo paradigma.

Ciertamente la investigación en redes activas ha estado mucho más enfocada hacia la plataforma (NodeOS y Entornos de Ejecución) que a las Aplicaciones Activas destinadas a usarla. Aún así se han desarrollado numerosas [196] Aplicaciones Activas: gestión de red [216, 156, 200], calidad de servicio [21], descarte selectivo de paquetes MPEG [36, 111], *multicast* fiable [34, 156], compresión web [90], *streaming* de audio HTTP/RTP [90], trans-codificación de flujos multimedia [185], construcción dinámica de *clusters* [95], despliegue de *relays* multimedia [246], gestión distribuida de QoS [249], etc.

La investigación en redes activas se considera actualmente agotada. Por ejemplo la principal conferencia sobre redes activas, IWAN [125], fue clausurada con su última edición en Octubre de 2004. No obstante, este autor considera que la investigación en las redes activas ha realizado una gran aportación al identificar explícitamente los beneficios e inconvenientes de realizar procesamiento dentro de la red. Esta opinión se sustenta en la creciente popularidad de servicios de red como los *firewalls*, NATs, balanceadores de carga L4/L7, *proxies* Web/SOCKS, etc. que a todos los efectos pueden considerarse nodos de red que realizan procesamiento "activo".

2.4. Definición de Interfaces Abiertas

A pesar de que la separación entre el plano de datos y el plano de control es un principio de diseño conocido y plenamente aceptado, en la práctica este

principio aún no ha sido aplicado hasta sus últimas consecuencias. Típicamente los equipos de red son sistemas integrados verticalmente, esto es, el fabricante proporciona tanto el *hardware* de comunicaciones como el *software* de control. Por tanto, aunque internamente los equipos de red tienen planos de datos y control diferenciados, esta separación lógica no permite que cada plano evolucione de forma totalmente independientemente, puesto que el fabricante es el único que puede innovar en cualquiera de los dos campos. Obviamente esta situación implica que los operadores de red dependen totalmente de los fabricantes de sus equipos a la hora de añadir nuevos servicios a su red.

De nuevo aparecen dificultades para el desarrollo y despliegue de nuevos protocolos y servicios de red, y que, tal como se explicó en la sección anterior, fue el origen del paradigma de redes activas. Sin embargo este problema también ha sido el punto de partida de otra línea de trabajo dentro del ámbito de las redes programables y que podríamos denominar como "interfaces abiertas". Esta línea de investigación no propone cambios tan revolucionarios como las redes activas, y quizá por eso pueda resultar mucho más fructífera.

Por resumirlo brevemente, su objetivo consiste en definir interfaces abiertas [266] para los sistemas de comunicaciones, que permitan la interoperabilidad entre fabricantes y de esta forma se simplifique la investigación, desarrollo e implantación de nuevos servicios de red.

No obstante, la frontera entre las redes activas y las redes programables con interfaces abiertas es extremadamente difusa puesto que, al fin y al cabo, los grupos de investigación de redes activas también se han dedicado a definir y adoptar interfaces abiertas dentro los nodos de red, esto es, entre el NodeOS y los diferentes Entornos de Ejecución, o entre estos y las Aplicaciones Activas. Sin embargo las redes programables con interfaces abiertas tienen dos diferencias muy importantes respecto a las redes activas:

1. La programabilidad de los nodos de red está mucho más limitada, puesto que el objetivo principal no es modificar el paradigma actual de procesamiento de los paquetes de datos, sino simplificar el desarrollo de nuevos servicios de red y protocolos de gestión o señalización.

2. La programabilidad de la red y el despliegue de nuevos servicios y protocolos es responsabilidad exclusiva del administrador de red. Por tanto, al contrario que en el caso de las redes activas, los mecanismos de distribución de código son secundarios.

La investigación en redes programables ha sido tan prolífica [37, 107] como la de las redes activas. Por tanto, en esta sección nos centraremos exclusivamente en las iniciativas de los organismos estandarización más destacados, tales como el *Instituite of Electrical and Electronics Engineers* (IEEE), el *Internet Engineering Task Force* (IETF) y el *Network Processing Forum* (NPF).

2.4.1. IEEE P1520

El objetivo del proyecto IEEE P1520 [29] ha sido la definición de una arquitectura abierta de red, mediante la estandarización de un conjunto de *Application Programming Interfaces for Networks*. Esos APIs orientados a objetos han sido definidos en IDL[15] para permitir el desarrollo de protocolos abiertos de control y señalización, así como otros servicios de red de valor añadido. Aunque el grupo de trabajo P1520 del IEEE está centrado en tecnologías como ATM o conmutadores de circuitos SS7, también se ha formado un sub-grupo de trabajo [151] para redes IP, del que tratará este apartado.

La figura 2.4 muestra el modelo de referencia definido por el IEEE P1520. Dicha arquitectura está formada por cinco capas y cuatro interfaces para acceder a los servicios de los niveles inferiores:

- Las entidades del *Value Added Services Level* (VASL) proporcionarían, a través de la interfaz V, servicios de valor añadido tanto a los proveedores de servicio como a los usuarios finales de la red, aunque se esperaba que esos servicios fueran más parecidos a los definidos en la comunidad de *Advanced Intelligent Network* (AIN) que aplicaciones de usuario finales.

- El nivel *Network-Generic Services Level* (NGSL) se encargaría de tareas como el encaminamiento o la señalización para el establecimiento/liberación de las conexiones. Al igual que VASL, este nivel contendría entidades que tuviesen una visión global de la red. La funcionalidad de esta capa sería accesible por el nivel VASL a través de la interfaz U.

- El *Virtual Network Device Level* (VNDL) proporcionaría una abstracción (*software*) programable de cada elemento de red para acceder a sus funcionalidades, recursos o estado actual. La interfaz L permitiría el desarrollo de mecanismos de gestión y de control multi-plataforma, para equipos de red de cualquier fabricante.

[15] Interface Definition Language, en inglés

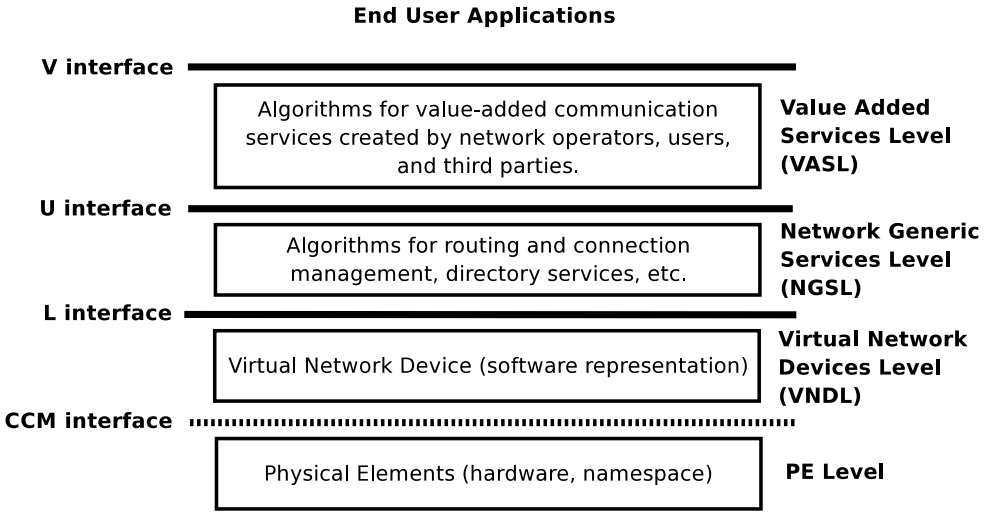


Figura 2.4: Modelo de referencia de IEEE P1520

- En la base de la torre de protocolos estarían los *Physical Elements* (PEs), esto es, el *hardware* de los nodos de red. La interfaz de *Connection Control and Management* (CCM), no sería realmente una interfaz de programación, sino algún tipo de protocolo, similar a GSMP o ForCES, para gestionar el elemento de red desde un agente de control remoto.

Este modelo de referencia no sólo intenta definir una abstracción de los nodos de red, sino también de la red en su conjunto. Sin embargo la mayoría de sus actividades se han limitado a las interfaces L y CCM. En particular, el principal objetivo del grupo de trabajo P1520.3 [152] ha sido la abstracción de un *router* IP mediante la definición de su interfaz L [69]. Dicha interfaz ofrecería al nivel NGSL superior diversas funcionalidades, como el control del encaminamiento [158] u ofrecer diferentes calidades de servicio mediante RSVP o Diffserv.

Inicialmente, la arquitectura de la interfaz L estaba compuesta por dos subcapas: bloques de recursos y bloques de servicios, pero posteriormente ha sido dividida en tres sub-niveles:

- La interfaz L+ ofrece una abstracción del *router* basada en un determinado servicio. Por ejemplo, la interfaz L+ de Diffserv [198] modela objetos como los *Per Hop Behaviors* (PHBs) en lugar de centrarse en abstracciones de bajo nivel.

- La interfaz L- [199] está justo debajo del L+ y proporciona una abstracción de los recursos del *router* (puertos, memoria, *data path*) sin centrarse

en ningún servicio en concreto. Cada interfaz L+ sería la encargada de componer los diferentes recursos y funcionalidades, de acuerdo al servicio que proporcionase.

- La capa inferior tiene el mayor grado de abstracción y tan sólo define conceptos básicos [30] como `Acción`, `Componente`, `Condición`, `Objetivo` y `Unidad de Procesamiento` (paquete). El resto de los bloques de los niveles superiores hereda de estos elementos raíz.

A pesar de que finalmente no han llegado a estandarizarse, algunas capas del modelo de referencia IEEE P1520, o la idea de utilizar protocolos de control sobre arquitecturas distribuidas de nodo de red, han sido fundamentales para trabajos posteriores sobre redes programables.

2.4.2. Network Processing Forum (NPF)

El *Network Processing Forum* (NPF) [169] es un consorcio de compañías formado para impulsar la adopción de los Procesadores de Red. Para ello ha definido un conjunto de especificaciones *Hardware*, *Software* y de pruebas de rendimiento[16] para promover la interoperabilidad entre los diferentes fabricantes, así como para simplificar el trabajo de los desarrolladores de *software* para procesadores de red.

Las especificaciones *software* del NPF consisten en una serie de APIs para acceder a las funcionalidades de los sistemas implementados con procesadores de red. Dichas APIs separan los planos de datos y de control de los nodos de red, de modo que sea posible desarrollar aplicaciones de control compatibles con los sistemas de diferentes fabricantes.

El modelo *software* [170, 171] definido por el NPF tiene dos niveles de funcionalidad, tal como se muestra en la figura 2.5. La primera capa se denomina *System Abstraction Layer* y contiene las APIs de servicio del NPF. Esta capa proporciona una abstracción del sistema, de forma que los Servicios superiores no son conscientes de la distribución física del nodo de red, por ejemplo si está formado por varias tarjetas procesadoras. La segunda capa se denomina *Element Abstraction Layer* y define un modelo del plano de datos independiente del fabricante. Tal como se puede ver en la figura 2.6, las APIs de este nivel

[16] Benchmark, en inglés

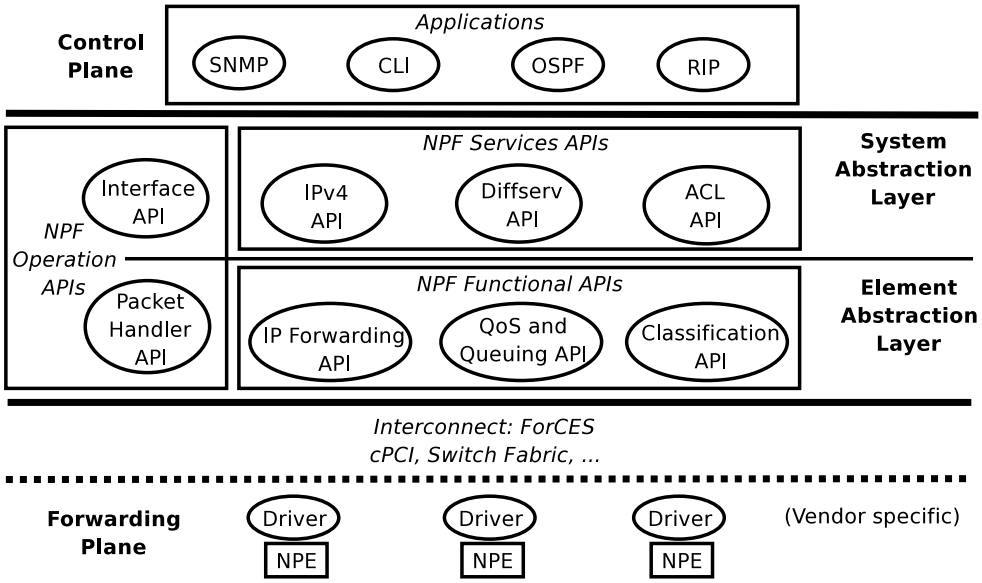


Figura 2.5: Modelo *Software* del *Network Processing Forum* (NPF)

sí son conscientes de la existencia de múltiples procesadores de red y de sus capacidades, en lugar de la visión de "caja negra" que tiene la capa superior.

Este conjunto de APIs del NPF, y las aplicaciones que las emplean, pertenecen a los planos de gestión y de control del nodo de red. El plano de datos puede estar conectado con el plano de control mediante cualquier tipo de red de interconexión lógica o física. En ese caso, cuando los planos de datos y de control residen en elementos separados, la comunicación requiere algún tipo de protocolo propietario, o estándar como GSMP o ForCES.

Todas las APIs [174] del NPF están definidas en ANSI C e IDL y son asíncro-

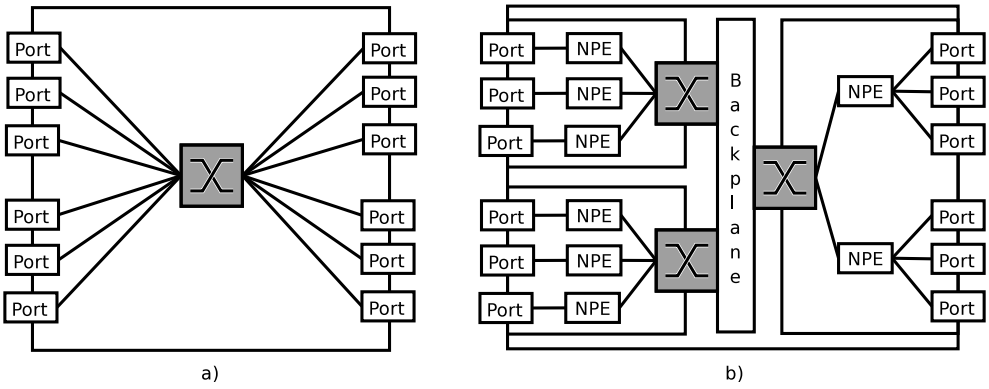


Figura 2.6: Vista de las capas del NPF: a) *System Abstraction Layer*, b) *Element Abstraction Layer*.

nas para permitir un alto grado de eficiencia y de paralelismo. Para ello se han definido un gran número de funciones de *callback* y de manejadores de eventos.

APIs de Servicios

Este conjunto de interfaces está formado por API específicas para cada servicio, que proporcionan un modelo abstracto de alto nivel del nodo de red. Por ejemplo permiten la creación y mantenimiento de la tabla de ruta y de las reglas que controlan el encaminamiento de los paquetes. Algunas de las APIs de este tipo definidas por el Network Processing Forum son: el IPv4 API [176], el IPv6 API [177] y el MPLS API [172].

APIs de Operaciones

Estas APIs ofrecen funciones comunes a las dos capas de abstracción definidas por el NPF. Por ejemplo la configuración y la gestión de las interfaces de red, o permitir a las aplicaciones enviar y recibir paquetes. Las especificaciones de APIs de Operaciones publicadas hasta la fecha son: el *Packet Handler API* [173] y el *Interface Management API* [175].

APIs de Funcionalidades

El modelo funcional del NPF representa al plano de datos como una secuencia ordenada de bloques lógicos con una determinada funcionalidad. El API de funcionalidades proporciona mecanismos para ajustar el comportamiento de esas "etapas", desde simples descartadores o contadores de paquetes, a elementos más complejos como aquellos que implementan políticas de gestión de colas.

2.4.3. *General Switch Management Protocol* (GSMP)

El protocolo GSMP ha sido definido por el IETF para permitir que los conmutadores de etiquetas puedan ser gestionados por un controlador externo. GSMP puede aplicarse [75] a cualquier tipo de conmutador de celdas o de paquetes, basado en etiquetas y orientado a conexión.

Tal como muestra la figura 2.7, en GSMP el modelo de conmutador es bastante sencillo: un conmutador tiene varios puertos. Cada puerto, a su vez, tiene

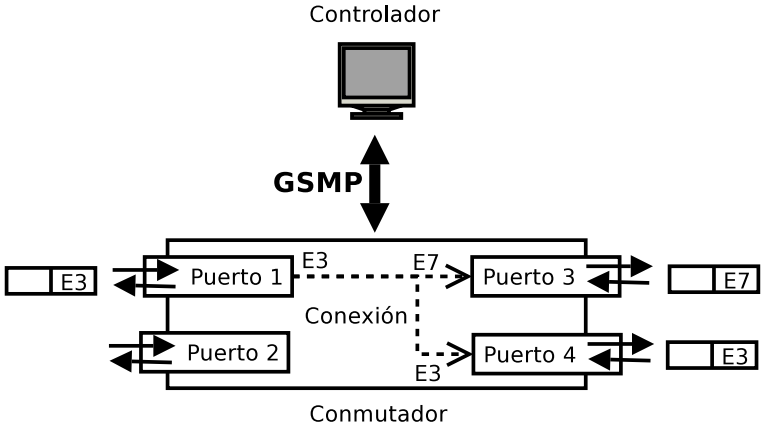


Figura 2.7: Arquitectura de un conmutador GSMP.

un puerto de entrada y otro de salida. Las conexiones que atraviesan los conmutadores se forman asociando la etiqueta de un puerto de entrada con una o más etiquetas de puertos de salida. Por tanto una conexión tiene una única tupla <puerto de entrada, etiqueta>, pero puede tener varias ramas conectadas a puertos de salida diferentes, cada una con su etiqueta correspondiente. De este modo, cuando llega un paquete por un puerto de entrada, su etiqueta identifica a la conexión a la que pertenece, y por tanto a los puertos de salida por lo que debe enviarse el paquete, reemplazando su etiqueta de entrada por la etiqueta de salida adecuada.

Las versiones iniciales de GSMP, la 1.1 [178] y la 2 [179], fueron desarrolladas por Ipsilon Networks y se centraban en conmutadores ATM. Más tarde, la versión 3 [74, 224] extendió GSMP para soportar cualquier tipo de conmutador de etiquetas, incluyendo ATM, Frame-Relay, MPLS, o incluso conmutadores con puertos de diferentes tecnologías, pero que emplean algún mecanismo de adaptación entre ellas. Aunque la mayoría de las características se han mantenido, cada versión del protocolo ha definido un modelo de calidad de servicio diferente. La versión 1.1 tan sólo permitía celdas con diferentes prioridades. En la versión 2 se definió un modelo de calidad de servicio bastante complejo y que fue reemplazado en la siguiente versión. En GSMPv3 se adoptó el modelo de prioridades inicial, además de los servicios de tráfico definidos por cada tecnología soportada (e.g. CBR y NRT-VBR de ATM), o cualquier otro perfil de calidad de servicio definido fuera de GSMP.

Además, para permitir que varios proveedores puedan acceder a la misma infraestructura de red, GSMP permite que un conmutador físico pueda dividirse

estáticamente en múltiples conmutadores virtuales, cada uno con sus propios recursos y funcionando como entornos independientes.

Después de publicar la tercera versión, el grupo de trabajo de GSMP se fijó dos objetivos: permitir el particionamiento dinámico [11] de un conmutador físico, y extender GSMP para soportar conmutadores ópticos [145] y TDM. Para ello dividió la especificación de GSMP en un documento base [72] con los mensajes y procedimientos comunes, y otros documentos [234, 42] donde se incluía la información específica de cada tecnología, como por ejemplo el formato de las etiquetas. Estos cambios no alcanzaron el estatus de RFC ya que el grupo de trabajo de GSMP fue disuelto en Enero del 2004, aunque parte de sus miembros han continuado su trabajo en otros grupos del IETF como ForCES, del que se tratará en el siguiente apartado.

GSMPv3 es un protocolo maestro-esclavo, basado en transacciones, que se emplea entre un conmutador y su controlador. Los mensajes GSMP pueden transportarse sobre TCP [257], o directamente en tramas AAL5 sobre ATM [178] o en tramas Ethernet [179].

Hay seis familias de mensajes de control en GSMPv3, y cada una agrupa diferentes tipos de operaciones:

Gestión de conexiones: Añadir rama, Borrar árbol, Borrar todas las entradas, Borrar todas las salidas, Borrar ramas, Mover rama de entrada, y Mover rama de salida

Gestión de puertos: Gestión de puerto y Rango de etiquetas

Estado y Estadísticas: Actividad de conexión, Estadísticas de puerto, Estadísticas de conexión, Estadísticas de clases de calidad de servicio, e Informe de estado de conexión

Configuración : Configuración del conmutador, Configuración de puerto, Configuración de todos los puertos, y Configuración de servicio

Reserva : Petición de reserva, Borrar reserva, y Borrar todas las reservas

Adicionalmente, GSMP permite que los conmutadores notifiquen a los controladores eventos importantes como la conexión/desconexión de un puerto, tanto físicamente (`new port`, `dead port`) como lógicamente (`port up`, `port down`), o la llegada de una trama con una etiqueta desconocida.

Sin embargo, antes de que el controlador pueda enviar cualquier mensaje de configuración, debe establecerse una sesión con el conmutador, mediante el protocolo de adyacencia de GSMP. Este protocolo es muy importante y se emplea para asegurarse de que el controlador y el conmutador estén siempre sincronizados. Tanto es así que cada puerto tiene un contador de sesión, si los mensajes del controlador no tienen el número de secuencia correcto, el conmutador debe cerrar la sesión y reiniciar el procedimiento de adyacencia.

2.4.4. *Forwarding and Control Element Separation* (ForCES)

En 2001 el IETF creó el grupo de trabajo ForCES [85] para definir un protocolo estándar para la comunicación entre el plano de datos y el plano de control de los *routers* IP basados en arquitecturas distribuidas [61]. En particular, el protocolo ForCES permite a la CPU principal del *router*, o Elemento de Control[17] (CE), configurar las tarjetas procesadoras que lo forman, o Elementos de Reenvío[18] (FE).

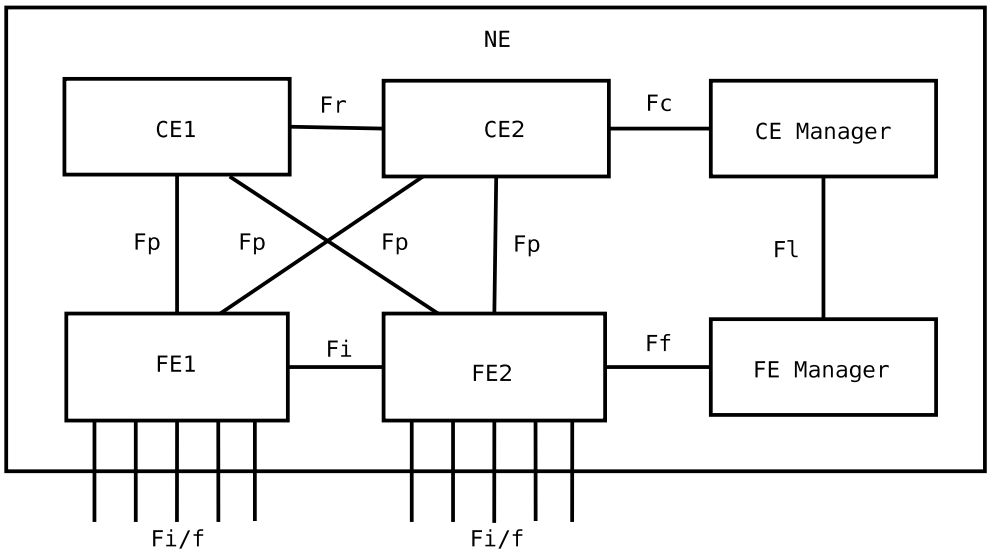


Figura 2.8: La arquitectura ForCES

La figura 2.8 muestra la arquitectura [263] de un Elemento de Red[19] (NE) ForCES. Además de los Elementos de Control y de Reenvío, un NE tiene otras

[17] Control Element, en inglés
[18] Forwarding Element, en inglés
[19] Network Element, en inglés

dos entidades: el Gestor de CE[20] y el Gestor de FE[21]. Sin embargo estos elementos tan sólo se emplean en la fase de Pre-Asociación [166], cuando los CEs y descubren los FEs disponibles y sus funcionalidades, mientras que actualmente el grupo de trabajo está centrado en la fase de Post-Asociación, cuando los CEs y FEs ya se conocen entre sí.

Modelo de un *Forwarding Element* ForCES

Los requisitos de ForCES [143] no cubren solamente la funcionalidad básica de encaminamiento IP, sino además capacidades avanzadas tales como la calidad de servicio, el filtrado de paquetes, la manipulación de cabeceras (e.g. NAT), o incluso otras que puedan definirse en el futuro. Por tanto, a diferencia de GSMP, en ForCES no es posible definir todas las operaciones posibles del plano de datos del *router*. En su lugar, ForCES está definiendo un lenguaje XML que permita construir un modelo genérico [265] de Elemento de Reenvío, en el que se divide el proceso de encaminamiento en varias etapas.

Según este modelo, un FE está formado por *Logical Functional Blocks* (LFB) interconectados entre sí. Cada LFB modela una parte del proceso de reenvío de un datagrama IP. Por ejemplo la librería de LFBs [264] definida hasta la fecha incluye: *Port*, *Dropper*, *Redirector (de-MUX)*, *Scheduler*, *Queue*, *Counter*, *Meter*, *Policer*, *Classifier* y *Packet Header Rewriter*.

Los LFBs pueden modificar el contenido del paquete que procesa (e.g. cambiar la dirección IP origen en un LFB de NAT, o des-encapsular un datagrama en un LFB de Túnel IP-IP), o cambiar/añadir algún campo de la meta-información [68] que tiene asociada dicho paquete (e.g. indicar cuál es el puerto de salida de la trama). Además, los LFBs pueden mantener estadísticas sobre los paquetes procesados (e.g. se ha definido un LFB Contador). Por tanto el protocolo ForCES puede emplearse tanto para acceder al estado de los LFB, así como para configurar su comportamiento, de forma que cada LFB tendrá atributos de estado y atributos de configuración, muy en la línea de las MIB de SNMP.

Los LFBs pueden tener varias entradas y salidas, que además pueden agruparse para formar grupos de entrada/salida. En general las salidas de un LFB se agrupan cuando todas ellas transmiten el mismo tipo de paquetes, pero deben ser procesados por LFBs diferentes (e.g. un datagrama IPv4 puede ser enviado

[20] CE Manager, en inglés
[21] FE Manager, en inglés

a otro FE o reenviado por otra interfaz del FE que lo recibió), mientras que las excepciones en el procesamiento de los paquetes tienen salidas propias. Por tanto, los LFBs con varias salidas deben elegir una de ellas basándose en el contenido del paquete o en su meta-información asociada.

Las entradas/salidas de un LFB siempre están unidas a otros LFBs, de forma que un LFB recibe un paquete por alguna de sus entradas, lo procesa y lo reenvía por alguna de sus salidas. Las únicas excepciones a este comportamiento de flujo son: los LFB *Dropper* que no tienen salidas sino que sirven para descartar paquetes, y los LFB de *Port* que representan los puertos físicos del FE, y por tanto cuando reciben un paquete, lo envían por la línea, mientras que sus paquetes de salida, acaban de ser recibidos por el FE. De hecho, incluso la comunicación dentro del *router* (e.g. CE-FE o FE-FE) se modela con Puertos LFBs, lo que permite independizar ForCES y la comunicación inter-FE de la tecnología de interconexión empleada.

Este esquema tan simple de LFBs interconectados entre sí permite representar fielmente el procesamiento de los paquetes dentro de un FE, aunque la descomposición del plano de datos del *router* en LFBs básicos puede degenerar rápidamente en diagramas de flujo muy complejos, con un gran número de LFBs.

Ejemplos de Modelos ForCES

En la figura 2.9 se muestra un sencillo ejemplo de modelo ForCES. Está formado por un Clasificador de Flujos IP que añade un identificador de flujo a la meta-información de los paquetes que lo atraviesan. Posteriormente un Demultiplexor decide que acción debe realizarse sobre dicho paquete, y lo envía al LFB apropiado. Por ejemplo a un Contador antes de descartar el paquete, o un LFB que permite reescribir la cabecera IP de los paquetes. Con este modelo tan simple sería posible implementar un *firewall* (los paquetes no autorizados se descartarían) y un NAT (sobre los paquetes con direcciones privadas).

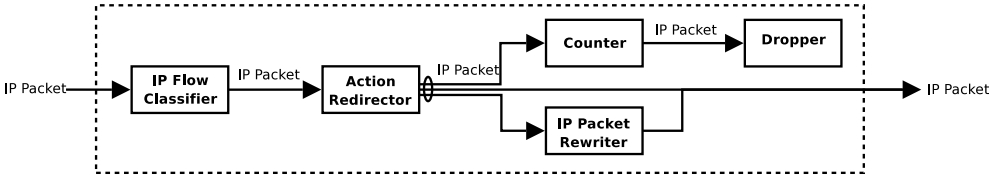


Figura 2.9: Sub-modelo ForCES de un FE con *Firewall*/NAT

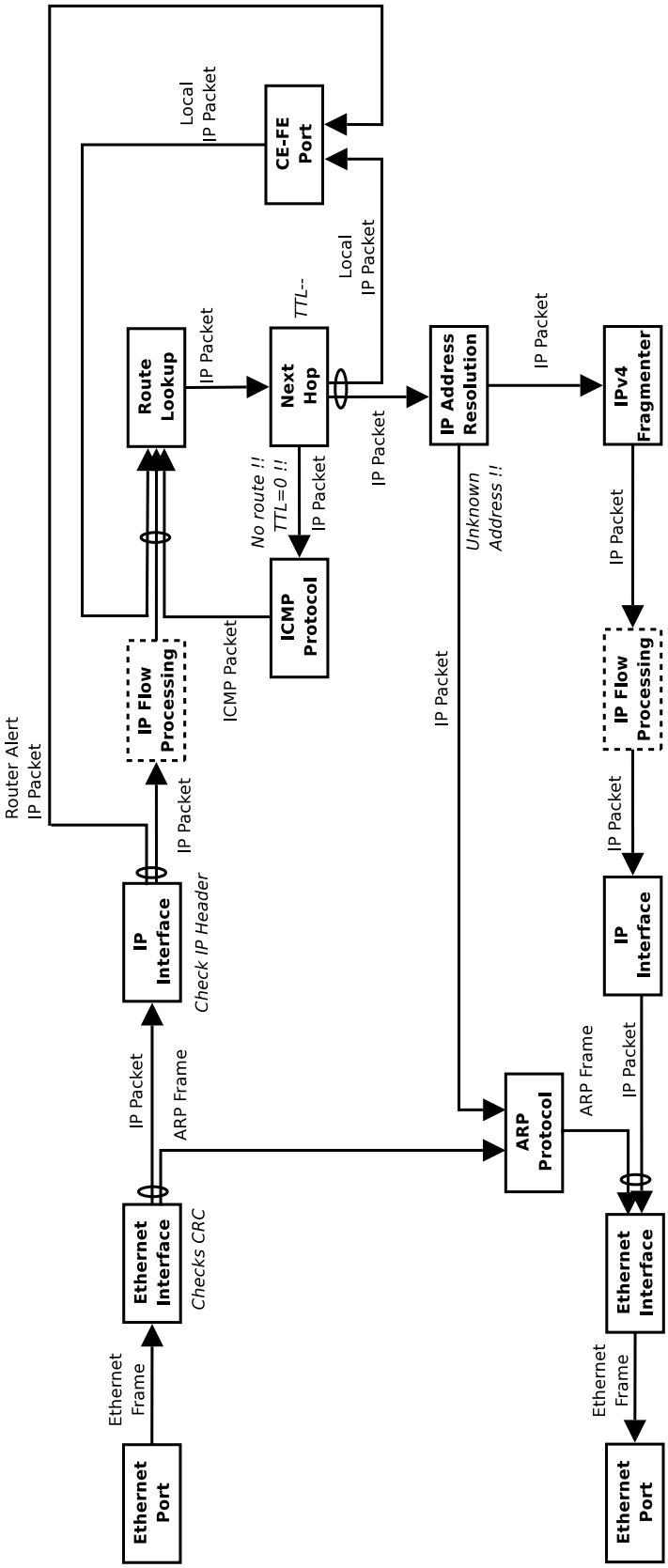


Figura 2.10: Modelo ForCES del FE de un *router* de memoria compartida

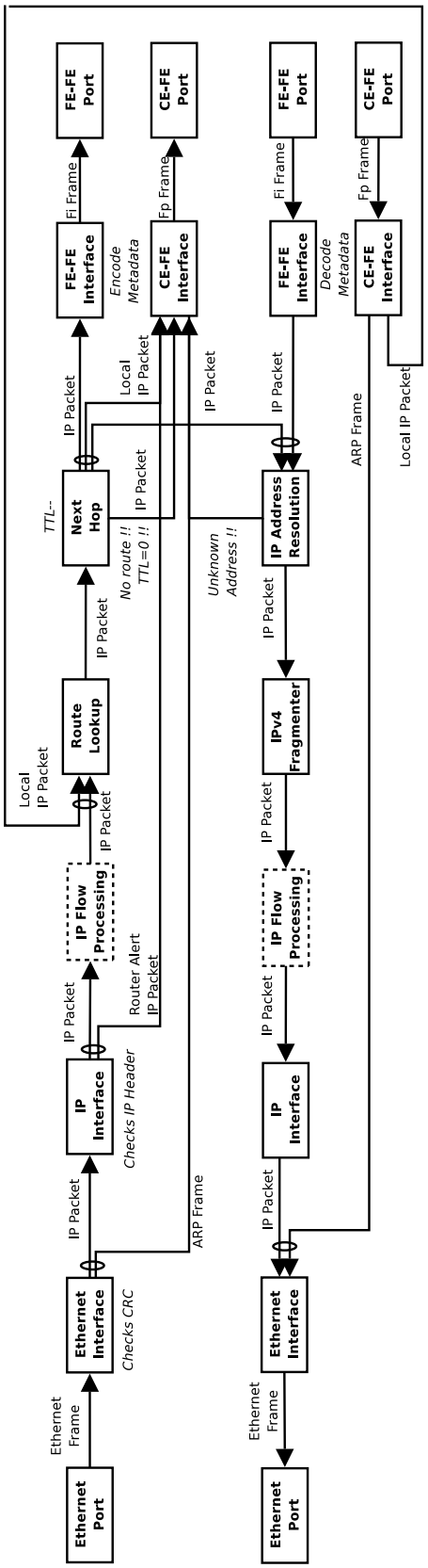


Figura 2.11: Modelo ForCES del FE de una tarjeta de línea

En la figura 2.10 se muestra un modelo simplificado del FE de un *router software* de memoria compartida. Dado que todo el procesamiento se realiza en una única CPU no tiene sentido dividir *router* en múltiples FEs, y por tanto el NE está formado por un único FE y un CE que lo controla. En este caso la separación CE/FE es puramente lógica, entre el espacio de usuario y el *kernel*. Las rutinas de reenvío y de procesamiento de paquetes forman parte del sistema operativo, mientras que los demonios de encaminamiento, de gestión o los comandos de configuración son procesos de usuario.

Al emplear una CPU de propósito general, la mayoría de los mecanismos de control, como la implementación del protocolo ARP o la generación de mensajes ICMP, se realizan dentro del propio FE. Por tanto el CE tan sólo procesa los paquetes IP locales, por ejemplo los de los protocolos de encaminamiento (e.g. RIP), y ciertos mensajes de señalización, como los datagramas RSVP que emplean la opción IP de *Router Alert*.

Por el contrario, la figura 2.11 muestra el modelo ForCES de una tarjeta de línea de un *router* multi-procesador. En este tipo de sistemas el rendimiento es crítico así que es bastante habitual implementar parte del reenvío de datagramas IP en ASICs, mientras que el resto de tareas de control o gestión se delegan a la CPU central. En este caso se tiene una arquitectura ForCES completa: cada FE es una tarjeta de línea, mientras que el CE es el procesador central que las gestiona. Por tanto, además de los puertos de comunicación con el exterior, es necesario algún tipo de red de interconexión entre FEs (interfaz `Fi`) y otra para la comunicación CE-FE (interfaz `Fp`)[22].

El protocolo ForCES CE-FE

Aunque en realidad podrían existir múltiples protocolos dentro de la arquitectura ForCES (uno por cada interfaz definida), el protocolo ForCES que está siendo diseñando tan sólo implementa el punto de referencia `Fp` entre un CE y un FE. Cualquier otra interacción dentro del NE, como por ejemplo la interconexión FE-FE se encuentra fuera de la lista de objetivos actual del grupo de trabajo.

Por tanto el denominado "protocolo ForCES" será un protocolo de control

[22]En principio sería posible emplear la misma tecnología para la comunicación CE-FE y FE-FE. Sin embargo en general las tarjetas de línea se interconectan mediante una matriz de conmutación de alta velocidad, mientras que para la interfaz de gestión CE-FE suelen emplearse tecnologías más asequibles como Ethernet.

que permitirá a los CE modificar los atributos de cada LFB de un FE. También será posible cambiar la topología [250] de los LFBs, esto es, como se interconectan entre sí, para permitir el provisionamiento de nuevos servicios con caminos de reenvío diferentes entre sí. Sin embargo aún no existe un consenso completo sobre el tema dado que es posible re-combinar LFBs lógicos implementados en el microcódigo de un Procesador de Red, pero es mucho más difícil, o incluso imposible, si los LFBs han sido implementados en ASICs.

Aunque inicialmente surgieron varias propuestas de "protocolo ForCES" [16, 209, 251], tras ser evaluadas [197], el grupo de trabajo ha decidido integrarlas en un único protocolo que sea capaz de cumplir todos los requisitos originales [143]. Por ejemplo, uno de ellos indica que el protocolo ForCES debe ser independiente de la tecnología de interconexión subyacente, ya que os elementos de Control y de Reenvío pueden estar interconectados por diferentes tecnologías como por ejemplo un bus de *backplane*, una matriz de conmutación ATM, o incluso Ethernet si residen en chasis diferentes.

A raíz de este requisito, el protocolo ForCES se subdivide en dos capas: *Protocol Layer* (PL) y *Transport Mapping Layer* (TML). La capa PL define los mensajes, la máquina de estados, y en general todos los aspectos del protocolo ForCES que sean independientes de la tecnología de comunicación subyacente. Por el contrario, la capa TML se ocupa del transporte de los mensajes de la capa superior a través de la red de interconexión interna, por tanto para cada tecnología se definirá una capa TML adecuada. Por ejemplo, en [17, 144] se proponen protocolos de TML para transportar los mensajes ForCES PL sobre redes IP.

Los mensajes del protocolo ForCES PL [73] tienen una cabecera común, que identifica los extremos de la comunicación (CE, FE, *multicast*) mediante identificadores de 32 bits, y una o más operaciones, agrupadas según el LFB sobre el que se aplican (cada LFB se identifica por su identificador de clase y un identificador de instancia).

El protocolo PL está basado en transacciones y, una vez que se establece la asociación ForCES entre el CE y cada FE, permite al CEs, no sólo configurar y obtener el estado de los atributos definidos por cada LFB, sino también suscribirse y recibir eventos asíncronos de los FEs, lo que evita la necesidad de realizar sondeos periódicos.

2.5. *Clustering* en redes programables

A grandes rasgos, la mayoría de las propuestas de redes programables se ha centrado en la división de los sistemas de comunicaciones en diferentes capas independientes entre sí, pero capaces de interaccionar entre ellas empleando interfaces de programación abiertas.

Un paso más allá en este proceso consiste en separar estos elementos no sólo de forma lógica, sino también físicamente, esto es, que cada entidad resida en un sistema diferente. Con este nuevo enfoque, el nodo de red estaría formado por múltiples sistemas independientes, interconectados mediante una red de comunicaciones interior al nodo, y que cooperarían empleando algún tipo de protocolo de comunicaciones. Este es el escenario de aplicación de GSMP, donde el plano de datos y de control residen en sistemas diferentes, y especialmente el de ForCES, donde incluso el plano de datos puede estar distribuido en varios Elementos de Reenvío.

Sin embargo estos protocolos no exploran una capacidad muy interesante: esta arquitectura distribuida es ideal para aplicar técnicas de *clustering* a los servicios de red computacionalmente intensos, y de esta forma conseguir una alta escalabilidad, tolerancia a fallos y reducción de costes.

Dado que esta característica será uno de los aspectos fundamentales de esta tesis doctoral, es interesante estudiar los trabajos relacionados más importantes:

- La arquitectura ABLE [200] ha sido diseñada para optimizar la gestión de red, empleando el paradigma de redes activas. Su principal característica consiste en delegar el procesamiento activo a un *active engine* adjunto al *router* IP, y que accede al estado del mismo empleando SNMP. Por su parte, el *router* es capaz de desviar paquetes al *active engine* basándose en los puertos UDP del paquete ANEP.
- En [94] se explora la idea de utilizar un *cluster* basado en *Distributed Shared Memory* (DSM), para incrementar la capacidad de proceso de los nodos de red activos. En dicha propuesta los paquetes activos se entregan a los diferentes miembros del *cluster* empleando *round-robin* para balancear la carga. Para las aplicaciones que necesitan estado por sesión, su autor sugiere emplear la memoria compartida del *cluster*. Sin embargo, este mecanismo puede tener un rendimiento muy limitado, y lamentablemente no se han publicado resultados prácticos para aclarar este aspecto.

- El proyecto Journey [185] estudia la viabilidad de la trans-codificación de flujos multimedia dentro de la red. Dadas las enormes necesidades de computación de la trans-codificación, se proponen varios mecanismos para distribuir la carga. Por ejemplo, no garantiza el procesamiento en la red sino que cada nodo decide independientemente si debe procesar un conjunto de fotogramas o no. Además ha desarrollado la arquitectura CLARA [252] en la que un *router* tiene un *cluster* de computación asociado para realizar el tratamiento de los flujos multimedia. Los paquetes que requieren procesamiento emplean la opción IP *Router Alert* y se distribuyen entre los nodos del *cluster*, teniendo en cuenta, no sólo una política de balanceo de carga [97], sino también las asignaciones anteriores y los recursos utilizados por el flujo. El prototipo de *cluster* desarrollado está formado por PCs interconectados mediante Myrinet, y la gestión del *cluster* utiliza un mini-*Object Request Broker* (ORB) para acceder a interfaces definidas en IDL.

- La arquitectura NAC (Nodo Activo en *Cluster*) [79] construye un nodo activo de alto rendimiento empleando múltiples sistemas: un nodo de filtrado de paquetes activos y varios nodos de procesamiento. Para ello emplea un NodeOS denominado NAC-OS que gestiona todos los recursos del *cluster* de manera integrada. De esta forma el sistema se comporta externamente como un único *router* activo, aunque su topología interna no es totalmente transparente para los servicios de red. De este modo es posible optimizar el paralelismo de las aplicaciones activas, y además se evita sobrecargar la red de interconexión.

En todas estas propuestas (exceptuando [94]) el procesamiento se reparte entre servidores totalmente desacoplados entre sí. Este principio de diseño se debe a las restricciones, tanto temporales como de ancho de banda, del procesamiento de datagramas a altas velocidades. Supongamos que, como en [94], la arquitectura permite acceder al estado almacenado en otros servidores. Entonces, en el peor caso, el procesamiento de cada datagrama implicaría el envío de al menos otros dos paquetes para el intercambio de información, además del retardo asociado que, incluso empleando redes de baja latencia como Myrinet [31], sería inaceptable dado el exiguo tiempo de procesamiento por paquete que requieren las interfaces de alta velocidad actuales.

Por esta razón en este documento no se estudiarán entornos de computación distribuida con memoria compartida, típicos de los sistemas de computación

científica, sino que se analizará la tecnología de granjas de servidores desacoplados, que han tenido un gran auge gracias al desarrollo de servicios web en Internet.

2.6. Reparto de carga en granjas de servidores

Uno de los principales problemas de los servicios de red es su escalabilidad, ya que incluso los *routers* frontera pueden estar compartidos por cientos o miles de usuarios, y es necesario que la capacidad de procesamiento de los nodos de red pueda incrementarse gradualmente, para soportar nuevos servicios, o debido al aumento en el número de usuarios a los que dar servicio.

Hay dos formas de incrementar la capacidad de un servicio: la primera, denominada escalabilidad vertical, consiste en tener un único servidor e ir añadiéndole más recursos (e.g. memoria, disco) según sea necesario. La escalabilidad horizontal, por el contrario, trata de aumentar la capacidad global del servicio añadiendo nuevos servidores.

De las dos opciones, la escalabilidad vertical es la más costosa en términos económicos, puesto que en algún momento es necesario comprar un servidor más potente y deshacerse del antiguo. Además, un sólo servidor significa un punto único de fallo, por lo cual su fiabilidad se convierte en un aspecto crítico, y su coste suele incrementarse aún más.

Por estas razones, la escalabilidad horizontal está ganando terreno a los grandes servidores tradicionales. El bajo coste de esta solución se basa, no sólo en la posibilidad de seguir utilizando la infraestructura antigua, sino también de la mejor relación capacidad/coste [12] de las estaciones de trabajo frente a los servidores multi-procesador, derivada de la economía de escala.

Además, esta solución permite actualizar la infraestructura rápidamente, basta con añadir otro servidor al *cluster*, y en principio se evita el temido punto único de fallo: si un servidor se estropea (o se da de baja por mantenimiento), el resto puede mantener el servicio activo. Sin embargo, esta solución requiere que el servicio pueda ejecutarse simultáneamente en varios servidores, y por tanto la complejidad del sistema aumenta.

Resumiendo, la utilización de múltiples servidores replicados para ofrecer un servicio tiene dos objetivos:

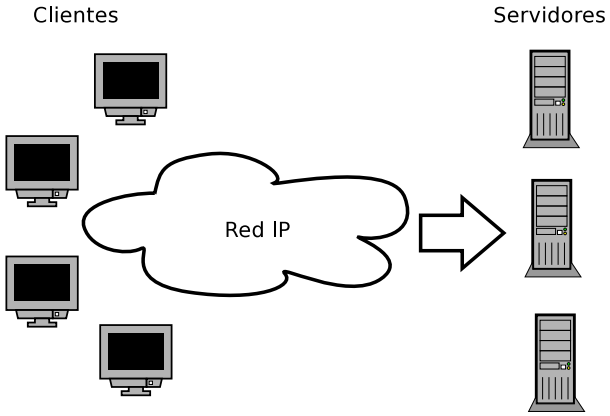


Figura 2.12: Arquitectura de una granja de servidores.

1. Distribuir las peticiones de los clientes entre los diferentes servidores para evitar que se sobrecarguen y obtener mejores tiempos de respuesta.

2. Ofrecer protección ante fallos mediante la redundancia de servidores.

2.6.1. Arquitectura de una granja de servidores

En este apartado se revisará someramente la arquitectura de las denominadas granjas de servidores (o *clusters*) y cómo funcionan para proporcionar un servicio escalable y tolerante a fallos.

En la figura 2.12 se muestra la arquitectura simplificada de una granja de servidores para un servicio Cliente-Servidor. En ella se tienen varios servidores que ofrecen un servicio y múltiples clientes que desean acceder a dicho servicio.

Supongamos que todos los servidores pueden ofrecer exactamente el mismo servicio de forma que, en principio, da igual a que servidor se conecte un cliente. En este caso, el funcionamiento es muy sencillo: antes de iniciar la sesión del servicio sería necesario elegir uno de los servidores que lo ofrecen. Una vez seleccionado, el cliente establecería una sesión con dicho servidor (típicamente mediante una conexión TCP) en la que se realizaría el intercambio de información del servicio propiamente dicho. Por tanto, existen dos fases diferenciadas:

1. Selección de uno de los servidores de la granja que implementan la funcionalidad requerida.

2. El acceso al servicio en sí, lo cual implica el intercambio de información

entre el cliente y el servidor seleccionado en la fase anterior.

Obviamente la segunda fase depende del servicio en sí y por tanto no es posible realizar un estudio de requisitos general. En cuanto a la primera fase, la selección del servidor debe cumplir dos propiedades:

- El servidor seleccionado no debe estar fuera de servicio.
- No debería seleccionarse un servidor sobrecargado si existen otros servidores alternativos "mejores".

La primera propiedad podría considerarse parte de la "tolerancia a fallos" del sistema, mientras que a la segunda, la denominaremos "reparto de carga", y será el tema fundamental de esta sección.

2.6.2. Reparto de carga

La planificación[23] de tareas entre múltiples procesadores es uno de los aspectos fundamentales de los sistemas distribuidos y, como tal, ha sido ampliamente estudiado [40]. Dentro de ese contexto, el término "reparto de carga" se refiere a conseguir una distribución equitativa de las tareas entre los diferentes procesadores, de forma que no haya procesadores sobrecargados mientras que otros se encuentran ociosos.

En este punto habría que diferenciar entre dos términos que se encuentran habitualmente en la literatura [223]: *load sharing* y *load balancing*. Aunque en muchas ocasiones se utilizan indistintamente, formalmente indican objetivos diferentes. Mientras que el primer término se refiere a evitar la sobrecarga de algún procesador en periodos de alta carga, el término *load balancing* va un paso más allá e intenta que todos los procesadores tengan exactamente la misma carga, y por tanto requiere mecanismos de planificación y migración de tareas mucho más sofisticados. En este trabajo nos limitaremos al concepto de *load sharing* y para ello se utilizará el termino "reparto de carga".

En las definiciones anteriores se ha empleado la terminología "clásica" de sistemas distribuidos, referida a "procesadores" y a "tareas". Estos términos son consistentes con el objetivo de los primero sistemas distribuidos, que se empleaban para construir un sistema de computación de altas prestaciones. Sin

[23] Scheduling, en inglés.

embargo, en el caso que nos ocupa, es más apropiado hablar de "servidores" y "sesiones de clientes". Además los mecanismos de "migración de trabajos" no tendrán sentido en este contexto, dado que los servidores están desacoplados y no comparten estado entre ellos. Por tanto la selección del mejor servidor tan sólo ocurrirá una vez, cuando llegue una nueva sesión de cliente, y una sesión con estado asociado jamas pasará de un servidor a otro.

Formalmente, el término "reparto de carga" no es un proceso, sino que es una propiedad de los algoritmos de asignación de trabajos, que indica su capacidad para distribuir la carga del *cluster* equitativamente entre todos los servidores que lo forman. Para alcanzar dicho objetivo será necesario realizar un proceso continuo de asignación de las conexiones nuevas de los clientes a alguno de los diferentes servidores que ofrecen un mismo servicio, siguiendo unas políticas de reparto de carga determinadas.

En función de quién realiza este proceso de selección/asignación al mejor servidor, las técnicas de reparto de carga pueden dividirse en dos grandes grupos:

- **Balanceadores de carga**. La asignación de sesiones a servidores las realiza un elemento intermedio situado entre el cliente y los servidores.
- **Selección en cliente**. En estas técnicas el cliente decide qué servidor es el más adecuado y se conecta directamente a él.

En los siguientes apartados se explicarán las principales técnicas de reparto de carga dentro de cada uno de estos grupos.

2.6.3. DNS *Round-Robin*

La NCSA fue la primera organización en emplear múltiples servidores web y repartir la carga entre ellos [136]. La técnica utilizada se denomina DNS *Round-Robin* y consiste en asignar a todos los servidores web el mismo nombre DNS, de forma que al resolver dicho nombre se devuelven todas las direcciones IP de los servidores. Sin embargo, en cada respuesta el servidor DNS coloca las direcciones en un orden diferente (generalmente rotando la lista de direcciones).

Esta técnica es transparente para los clientes y se basa en que normalmente las URLs no contienen la dirección IP del servidor sino su nombre simbólico. En la figura 2.13 se muestra el funcionamiento de esta técnica mediante un ejemplo:

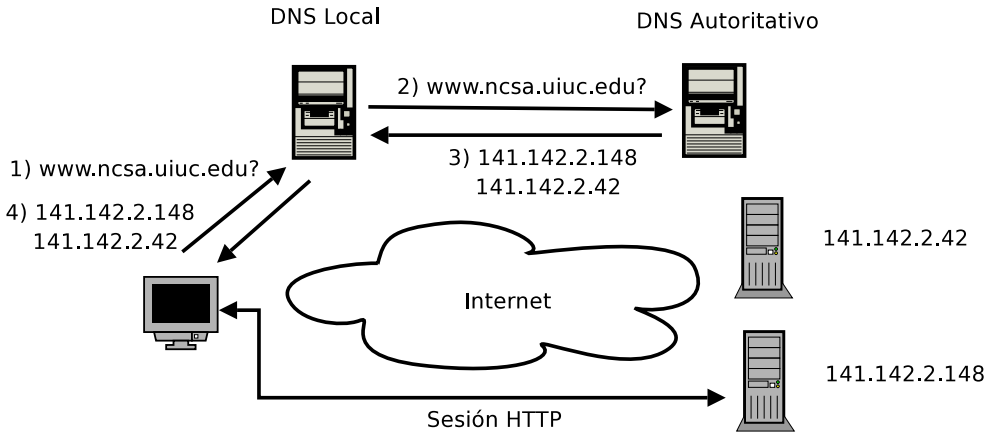


Figura 2.13: Ejemplo de DNS *Round-Robin*.

Supongamos que un usuario quiere acceder desde su navegador a la siguiente URL: `http://www.ncsa.uiuc.edu/index.html`

1. Antes de poder contactar con el servidor web, el navegador debe resolver el nombre `www.ncsa.uiuc.edu`. Para ello le pregunta a su servidor DNS local [159].

2. Si el servidor de DNS local no tiene la respuesta en su cache, realizará un proceso de consultas recursivas [160] hasta dar con el servidor DNS autoritativo del registro `www.ncsa.uiuc.edu`.

3. El servidor DNS responderá con todas las direcciones IP de los servidores web, en un determinado orden. El servidor DNS local guardará el registro en su cache y responderá al cliente.

4. Una vez que el cliente obtiene la respuesta, escoge una de las direcciones (normalmente la primera) y establece la sesión TCP/HTTP con dicho servidor web.

La siguiente consulta al servidor DNS autoritativo obtendrá la respuesta en otro orden así que, posiblemente, el siguiente cliente se conecte a un servidor web diferente. De esta forma tan simple es posible distribuir las diferentes peticiones web entre todos los servidores que forman el *cluster*.

Este mecanismo es extremadamente sencillo, transparente y compatible hacia atrás con cualquier servidor DNS [33]. Sin embargo tiene ciertas limitaciones:

Sin estado de los servidores. El servidor DNS no conoce la carga actual [214] ni establece diferencias entre la capacidad de los servidores. Por lo tanto un servidor potente puede recibir tantas conexiones como otro menos potente. Es más, puede devolver la dirección de un servidor sobrecargado o fuera de servicio.

La cache del sistema DNS. Los servidores locales de DNS almacenan en una cache los registros consultados recientemente. Por tanto, si un segundo cliente pregunta por `www.ncsa.uuic.edu`, el servidor DNS local responderá directamente con el mismo orden de las respuestas. Además hay aplicaciones que almacenan internamente los resultados del DNS sin tener en cuenta su TTL[24].

Debido a estas restricciones, se considera que el reparto de carga mediante el DNS no es demasiado preciso. Esto es, aunque en media el reparto de carga entre servidores es bastante equitativo, en los intervalos de alta carga que son los que realmente importan, unos servidores pueden estar sensiblemente más cargados que otros [161, 70].

La principal causa de este comportamiento se debe a la segunda limitación descrita, la cache del DNS. Una forma de limitar este efecto consiste en reducir el tiempo de vida (TTL) asociado a ese registro, o incluso fijarlo a cero para evitar que pueda ser almacenado en las caches.

Algunos autores [220, 38] han criticado esta opción indicando que el incremento del tráfico de DNS podría perjudicar la escalabilidad de todo el sistema DNS. Sin embargo, Jung *et al.* [130] han demostrado que el sistema de caches del DNS reacciona bastante bien a valores TTL bajos[25] y que la escalabilidad del sistema de DNS se mantiene mientras el TTL asociado a los registros `NS` sea razonablemente largo. De hecho, una de sus conclusiones afirma que, en el caso de deshabilitar la cache (TTL=0) de los registros tipo `A`, el tráfico de DNS de los servidores autoritativos se duplicaría pero el número de consultas a los servidores raíz se mantendría casi constante.

Otro aspecto a tener en cuenta es el retardo introducido por la resolución del nombre del servidor mediante DNS. Este punto ha sido bastante estudiado [256, 220, 56], especialmente en lo referido al tráfico web. Al invalidar las caches DNS siempre es necesario contactar con el servidor DNS autoritativo con la

[24] El navegador Mozilla almacena la dirección IP del servidor durante 15 minutos.
[25] Un valor del TTL de 15 minutos consigue tasas de aciertos del 80 %

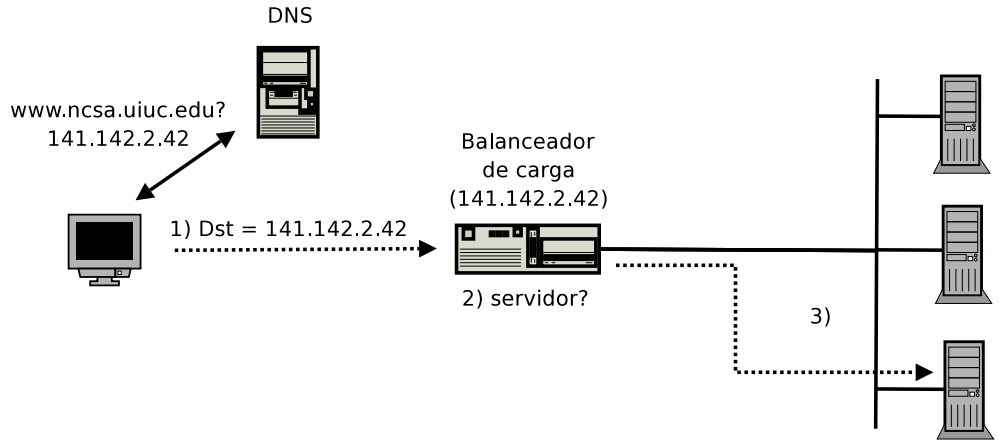


Figura 2.14: Arquitectura de un *cluster* con balanceador de carga.

latencia que eso supone. Aunque en estos artículos se indica que la mayoría de las consultas al DNS se resuelven en 100 ms [130], el incremento del número de servidores DNS a consultar y la retransmisión[26] de consultas perdidas pueden incrementar significativamente la latencia subjetiva del servicio.

Aún así, esta técnica es tan sencilla que se sigue empleando profusamente. Además puede combinarse con otras técnicas de reparto de carga para aumentar la equidad de la distribución entre servidores heterogéneos [70].

2.6.4. Balanceadores de carga

Los "balanceadores de carga"[27] surgieron con el objetivo de proporcionar un reparto de carga más equitativo que el que podían proporcionar otras técnicas más simples, como el DNS *Round-Robin* [136]. Estos dispositivos realizan un reparto de carga centralizado, observando todas las sesiones de los clientes y distribuyéndolas entre los servidores del *cluster*.

A diferencia de las técnicas como el DNS *Round-Robin* o las de selección en cliente, donde cada servidor tiene su propia dirección IP pública, en los *clusters* con balanceadores de carga los servidores no son visibles directamente para los clientes, sino que todo el *cluster* se comporta como un sólo sistema con una única dirección IP.

En la figura 2.14 se muestra la arquitectura típica de una granja de servidores

[26] Muchas implementaciones de DNS usan un temporizador de retransmisión de 5 segundos.
[27] Load balancer, en inglés.

con un balanceador de carga. Éste se coloca delante de los servidores para: 1) interceptar los paquetes de los clientes dirigidos a la dirección IP del *cluster*. Para cada paquete: 2) el balanceador decide a qué servidor pertenece ese paquete, y 3) se lo reenvía.

Cada paso de este proceso puede realizarse de diversas formas, lo que ha dado lugar a un gran número de propuestas, que van de simulaciones a productos comerciales. Cardellini *et al.* [38] han realizado un análisis muy completo de las técnicas de reparto de carga en *clusters* de servidores web, incluyendo una taxonomía de las mismas. Dicha taxonomía se basa en la capacidad de los balanceadores para analizar las diferentes cabeceras de los paquetes y elegir el mejor servidor en función de dicha información:

- Los denominados conmutadores de nivel 4 inspeccionan, para cada paquete, la cabecera del protocolo de nivel de transporte para decidir a que sesión de cliente pertenece dicho paquete.
- Los conmutadores de nivel 7 son capaces de analizar incluso hasta los datos de nivel de aplicación. Por ejemplo, en el caso de los servidores Web, la información analizada suele ser la URL, las *cookies* [202, 88] u otras cabeceras [181, 50] de la petición HTTP.

Conmutadores de Nivel 4

Los denominados "conmutadores de nivel 4"[28] inspeccionan, para cada paquete, la cabecera del protocolo de nivel de transporte para decidir a que sesión de cliente pertenece.

En muchos de estos dispositivos [43, 10, 269], el balanceador de carga se comporta como un NAT [228, 227]. Cada servidor posee una dirección privada [201] mientras que la IP pública del *cluster* está asociada al balanceador de carga. Por tanto, todas las conexiones de los clientes llegan inicialmente al balanceador.

En la figura 2.15 se muestra un ejemplo de esta arquitectura. Cada vez que se inicia una nueva conexión, el balanceador se la asigna a un servidor.

A partir de ese momento el balanceador debe cambiar la dirección destino de todos los paquetes del cliente (inicialmente la IP pública del *cluster*/balanceador) por la dirección IP privada del servidor para que, de esta

[28] Layer 4 Switch, en inglés.

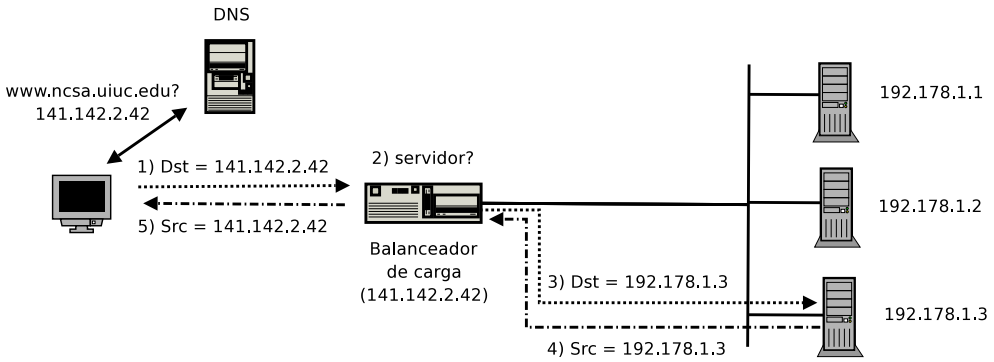


Figura 2.15: Arquitectura de un *cluster* con un conmutador L4/NAT

forma, pueda llegar al servidor asignado.

Dado que los clientes esperan recibir los paquetes de la dirección IP pública del *cluster*, el balanceador debe realizar también la acción opuesta. Cambiar en los paquetes enviados por los servidores la dirección fuente privada por la dirección del *cluster*.

Además de cambiar las direcciones IP, el balanceador de carga debe volver a calcular el *checksum* de la cabecera TCP dado que este también cubre en la pseudo-cabecera las direcciones antiguas. Dado que es necesario realizar estas operaciones para cada paquete, tanto de ida como de vuelta, la capacidad de procesamiento del balanceador de carga puede convertirse en un cuello de botella del *cluster*. Por esta razón algunos productos comerciales, como el LocalDirector de Cisco [43], implementan este proceso en *hardware*.

En el tráfico web normalmente las peticiones de los clientes son pequeñas mientras que las respuestas del servidor son mucho más grandes y por tanto más costosas de procesar por el balanceador. Por tanto, una forma de descargar al balanceador consiste en evitar que tenga que procesar las respuestas de los servidores. Por ejemplo en [70] se emplea la misma arquitectura descrita en la figura 2.15 con la salvedad de que los servidores responden directamente a los clientes sin pasar por el balanceador. Para ello, los servidores tienen un *kernel* modificado que responde con la dirección IP del *cluster*.

Otra alternativa que no requiere modificar el *kernel* de los servidores consiste en asignar, además de una dirección IP privada diferente para cada servidor, la dirección IP pública del *cluster* a todos los servidores [63, 116, 269]. De esta forma el balanceador no necesita cambiar la dirección IP destino, sino tan sólo

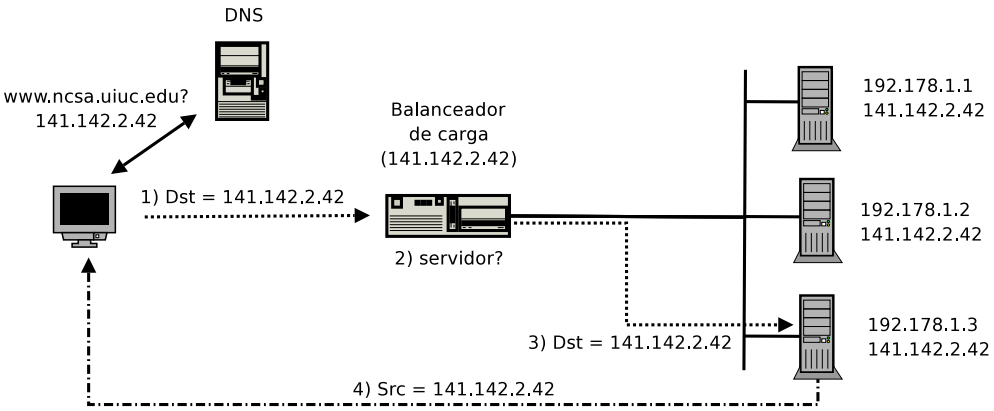


Figura 2.16: Los servidores del *cluster* comparten la misma dirección IP

entregar[29] el paquete al servidor asignado a nivel 2.

Como puede verse en la figura 2.16, este mecanismo requiere que el balanceador de carga esté en la misma subred que los servidores. Aunque este escenario es el más habitual, no cubre todos los casos. Así que, en el caso que el balanceador deba atravesar una red IP para alcanzar a los servidores, los paquetes pueden encapsularse [190] dentro de otros datagramas IP dirigidos a la dirección privada del servidor [269]. El servidor des-encapsula los paquetes de los clientes y responde directamente con su dirección pública.

Conmutadores de Nivel 7

En todas las técnicas analizadas hasta ahora se partía de una premisa básica: todos los servidores ofrecían exactamente el mismo servicio. Es decir, los servidores podían tener diferente capacidad de procesamiento o carga pero debían ofrecer exactamente la misma funcionalidad a los clientes. Los denominados "conmutadores de nivel 7"[30] intentan eliminar esta restricción, permitiendo el reparto de carga entre servidores con contenidos heterogéneos. Obviamente, para ello necesitan interpretar la información de nivel de aplicación, que en el caso de los servidores Web consiste en la URL, *cookies* [202, 88] u otras cabeceras [181, 50] de la petición HTTP.

Aunque estos dispositivos pueden parecer meras ampliaciones de los conmutadores/NAT de nivel 4, en realidad su comportamiento es bastante más

[29] Para evitar problemas con la dirección IP duplicada, es necesario desactivar el ARP para esa dirección.

[30] Layer 7 Switch, en inglés.

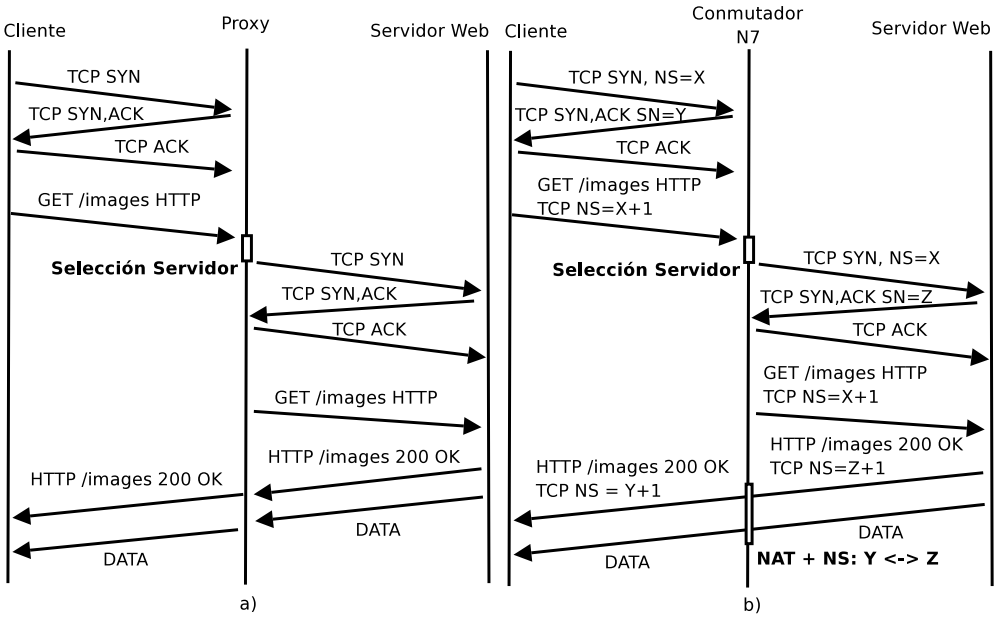


Figura 2.17: Comparativa entre: a) *Proxy TCP*, b) *TCP Splicing*

intrusivo. Mientras que en los dispositivos de nivel 4, la elección de servidor se producía en cuanto se recibía el primer paquete de la conexión (típicamente TCP), un conmutador de nivel 7 debe terminar el establecimiento de la conexión para poder interceptar la petición del cliente. Sólo en ese momento es posible analizarla y elegir el servidor adecuado para la misma.

La forma más sencilla de interceptar las peticiones de los clientes consiste en actuar simplemente como un *proxy*, esto es, recibir las peticiones de los clientes y reenviarlas a los servidores por otra conexión, e ir copiando la información de una sesión a otra.

Aunque es posible reducir la latencia de las respuestas manteniendo conexiones permanentes [261] entre el *proxy* y los servidores, parece claro que realizar la copia de información entre ambas conexiones en el *proxy* puede resultar costoso [55], incluso si este proceso se realiza dentro del *kernel* [270]. Una alternativa, denominada *TCP Splicing* [154] o *Delayed binding* [50], también consiste en establecer dos conexiones, pero una vez que se ha realizado la petición al servidor, estas se empalman entre sí. Para ello es necesario cambiar las direcciones de los paquetes y además modificar los números de secuencia de los paquetes que atraviesan el balanceador, tal como se muestra en la figura 2.17.

Al igual que en el apartado anterior, es posible reducir la sobrecarga en los balanceadores permitiendo que los servidores respondan directamente a los

clientes. De esta forma el balanceador tan sólo se encarga de procesar las peticiones de los clientes, que en el caso de la web normalmente tan sólo son paquetes de confirmación. Lamentablemente, en este caso no es posible realizar esta operación transparentemente y es necesario modificar los *kernel* de los servidores para que acepten sesiones TCP sin haber establecido previamente la conexión mediante el procedimiento de *Three-way-handshake* [195]. Los protocolos de *Connection Hand-off* [117], *TCP Hand-off* [187] o *TCP Connection Hop* [203] son ejemplos de esta técnica.

Un aspecto importante a tener en cuenta es que estas técnicas, aunque realizan las decisiones de balanceo a nivel de aplicación, redirigen las conexiones de los clientes a nivel de transporte. Esto puede ser un problema para protocolos persistentes, que emplean la misma conexión de transporte para realizar múltiples peticiones. Quizá el ejemplo paradigmático sea HTTP 1.1 [83], dónde es posible reutilizar la misma conexión TCP para múltiples peticiones-respuestas. Por tanto es posible que las peticiones de una misma conexión TCP tengan que redirigirse a diferentes servidores [14]. Las soluciones a este problema son variadas: mientras que en el caso de los *proxies* la solución es sencilla, en el caso de los balanceadores *hardware* es más complicado redirigir la sesión TCP a un servidor por cada petición [181, 182]. De hecho algunos productos comerciales [89] deshabilitan el uso de HTTP 1.1 y fuerzan a los clientes a emplear HTTP 1.0 (una petición por conexión).

Otras técnicas de reparto de carga

Aunque las granjas de servidores son intrínsecamente escalables, los balanceadores de carga pueden convertirse en un cuello de botella del mismo, incluso si tan sólo se encargan de uno de los sentidos de la comunicación.

Este razonamiento ha llevado a varios autores a diseñar sistemas en los que el proceso de reparto de carga está distribuido entre todos los servidores del *cluster*. Por ejemplo en [63, 221] los paquetes de los clientes se envían a todos los miembros del *cluster* en *broadcast*/*multicast* y cada uno decide si lo filtra o no. En [26] los servidores web reciben las peticiones de los clientes y pueden responder directamente o encaminar la conexión a otro servidor menos cargado.

Por supuesto, para que el sistema sea realmente distribuido es necesario repartir las peticiones de los clientes inicialmente. En el caso anterior se emplea el DNS *Round-Robin* aunque otros sistemas emplean técnicas diferentes. Por

ejemplo, en [15] se sigue empleando un conmutador N4 aunque los servidores son capaces de redirigir la conexión a otro mediante *TCP Hand-off* tras analizar la petición del cliente. Otra variación más reciente [108] también emplea esta técnica pero integrada con *TCP Splicing* para unir las conexiones en el balanceador.

Algunos autores [13, 165] también han propuesto estos sistemas totalmente distribuidos pero que emplean el mecanismo de redirección[31] de HTTP para distribuir las peticiones de los clientes equitativamente entre los miembros del *cluster*. Otro mecanismo similar [23, 150] consiste en distribuir y replicar las diferente páginas web entre múltiples servidores y reescribir la URL de los enlaces para que apunten a uno de los servidores que almacenan la página en ese momento.

Obviamente, estas técnicas sólo son aplicables a *clusters* de servidores web, pero aún así, muchas compañías se han decantado por emplear otros mecanismos específicos de HTTP para sus balanceadores de carga de nivel 7. Por ejemplo, ciertos productos [268, 88, 181, 50] pueden emplear una *cookie* para enviar las peticiones de un cliente siempre al mismo servidor web.

2.6.5. Algoritmos de reparto de carga

La elección del servidor al que redirigir las peticiones recibidas es un aspecto clave de los balanceadores de carga, del mismo modo que la planificación de trabajos es esencial en la investigación de sistemas de computación distribuidos [40].

A pesar del enorme paralelismo entre ambas áreas, las diferencias entre los algoritmos de reparto de carga utilizados en ambas es abismal. Mientras que en la segunda el planificador normalmente es la parte más compleja de todo el sistema, y suele emplear una gran cantidad de parámetros (CPU, memoria, E/S) e información histórica, muchos balanceadores de carga tan sólo soportan algoritmos de reparto sencillos y que no requieren estado, tales como selección aleatoria o *Weighted Round-Robin* [269].

En el caso de algoritmos con estado, esto es, aquellos en los que el balanceador es consciente de la carga actual de los servidores, aunque existen algunos protocolos para obtener el estado de los sistemas, como el *Cisco Dynamic Feed-*

[31] En lugar de responder directamente a un cliente, un servidor puede indicar que el recurso se ha movido a otra localización mediante los códigos HTTP 301 o 301.

back Protocol [45], normalmente los balanceadores de carga utilizan a lo sumo información local, como el número de conexiones que tiene cada servidor actualmente o el tiempo de respuesta de los mismos.

Estos algoritmos son muy populares ya que no requieren cambios en los servidores y, a pesar de su simplicidad, consiguen un reparto de carga razonable. Sin embargo algunos autores sostienen que la utilización de algoritmos más complejos, que por ejemplo tengan en cuenta la afinidad de las caches de los servidores con los contenidos podría mejorar el rendimiento del *cluster* [187].

2.6.6. Tolerancia a fallos

La segunda gran ventaja de los *clusters* de servidores es la tolerancia a fallos gracias a la replicación de funciones. Sin embargo para conseguir este objetivo, el balanceador de carga debe ser capaz de detectar que servidores han fallado y dejar de asignarles conexiones entrantes.

Un servidor puede "fallar" y dejar de ofrecer servicio por muchas razones y por ello los balanceadores de carga comerciales son capaces de comprobar el estado de los servidores a muchos niveles:

- Los chequeos pasivos son los más habituales y consisten en detectar errores en la redirección de las conexiones de los clientes: temporizadores que expiran, segmentos TCP con el *flag* RST activo o mensajes ICMP de Puerto no encontrado, son buenos indicadores de fallos en el servidor.
- Los chequeos activos requieren que el balanceador de carga compruebe periódicamente el estado de los diferentes niveles de la torre de protocolos del servidor [87]:
  - Nivel 2: ARP
  - Nivel 3: Ping
  - Nivel 4: Intentos de conexión TCP/UDP
  - Nivel 7: Pruebas para aplicaciones típicas (HTTP, FTP, etc.), *scripts* para comprobar la respuesta del servidor, o consultas SNMP a la MIB de la aplicación.

Además de la detección de errores, la tolerancia a fallos del *cluster* requiere que la funcionalidad del servidor caído esté replicada en alguno de los servidores

activos. Obviamente la complejidad de este requisito depende en gran medida de la aplicación, pero muchas veces implica que la información debe estar replicada [262] (por ejemplo utilizando `rsync`) o debe estar accesible para todo los servidores del *cluster*. Para ello se utilizan arquitecturas multi-capa que emplean bases de datos compartidas, o redes de almacenamiento (SAN o NAS), con servidores primarios y de respaldo para asegurar la integridad de los datos [204].

Un grave problema del uso de balanceadores de carga, es que el balanceador de carga se convierte en un punto único de fallo del *cluster* y por tanto es necesario el uso de varios balanceadores redundantes. En ese caso se suele repartir la carga entre ambos balanceadores empleando DNS *Round-Robin*, mientras que para permitir la tolerancia a fallos pueden utilizarse protocolos como el *Virtual Router Redundancy Protocol* (VRRP) [110] o el *Cisco Hot Standby Router Protocol* (HSRP) [48] para apropiarse de la dirección IP de un balanceador caído. Sin embargo, para conseguir que el fallo de un balanceador sea transparente a los clientes es necesario replicar las conexiones activas, para ello pueden utilizarse mecanismos *hardware* [46] o protocolos como el *Cisco Stateful Network Address Translation* (SNAT) [49], empleado para sincronizar tablas de NAT.

2.6.7. Reparto de Carga Global entre múltiples sedes

A lo largo de esta sección nos hemos referido a granjas de servidores situadas en una única sede. Sin embargo hay organizaciones cuyas necesidades de alta disponibilidad o de reparto de carga van más allá y emplean múltiples sedes repartidas por todo el mundo para ofrecer sus servicios [3].

A este tipo de arquitecturas se las conoce como "Redes de Distribución de Contenidos"[32], y requieren protocolos, algoritmos [57] y técnicas de reparto de carga global, diferentes a las estudiadas hasta ahora, tales como la utilización de servidores de DNS inteligentes [86, 44, 52], la inyección de rutas *anycast* [86, 52], o el uso de la redirección HTTP [44].

Dado que se dispone de múltiples sedes, lo ideal es seleccionar la sede más "cercana" al cliente. Sin embargo esta información es difícil de conseguir, por lo menos desde el punto de vista de un servidor, y por ello se han propuesto multitud técnicas como: tablas estáticas con prefijos de direcciones IP [52], la consulta dinámica de las tablas de encaminamiento [44], el uso de estadísticas

[32]Content Delivery Networks, en inglés.

sobre conexiones anteriores [86], calcular el *Round Trip Time* (RTT) entre el cliente y varias sedes [44, 52], o consultas inversas al DNS [52].

2.6.8. Reparto de carga en los clientes

Aunque hoy en día el uso de balanceadores de carga está muy extendido en las granjas de servidores, su utilización tiene numerosos problemas asociados:

- El balanceador de carga puede convertirse en un cuello de botella y en un punto único de fallo de todo el *cluster*, por lo que muchas veces es necesario emplear varios balanceadores redundantes, lo que encarece el coste total.
- Los balanceadores que se comportan como un NAT, tienen todos los problemas asociados a los mismos[228, 106]. Por otra lado, los balanceadores de carga basados en redirección de los paquetes a niveles 2/3 son menos intrusivos, pero no permiten emplear servidores heterogéneos, con reparto de carga a nivel 7.
- Los balanceadores de carga requieren almacenar una gran cantidad de estado para mantener la afinidad entre los clientes y los servidores.
- Los balanceadores de carga globales basados en el DNS no conocen la dirección IP del cliente sino la de su servidor de DNS, que suele estar a varios saltos de distancia [220].

Por esta razón, numerosos autores han estudiado la posibilidad de realizar el reparto de carga directamente en los sistemas finales, es decir, los clientes son conscientes de la existencia de múltiples servidores y eligen el más adecuado. Por tanto, no es necesario utilizar ningún elemento intermedio y los clientes pueden comunicarse directamente con los servidores. Además, en caso de fallo los clientes pueden conocer servidores alternativos a los que re-conectarse. Por ejemplo, muchas implementaciones de aplicaciones de red como `telnet`, `ftp`, o SMTP utilizan un esquema de tolerancia a fallos parecido: cuando la consulta al DNS devuelve varias direcciones IP, se van probando secuencialmente hasta encontrar una accesible.

Otro ejemplo rudimentario consiste en emplear una selección manual: Una página web lista un conjunto de sitios de réplica donde puede encontrarse el fichero, y el usuario elige uno de ellos para realizar la descarga. Obviamente esta técnica no ofrece la más mínima garantía de reparto de carga entre las

diferentes sedes (los usuarios tienden a elegir la primera o la última entrada), pero proporciona cierto nivel de tolerancia a fallos manual.

Pasando a técnicas automáticas de reparto de carga en el cliente, el primer experimento a gran escala fue realizado por Netscape en su portal web [164]. Cuando los navegadores de Netscape accedían a él, seleccionaban al azar un registro del DNS en el rango `home{1-32}.netscape.com`. Obviamente esta técnica requería modificar los navegadores y por tanto no era aplicable a otros sitios web. Por el contrario, los proyectos Smart Clients [267] y Web++ [248] empleaban *Applets* Java para poder utilizar sus algoritmos de selección de servidor en cualquier navegador web.

En las redes de distribución de contenidos, donde la información se encuentra replicada en múltiples sitios y se desea acceder al más cercano, la complejidad para que un servidor sea capaz de conocer la localización de un cliente [86, 44, 52] ha fomentado el desarrollo de técnicas basadas en el cliente donde: en primer lugar se obtiene del servidor principal una lista de servidores alternativos [20, 248], y luego se elige el mejor mediante algún heurístico [78], por ejemplo el más cercano al cliente. Para ello existen numerosas propuestas como: medir el número de saltos [105], mediciones activas del RTT [39, 218] o del tiempo de respuesta de los servidores [213].

Quizá la propuesta de selección de servidor en los clientes más interesante sea Rserpool, que se explica detalladamente en el siguiente apartado.

2.6.9. *Reliable Server Pooling* (Rserpool)

Rserpool es un grupo de trabajo del IETF cuyo objetivo [54] consiste en desarrollar una serie de protocolos para construir servicios con alta disponibilidad [77]. Los servicios se ejecutan en un conjunto de servidores redundantes de tal forma que si un servidor falla, otro servidor tomará su lugar. También es posible realizar balanceo de carga entre los servidores según una política configurable.

Tal como puede verse en la figura 2.18, la arquitectura de Rserpool [238] está formada por los siguientes elementos:

Pool. Un conjunto de servidores replicados que proporcionan la misma aplicación o servicio.

Pool Element (PE). Un servidor que pertenece a un *pool*.

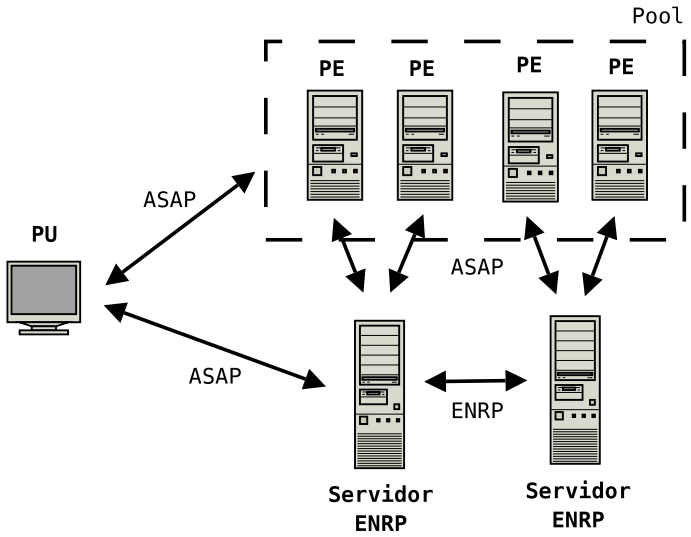


Figura 2.18: Arquitectura de Rserpool.

Pool User (PU). Un cliente de la aplicación que está ejecutando en un *pool* de servidores.

Servidor de nombres o Servidor ENRP. Es el responsable de gestionar el *pool* de servidores y, dado un nombre de *pool*, devolver la información para que un PU pueda conectarse a uno de los PEs que lo forman.

Además de estos elemento funcionales, en Rserpool se definen dos protocolos:

Aggregate Server Access Protocol (ASAP) Los PEs emplean este protocolo para registrarse en un servidor ENRP, mientras que los PUs lo emplean para obtener todos los PEs disponibles del *pool* preguntando a un servidor ENRP. Además, si se emplea en la comunicación PU-PE, ofrece un servicio de transferencia de datos tolerante a fallos, de forma que en caso de fallo, el cambio de servidor sea transparente para la aplicación cliente.

Endpoint haNdlespace Resolution Protocol (ENRP) Los servidores ENRP emplean este protocolo para sincronizar la información que tienen sobre un *pool*, y de esta forma ofrecer un servicio de resolución de nombres distribuido y tolerante a fallos.

En la arquitectura Rserpool se ofrecen dos clases de servicios [54] en función del uso de ASAP para la comunicación PE-PU. En el primer modo, Rserpool

sólo ofrece un mecanismo de resolución del nombre de *pool* mediante una conexión ASAP entre el PU y el servidor ENRP. La comunicación PU-PE no está gestionada por ASAP y por tanto la aplicación debe encargarse de la recuperación frente a fallos del servidor. En el segundo modo, sin embargo, la aplicación se comunica con el servidor mediante ASAP, incluyendo el intercambio de datos, por lo que utiliza automáticamente los mecanismos de tolerancia a fallos y de balanceo de carga de Rserpool.

Endpoint haNdlespace Resolution Protocol (ENRP)

Gracias al protocolo ENRP [258, 230] es posible formar un sistema de registro distribuido y tolerante a fallos para: almacenar, obtener y distribuir información de los *pools* de servidores.

Un *pool* de servidores puede tener varios servidores de nombres en los que se registran los PEs. Así, dado el nombre del *pool*, los PUs puede obtener de un servidor ENRP todos los PEs que forman dicho *pool*.

El servidor ENRP con el que se registra un PE se denomina servidor de nombres "hogar"[33] y es el encargado de mantener actualizada la información de sus PEs (como por ejemplo su carga, o si están accesibles) e informar al resto de los servidores ENRP de cualquier cambio en dicha información. Por tanto, cada servidor de nombres debe tener dos listas de PEs registrados, una con aquellos de los que es su servidor ENRP "hogar" y otra con los PEs gestionados por otros servidores de nombres.

Todos los servidores ENRP de un *pool* están conectados entre sí mediante una malla de conexiones SCTP (Stream Control Transmission Protocol [184]), a través de las cuales se intercambian los mensajes de control. ENRP define un conjunto de mensajes bastante reducido para: intercambiar la información de los *pools* entre los servidores ENRP, conocer la lista de todos los servidores ENRP o el estado de alguno de ellos, y repartirse los PEs asignados a un servidor ENRP caído.

[33] Home, en inglés

Aggregate Server Access Protocol (ASAP)

El protocolo ASAP [231, 230] se emplea para todas las comunicaciones entre elementos de la arquitectura Rserpool, exceptuando a la comunicación entre los servidores de nombres, para lo cual se utiliza el protocolo ENRP descrito anteriormente.

ASAP emplea un mecanismo de direccionamiento basado en nombres que aisla un extremo lógico de la comunicación de sus direcciones IP, lo que permite tener múltiples servidores que ofrecen el mismo servicio. ASAP seleccionará uno de ellos basado en la política de balanceo de carga del *pool*, y enviará los mensajes de aplicación a dicho PE.

A la vez que transmite los mensajes de datos, ASAP monitoriza que el PE en uso siga estando accesible. En el caso que ese PE falle, ASAP es capaz de seleccionar automáticamente otro PE al que enviar los mensajes de la aplicación y por tanto recuperarse ante un fallo del servidor.

Cuando se emplea ASAP para la comunicación PU-PE se establecen dos canales: uno de datos y otro de control. Por el canal de control se transmiten los mensajes ASAP entre el PU y el PE, mientras que el canal de datos se emplea para transportar los mensajes de la aplicación superior. Ambos canales pueden multiplexarse en la misma conexión de transporte o puede suprimirse el canal de datos. Este segundo escenario puede ser utilizado por las aplicaciones legadas, que pierden la tolerancia a fallos de Rserpool, pero a cambio pueden seguir utilizando el mismo protocolo de transporte para acceder al servicio.

Tanto ASAP como ENRP fueron diseñados para trabajar sobre SCTP [229] así que, aunque es posible emplear otros protocolos de transporte como TCP, es necesario definir una capa intermedia entre ASAP y TCP [59] que ofrezca los servicios adicionales de SCTP [53], como puede ser la multiplexación de diferentes canales sobre una misma conexión.

En ASAP se definen primitivas para: (de)registrarse de un *pool* (PE-ENRP), conocer el estado de un PE (ENRP-PE), resolver el nombre del *pool* (PU-ENRP), indicar que un PE se ha caído (PU-ENRP)[34], e intercambiar información para conseguir una sesión de datos fiable (PU-PE).

La compartición de estado entre servidores redundantes queda fuera de los

[34]Este mecanismo puede tener importantes implicaciones de seguridad [233] ya que obliga a que los servidores ENRP confíen en los PUs externos al *pool*.

requisitos [239] de Rserpool, puesto que no es posible definir un mecanismo común para todas las aplicaciones posibles. Sin embargo, Rserpool proporciona un mecanismo de compartición de estado en el caso de que se use ASAP para acceder a los servidores. Con ASAP es posible que el servidor envíe una *cookie* al cliente con el estado de la sesión. En caso de fallo, el cliente se conecta a otro servidor y envía la *cookie* para recuperar el estado en la que se encontraba la sesión justo antes del fallo. Con ASAP cada cliente sólo almacena una *cookie* por sesión, aunque algunos autores [76] sugieren el uso de múltiples *cookies* para minimizar el trasiego de datos en caso de tener información de estado que cambia continuamente.

A diferencia de los balanceadores de carga web, en Rserpool los servidores ENRP no realizan el reparto de carga, sino que los PEs les indican a los PUs que política de balanceo de carga deben emplear. Dado que Rserpool no se centra en ningún aplicación en concreto, su modelo de algoritmos de reparto de carga es extensible, aunque se han definido varias políticas estándar:

Round Robin Cada mensaje se enviará a un PE diferente secuencialmente.

Weighted Round Robin Como la política anterior, pero es posible indicar que unos PEs tienen más capacidad que otros y por tanto recibirán más mensajes.

Menos usado Todos los mensajes se enviarán al PE con menos carga, con lo que, hasta no se actualice la información de carga del *pool*, siempre se escogerá al mismo PE.

Menos usado con degradación Como la política anterior pero cada vez que se selecciona un PE se aumenta su valor de carga.

Modelo de redundancia [259] Con esta política un PE puede definir su rol como servidor activo o de respaldo, si está en servicio o no, o si tiene algún servidor de respaldo designado.

Con estas políticas es posible tener tanto balanceo de carga entre sesiones (cada PU se conecta con el PE menos cargado) o dentro de la misma sesión con varios servidores (cada petición se envía a un PE diferente).

2.7. Descubrimiento de Servicios

Tal como se explicó en la sección anterior, el acceso a una granja de servidores se compone de dos fases: selección del mejor servidor y el acceso propiamente dicho al servicio. Sin embargo, para seleccionar el mejor servidor es necesario conocer previamente qué servidores se encuentran disponibles en ese momento, y por tanto, la fase de selección podría subdividirse en otras dos fases:

1. Localización de los servidores disponibles.
2. Elección del mejor servidor disponible.

La primera fase puede ser tan sencilla como pre-configurar la dirección IP de los servidores en el balanceador de carga o en los clientes finales, o emplear algún otro mecanismo más o menos dinámico. Típicamente para conocer la dirección IP de los servidores que ofrecen un servicio se emplea la resolución de nombres del DNS. Aunque existen registros especiales para especificar servicios tales como el `MX` [160] para los servidores de correo, o el `SRV` [96] para servicios genéricos, normalmente se emplea el mecanismo de resolución de nombres básico, donde el nombre del servidor identifica qué servicio ofrece (e.g. `www.google.com` o `ftp.rediris.es`).

Esto significa que, en última instancia, el cliente tiene que recordar, en lugar de la dirección IP, el nombre DNS del servidor que ofrece cada servicio. Además, el sistema de caches del DNS no es lo suficiente dinámico como para mantener el estado real de los servidores, como por ejemplo si están activos o no.

Por estas razones se han propuesto un gran número de protocolos [27] centrados en el descubrimiento de servicios, esto es, que dado un nombre de servicio, localizan a los servidores que lo ofrecen.

Esta sección estudiará los protocolos de descubrimiento de servicios más importantes, haciendo especial énfasis en la propuesta del IETF, el *Service Location Protocol* (SLP).

2.7.1. *Service Location Protocol* (SLP)

El objetivo del protocolo SLP [98] es la localización dinámica de servicios, dentro de una red local, con un dominio administrativo común. Mediante este

protocolo se intenta evitar que los servicios tengan que estar pre-configurados en cada máquina cliente, o que los usuarios tengan que recordar en qué máquina se encuentra cada servicio. Por el contrario, los usuarios tan sólo tienen que indicar el tipo y los atributos del servicio que desean emplear, y mediante SLP es posible encontrar los servicios activos en la red con dichas características. Una vez localizado el servicio, se puede acceder a él mediante cualquier otro mecanismo de comunicación, puesto que SLP se limita simplemente a encontrar los servicios para los clientes.

SLP puede emplearse tanto en subredes sin infraestructura, como pueda ser la red de una casa o de una pequeña oficina, o en redes de empresas con múltiples subredes, en las que es necesario desplegar cierta infraestructura para centralizar las búsquedas de servicios. Sin embargo SLP no es aplicable para localizar servicios en redes WAN como Internet o en redes con cientos de miles de clientes o decenas de miles de servicios.

En la arquitectura SLP existen tres tipos de entidades:

User Agent (UA) . Un Agente de Usuario es una entidad que emplea SLP para obtener información sobre los servicios, normalmente su URL, en nombre de un usuario/cliente. El Agente de Usuario obtiene la información de los servicios de los Agentes de Servicio o de los Agentes de Directorio.

Service Agent (SA) . Un Agente de Servicio es un proceso que trabaja en nombre de uno o más servicios para publicar la información de dichos servicios. Además, el Agente de Servicio es el encargado de registrar la información de sus servicios en el Agente de Directorio, si este se encuentra presente.

Directory Agent (DA) . Un Agente de Directorio es una entidad que recoge la información de los Agentes de Servicio para ser un repositorio centralizado de información de servicios, y de esta forma optimizar las consultas de los Agentes de Usuario.

La operación básica del protocolo consiste en un cliente que quiere localizar un servicio dentro de la red. Para ello el servicio delega en un Agente de Servicio para publicar su información, mientras que el Cliente emplea a un Agente de Usuario para realizar la búsqueda.

En redes de tamaño reducido, el Agente de Usuario envía la consulta SLP a un grupo *multicast* conocido (o en *broadcast* si la red no soporta *multicast*).

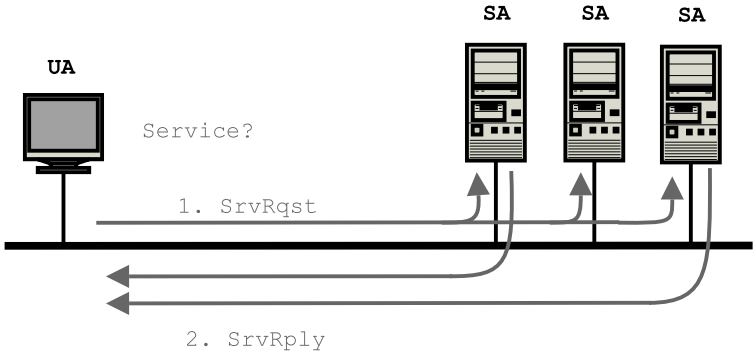


Figura 2.19: Consulta SLP en *multicast*/*broadcast*.

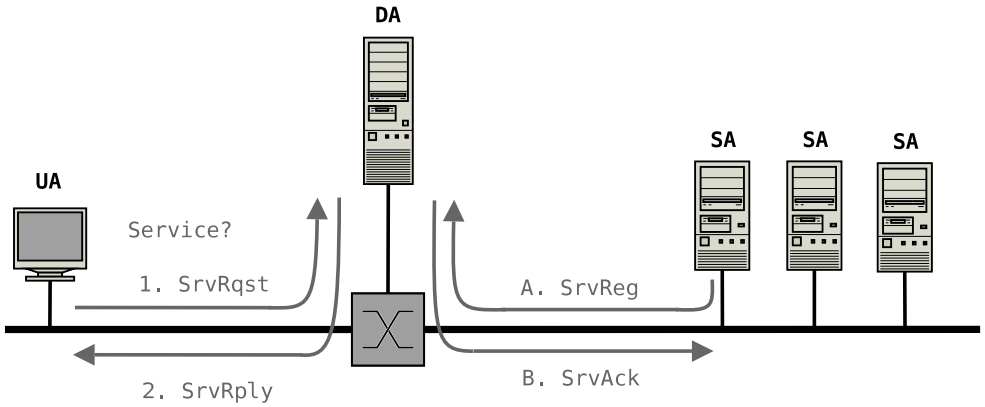


Figura 2.20: Registro de un servicio y consulta SLP a un DA.

Los Agentes de Servicio que estén subscritos a ese grupo *multicast* responden en *unicast*, directamente al UA, con los servicios que se adecuen a la consulta formulada, tal como se muestra en la figura 2.19.

En redes de mayor tamaño es necesario limitar el alcance de las consultas ya que una gran cantidad de mensajes *multicast*/*broadcast* podría saturar la red. Para ello se introduce el concepto de Agente de Directorio, a modo de repositorio centralizado. Los Agentes de Usuario preguntan directamente al DA por la información de los servicios. De esta forma la comunicación es *unicast* y es posible emplear TCP en lugar de UDP, que se usa para las consultas *multicast*. Para que este sistema centralizado funcione (figura 2.20) los Agentes de Servicio deben registrar previamente la información de sus servicios en el DA y opcionalmente de-registrarlos si los servicios dejan de estar accesibles.

Para dividir redes de gran tamaño en unidades organizativas, tales como departamentos o grupos de trabajo, cada servicio puede asignarse a diferentes

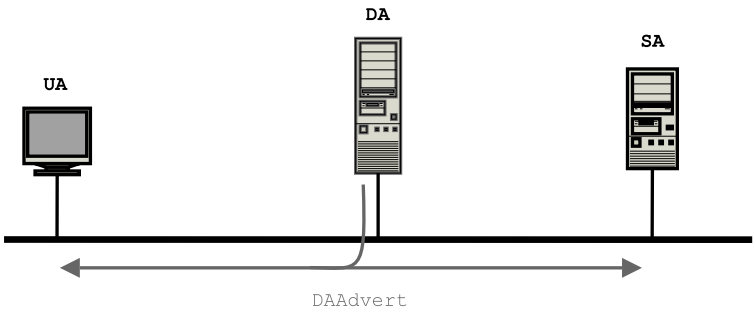


Figura 2.21: Anuncio de un DA en SLP.

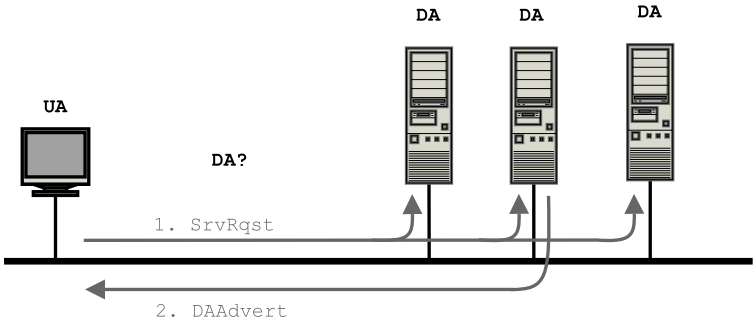


Figura 2.22: Búsqueda de un DA en SLP.

Grupos[35]. De esta forma es posible tener múltiples Agentes de Directorio en una red, donde cada uno se encarga de uno o varios Grupos, y los Agentes de Servicio solamente registran los servicios en los DAs encargados de sus Grupos. Las consultas de los Agentes de Usuario también están limitadas a determinados Grupos, de forma que se envían al DA de cada Grupo, y estos sólo devolverán los servicios pertenecientes a los Grupos especificados en la consulta.

De los dos tipos de funcionamiento, el primer modo (consultas SLP en *multicast*) está limitado a redes LAN pequeñas, pero no es necesario ningún tipo de configuración o de infraestructura. Por otro lado, al dividir los servicios de la red en múltiples Grupos y desplegar varios Agentes de Directorio, es posible aplicar SLP en redes que abarquen varias LAN y tengan un número elevado de usuarios.

El uso de un Agente de Directorio requiere que tanto los Agentes de Usuario como los Agentes de Servicio deben conocer previamente la situación del mismo. Para ello se han definido varios mecanismos: configuración manual, mediante

[35]Scope, en Inglés.

una nueva opción[36] [191] de DHCP, escuchando los anuncios que periódicamente envían los DAs (figura 2.21), o realizando consultas a un grupo *multicast* de una forma parecida (figura 2.22) a la descrita anteriormente para la localización de servicios.

Las operaciones más importantes de SLPv2 [104] permiten: obtener las URLs de los servicios de un cierto tipo (UA-SA/DA), (de)registrar servicios (SA-DA), obtener los atributos de un servicio determinado (UA-SA/DA), localizar a los DAs (UA/SA-DA), y conocer todos los tipos de servicio disponibles en un *Scope* (UA-SA/DA).

Para simplificar el funcionamiento de SLP con consultas *multicast*, un mensaje SLP debe estar contenido en un único datagrama UDP. Por tanto su tamaño máximo está limitado por la MTU de la subred (se especifica el valor de 1400 octetos por defecto). Si un mensaje SLP (una petición o su respuesta) no cabe[37] en un único datagrama UDP, debe repetirse la operación empleando TCP.

En redes sin un Agente de Directorio, pero en las que se quiere realizar una búsqueda exhaustiva, es posible emplear un algoritmo de búsqueda en el que se van eliminando las respuestas de Agentes de Servicio hasta que la búsqueda converja. Para ello, todas las peticiones *multicast* tienen un campo que incluye las direcciones IPv4 de los Agentes de Servicio que han respondido previamente, y sólo deben responder a esa petición los SA que no estén en esa lista. En un datagrama podrían listarse hasta 60 SA, así que este algoritmo sólo puede aplicarse en redes con un tamaño limitado, mientras que para redes mayores sería necesario el uso de un Agente de Directorio.

Además, SLPv2 es extensible, permitiendo definir nuevas cabeceras y operaciones. Por ejemplo, [100] define una extensión para que, junto con la localización de un servicio, pueda obtenerse su lista de atributos en un única petición. En [276, 271] se definen extensiones para limitar el número de respuestas y ordenarlas según los valores de sus atributos, por ejemplo para localizar la impresora más rápida [115]. También se han propuesto nuevas extensiones, como un mecanismo de exclusión [65] que emplea caches en los Agentes de Servicio para que no respondan a peticiones repetidas y de esta forma aumentar la capacidad del algoritmo de búsqueda *multicast*.

Tal como fue definido, SLP sólo permite mecanismos de *pull*, es decir los Agentes de Usuario deben sondear continuamente la red si desean descubrir

[36] Otra opción permite configurar los grupos a los que pertenecen los Agentes
[37] Hay un flag en la cabecera SLP para indicar "overflow".

algún cambio en los servicios. Mediante el uso de una extensión [137] se ha añadido un mecanismo de *push*. Dicha extensión permite a los Agentes de Usuario suscribirse a determinados tipos de servicio, de forma que cuando cambian los servicios de un Agente de Servicio, este anuncia dicho evento a los Agentes interesados. Los DAs tan sólo envían mensajes de notificación cuando expira un registro, y simplemente actúan como una caché de subscripciones de los UAs, para indicar a los SAs a que grupos *multicast* deben notificar cualquier cambio en los servicios del tipo especificado.

Recientemente se ha definido mSLP (*Mesh-enhanced Service Location Protocol*) [274, 275], una extensión a SLP para aumentar la tolerancia a fallos del protocolo. Con este mecanismo, los Agentes de Directorio de un mismo grupo se comunican entre sí formando una malla, e intercambiándose la información de los servicios registrados por los Agentes de Servicio. Utilizando esta extensión un SA mSLP tan sólo tiene que registrar sus servicios en un DA mSLP de cada Grupo, y el DA será el encargado de actualizar la información del resto de los DAs del Grupo, tanto a los que soporten mSLP como a los que no.

Un característica que distingue a SLP de otros protocolos de descubrimiento de servicios, consiste en la utilización de predicados de búsqueda para localizar servicios con unos atributos determinados. Sin embargo dicha característica ha resultado bastante problemática. En SLPv1 se definía una sintaxis de búsqueda propia, mientras que en SLPv2 se optó por los filtros de búsqueda de LDAPv3. Posteriormente, los trabajos de revisión de SLPv2 [99] sugieren limitar los predicados de búsqueda LDAPv3 y permitir solamente operaciones AND.

Otro cambio importante se refiere a la seguridad (para un estudio de la seguridad de SLPv2 ver [247]). En SLPv2bis [102] se abandonan los bloques de autenticación definidos en SLPv1, y se emplea mecanismos de seguridad estándar como IPSec [142] o TLS [71] para asegurar las operaciones de SLP. Un problema de estos mecanismos es que no pueden proteger las búsquedas *multicast* y por tanto se prohibe su uso en entornos sensibles.

Otro aspecto controvertido de SLP es la identificación y el modelado de los servicios. En SLP se emplea el esquema URI [25] `service:` tanto para identificar el tipo de un servicio como para especificar su localización mediante una URL [155]. Además, cada tipo de servicio puede tener una plantilla [103] asociada que define la sintaxis de la URL del servicio y, opcionalmente, un conjunto de pares atributo-valor con información adicional sobre el servicio. Para cada atributo es posible definir su tipo (booleano, entero, cadena de texto o secuencias opacas de

bytes) y otras características, como si son opcionales u obligatorios, si pueden tener múltiples valores, un valor por defecto, o el rango de valores permitidos.

Se han definido plantillas SLP para localizar diferentes servicios, tales como[38]: impresoras [84, 114], servidores Diameter [35], pasarelas de Telefonía IP-POTS [273], servidores de almacenamiento [192, 24], servidores de terminales TN3270E [168], pasarelas [139] RSIP, o servidores SIP [205].

Aunque SLP no define ningún mecanismo de reparto de carga, algunas de estas plantillas permiten emplean técnicas simples para balancear sesiones. Por ejemplo, en [273] se define un atributo de servicio que indica cuantas líneas libres tiene un pasarela, mientras que en [168, 139] se define específicamente un atributo de "carga del servidor". Este atributo puede tener valores entre 0 y 100, y permite que el cliente pueda elegir el servidor con menos carga. Aunque no se detalla cómo debe calcularse la carga de los servidor, en [168] se define mejor la técnica de balanceo de carga, indicando que la prioridad de un servidor puede ser ajustada por el administrador, sumando a la carga de cada servidor un valor constante.

Nótese que, a no ser que se emplee alguna extensión [100, 276] de SLPv2 para obtener la lista completa de atributos junto con su localización, es necesario realizar múltiples peticiones SLP para seleccionar el servidor menos cargado. En primer lugar es necesario obtener la lista con la localización de los servicios, y después pedir para cada servicio su lista de atributos.

Aunque el objetivo de SLP es la localización de servicios en redes de área local, o por lo menos de una extensión reducida, se ha definido un mecanismo experimental [277] para emplear Agentes de Directorio que sirvan de pasarela para localizar servicios en redes remotas. Para ello, en lugar de establecer su propia jerarquía de servidores SLP, se aprovecha la infraestructura del DNS [159, 160]. Los Agentes de Directorio "pasarela" se anuncian en el DNS mediante registros `SRV` [96], y se configuran con la información sobre los DAs y SAs de su dominio accesibles desde el exterior. Con esta infraestructura, el cliente es capaz de localizar los DAs y SAs remotos, (anunciados como servicios `service:slp` en el DA pasarela) para, a continuación, realizar las peticiones SLP y localizar en última instancia los servicios remotos.

Como ya se ha mencionado, un aspecto muy criticado [101] de SLP es la representación de un servicio como una simple URL, que además se emplea

[38] La lista completa de las plantillas registradas por el IANA se encuentra en http://www.iana.org/assignments/svrloc-templates.htm

como su identificador. Algunos problemas derivados de este diseño son:

- Si un servicio está accesible en varias localizaciones (por ejemplo un servidor con varias direcciones IP) es necesario crear un servicio para cada una de ellas.
- Un servicio accesible mediante varios protocolos de nivel de aplicación (e.g. ipp, lpr) tiene que registrarse múltiples veces.
- La URL de un servicio no indica que protocolo de transporte puede utilizarse para acceder a dicho servicio (e.g. TCP o SCTP).
- No es posible retener la identidad de un servicio que cambia de localización.

Para solventar estos problemas se ha sugerido [101] reemplazar la URL del servicio con una URI (con un identificador UUID [149] único) para identificar al servicio en sí, y almacenar la localización del servicio en los atributos asociados a dicho identificador. Aunque este mecanismo es compatible con el protocolo SLP, tal como está definido, modifica en gran medida los procedimientos de uso, puesto que con esta solución es necesario recuperar los atributos de un servicio para obtener su localización.

2.7.2. Jini

Jini es una arquitectura [128] de servicios desarrollada por Sun Microsystems como una extensión de su lenguaje de programación Java. Su objetivo es desarrollar dispositivos Java que ofrezcan servicios RMI (*Remote Method Invocation*) a través de la red. Al contrario que SLP, Jini se encarga tanto de la localización de los dispositivos como de la interacción entre ellos.

La arquitectura de Jini es muy parecida a la de SLP. Para unirse a una comunidad Jini, los dispositivos y aplicaciones se registran en un servidor de directorio (similar a un DA de SLP) empleando un proceso denominado *Discovery and Join*. Además de registrar su localización, los dispositivos pueden cargar en el servidor el código Java del controlador para interaccionar con ellos. Cuando un cliente quiere utilizar el servicio, también puede descargarse dicho controlador, que por ejemplo puede tener una interfaz gráfica[39]. En [93] se hace una comparación exhaustiva entre SLP y Jini, según la cual SLP emplea menos ancho de banda y tiene una menor latencia que Jini.

[39]Esta característica es muy interesante en el caso que el cliente sea un humano

2.7.3. Salutation

Salutation ha sido desarrollado por un consorcio de empresas y ya hay varias implementaciones comerciales disponibles. Su arquitectura [212] está formada por varios intermediadores de servicios denominados *Salutation Managers* (SLMs). Los servicios registran sus funcionalidades en un SLM y los clientes preguntan a un SLM cuando necesitan un servicio. Después de descubrir el servicio solicitado, los clientes pueden pedir la utilización del servicio a través del SLM. Por tanto Salutation, a diferencia de SLP, también define la interacción entre los clientes y los servicios, aunque es independiente de los niveles de transporte y de red.

2.7.4. *Universal Plug and Play* (UPnP)

Universal Plug and Play (UPnP) está siendo desarrollado por un consorcio de compañías liderado por Microsoft, e intenta extender su tecnología *Plug and Play* a los dispositivos en red. UPnP ha definido el *Simple Service Discovery Protocol* (SSDP) [92], que emplea HTTP sobre UDP para localizar servicios en redes IP. El entorno de aplicación de UPnP se reduce a redes LAN pequeñas y, a diferencia de SLP o Jini, no emplea ningún servidor de registro centralizado, sino búsquedas y anuncios de servicios en *multicast*. Para intentar reducir el tráfico los clientes tienen una cache con los servicios encontrados/anunciados. Dado que se ha optado por la simplicidad, en la especificación no se permite que los clientes busquen servicios por sus atributos, sino tan sólo por su tipo.

2.7.5. *Secure Service Discovery Service* (SDS)

A diferencia del resto de arquitecturas descritas, el proyecto SDS [62], desarrollado en la universidad de Berkeley, tiene como objetivo el descubrimiento de servicios en Internet. Para ello emplea una estructura distribuida de servidores SDS. Cada servidor es el responsable de una zona, definida con prefijos de subred, y periódicamente anuncia, a través de un grupo *multicast* bien conocido, la dirección del grupo *multicast* que deben emplear los servicios de dicho dominio para anunciar su información, así como la información de contacto de la Autoridad de Certificación y el Gestor de Acceso locales. Estas entidades expiden certificados para autenticar a los diferentes elementos de la arquitectura, y para controlar el acceso de los clientes a la información de los servicios. Cada

servicio también se anuncia periódicamente para que los servidores SDS actualicen su información. Cuando un cliente desea localizar un servicio, realiza una consulta al servidor SDS de su dominio empleando RMI. Dicha consulta puede contener atributos del servicio, de forma que el servidor SDS sólo responderá con los servicios que se ajustan a dicha plantilla. Cabe destacar que SDS emplea XML para la descripción de los servicios, a diferencia de los protocolos descritos anteriormente, que utilizan modelos de servicio muy simples, basados en pares atributo-valor.

Capítulo 3

Simple Assistant-Router Architecture (SARA)

3.1. Planteamiento del problema

Los *routers* IP actuales son unos complejos sistemas capaces de encaminar millones de datagramas por segundo, y soportar múltiples protocolos de encaminamiento y gestión. Sin embargo, hoy en día las necesidades de las empresas y de los operadores de red van más allá del transporte de datos, y hay una clara tendencia a integrar en los *routers* aún más funcionalidades para el procesamiento de paquetes, que permitan el desarrollo de nuevos servicios de red.

Buena prueba de esta tendencia es el hecho de que la mayoría de los *routers* modernos ofrecen un gran número de funcionalidades adicionales, como la clasificación de paquetes, conformado de tráfico, traducción de direcciones, reglas de control de acceso, políticas de encaminamiento, balanceo de carga, túneles cifrados, etc. Es más, productos comerciales diseñados para regular el acceso a Internet de las organizaciones, como Packeteer [186], basan su éxito en su habilidad para identificar flujos de aplicaciones concretos, analizando los datos L7 en lugar de la clasificación clásica basada en puertos de nivel de transporte, y de este modo asignar a cada aplicación una serie de políticas de control de tráfico totalmente programables.

Aún así, la capacidad de los diseñadores de *routers* para añadir funcionalidades innovadoras a sus equipos es muy inferior a la que requiere el mercado. Esta situación se debe a una combinación de factores, pero esencialmente viene

dada por la integración vertical de los dispositivos de red, lo que dificulta la interoperabilidad entre fabricantes, el despliegue de nuevos servicios en la red, e impide que los planos de datos y de control evolucionen independientemente.

La comunidad investigadora no ha sido ajena a esta tendencia de incrementar las capacidades de procesamiento de los nodos de red, y ha desarrollado el paradigma de redes programables para intentar solucionar los problemas asociados a esta evolución. Bajo el término "redes programables" existen dos líneas de trabajo interrelacionadas, que intentan simplificar el desarrollo de nuevos protocolos y servicios de red: por un lado la estandarización de interfaces abiertas intenta fomentar la interoperabilidad entre fabricantes, mientras que la investigación en redes activas ha evaluado las ventajas de realizar procesamiento de alto nivel dentro de la red para permitir el despliegue dinámico de nuevos protocolos y servicios de red.

Parece claro que la solución a largo plazo pasa por rediseñar las arquitecturas actuales de los *routers* para construir nodos de red programables, con rutinas de encaminamiento flexibles, y que permitan el procesamiento avanzado de flujos a velocidad de línea. Sin duda, para ampliar la capacidad de procesamiento por paquete será necesario realizar avances *hardware* en procesadores de red genéricos, o específicos, en la línea de las tarjetas de servicios [133] de Juniper.

Sin embargo, no parece probable que a corto o medio plazo sea posible realizar cambios de diseño tan amplios, y en cualquier caso también será necesario reemplazar buena parte de infraestructura de red desplegada hoy en día, dado que los *routers* actuales son relativamente poco flexibles, al estar optimizados casi en exclusiva para el reenvío de datagramas. Incluso la utilización de la CPU principal para realizar procesamiento adicional, tal como se hace hoy en día para tratar los casos especiales, no ofrece el rendimiento adecuado, y además puede interferir con los procesos de encaminamiento y gestión del *router*.

Por tanto sería muy interesante desarrollar mecanismos que permitiesen extender las capacidades de procesamiento de los nodos de red actuales, pero que no requieran cambios significativos en la arquitecturas de los *routers* diseñados hasta la fecha, ni extensiones que comprometan la estabilidad o el rendimiento de los mismos.

3.2. Principios de diseño

El objetivo de esta investigación es el diseño de una arquitectura de nodo de red programable que permita ampliar las capacidades de procesamiento de un *router* de altas prestaciones "legado". Es decir, que esté basada en una plataforma *hardware* optimizada para el encaminamiento de paquetes, pero que dispone de un número de funcionalidades reducido, y que en cualquier caso no permite el desarrollo de servicios de red adicionales.

Esta arquitectura se basa en tres principios de diseño fundamentales:

1. **Desacoplar el encaminamiento del procesamiento**. Aunque en principio sería posible emplear la CPU central del *router* para realizar parte del procesamiento de los paquetes, es preferible separar el reenvío de los paquetes de su procesamiento de alto nivel. Por tanto el *router* debe limitarse al encaminamiento de los paquetes, la tarea para la que ha sido diseñado, mientras que el procesamiento avanzado se delega a un sistema de computación externo.

   De este modo, el encaminamiento de los paquetes está totalmente aislado del procesamiento adicional, y aunque algún servicio de red fallase, o se saturase y consumiese demasiados recursos, el *router* podrá seguir reenviando paquetes y ejecutando los protocolos de encaminamiento, por lo que la conectividad de la red no se vería afectada.

2. **Procesamiento transparente**. El procesamiento de los paquetes por entidades externas al *router* parece una plataforma ideal para el despliegue de redes superpuestas[1] [9], donde los paquetes se envían salto a salto entre *proxies* situados dentro de la red. Sin embargo también debe permitirse el procesamiento "transparente" de los paquetes, es decir, que los sistemas finales puedan enviar los paquetes directamente entre sí, y que sean los nodos de red programable intermedios los que los capturen y los procesen, de acuerdo al servicio de red que estén ejecutando

   Este segundo modo de funcionamiento simplifica enormemente el despliegue de los servicios de red puesto que, a diferencia de las redes superpuestas, los sistemas finales no tienen porque conocer a los *proxies* de procesamiento intermedios, es más, ni siquiera tendrían que ser conscientes de la existencia de ciertos servicios de red.

[1] Overlay Networks, en inglés.

3. **Código seguro**. A pesar de ser uno de los objetivos prioritarios de la investigación en redes activas [6], el despliegue dinámico y sin restricciones de servicios basado en código móvil tiene unas repercusiones en la seguridad demasiado graves como para justificar su uso. Los mecanismos de comprobación dinámica de código móvil son extremadamente costosos y los lenguajes seguros con capacidades limitadas son demasiado restrictivos. Por esta razón la responsabilidad del despliegue de nuevos protocolos y servicios debe estar limitada exclusivamente al administrador de la red.

Además, para que realmente tenga sentido extender un *router* legado, en lugar de reemplazarlo por otro que disponga de una arquitectura más flexible y que permita implementar los nuevos servicios de red, la plataforma resultante debe cumplir los siguientes requisitos:

- **Bajo coste**. Esto implica que el sistema de procesamiento externo debe tener un coste reducido, y la adaptación del *router* a la nueva arquitectura debe ser muy sencilla, por lo que deben minimizarse los cambios requeridos al *router*.
- **Extensible**. El nodo de red resultante debe permitir, dentro de ciertos límites, el despliegue y la ejecución de cualquier tipo de servicio de red. Por tanto es preferible que el sistema de procesamiento externo sea de propósito general en lugar de emplear *hardware* específico, lo que además encarecería su coste.
- **Escalable**. Normalmente los *routers* que no pueden ampliarse con nuevas funcionalidades son nodos de altas prestaciones con soporte de encaminamiento en *hardware*, por tanto es razonable pensar que darán servicio a cientos o a miles de usuarios. Esto quiere decir que el sistema de procesamiento externo debe ser capaz de soportar aplicaciones que den servicio a un gran número de clientes.

3.3. *Simple Assistant-Router Architecture* (SARA)

A partir de los objetivos de diseño enunciados en el apartado anterior, se ha desarrollado una arquitectura de nodo de red programable denominada *Simple Assistant-Router Architecture* (SARA) [148], que cumple con los requisitos y las restricciones del dominio de aplicación: construir un nodo de red programable

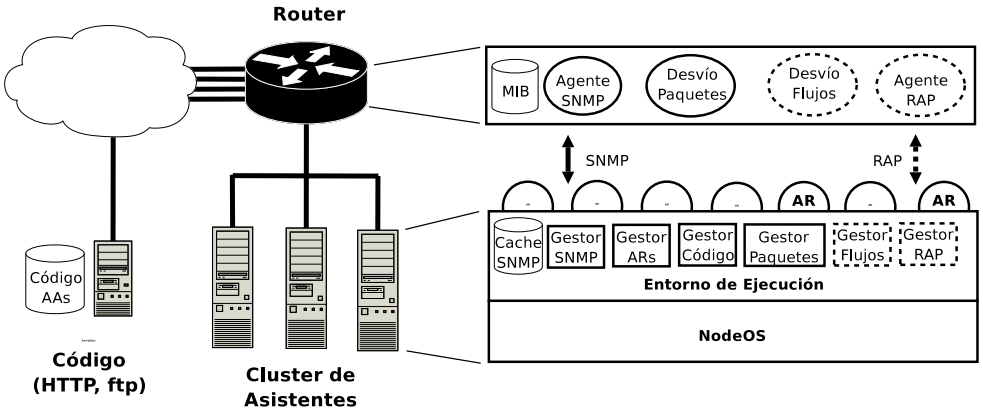


Figura 3.1: Arquitectura SARA.

a partir de un *router* de altas prestaciones actual, que no dispone ni de la flexibilidad ni de la capacidad de proceso necesarias para soportar servicios de red avanzados.

Tal como señalaba el primer principio de diseño, en SARA el nodo de red está formado por un *router* legado, que se encarga esencialmente del encaminamiento de los paquetes, y uno o más sistemas externos, denominados asistentes, que están unidos al *router* mediante una red local de alta velocidad, y en los que se delega el procesamiento avanzado de los servicios de red desplegados.

Los requisitos de escalabilidad, bajo coste, y extensibilidad restringen la utilización de tecnologías de propósito especifico como Myrinet [31] o ASICs. En su lugar se emplea un *cluster* de PCs o estaciones de trabajo, mientras que para la red de área local, que interconecta los asistentes y el *router*, puede emplearse cualquier variante de alta velocidad de Ethernet conmutada, como por ejemplo Fast o Gigabit Ethernet. Sin embargo la arquitectura SARA permite utilizar cualquier otra tecnología de interconexión y/o computación, como por ejemplo la utilización de procesadores de red [58], para servicios que requieran un procesamiento de paquetes a muy alta velocidad.

La figura 3.1 muestra los elementos que forman un nodo de red programable SARA. En particular se detalla la arquitectura de los componentes *software* de un asistente, así como las principales funcionalidades, tanto obligatorias como optativas, que puede implementar un *router* SARA.

3.3.1. Redes activas e Interfaces abiertas

Además de evaluar los beneficios de una arquitectura de nodo de red distribuida, SARA explora el espacio de soluciones situado entre las dos líneas de investigación en redes programables, estudiadas en el capítulo 2: la estandarización de interfaces abiertas, y las redes activas. En particular intenta simplificar tanto el desarrollo como el despliegue de los servicios de red, en lugar de centrarse solamente en uno de los dos objetivos.

Para ello, los asistentes implementan una arquitectura en capas (NodeOS, Entorno de Ejecución y Aplicaciones de Red) muy parecida a la de un nodo de red activa [64], pero al mismo tiempo mucho más pragmática en el uso y despliegue de código, y basada en interfaces bien definidas, razón por la cual la arquitectura SARA se ajusta exactamente al concepto de red programable.

Las ventajas de utilizar un Entorno de Ejecución, frente al uso de interfaces estáticas, son numerosas: además de simplificar el desarrollo y la portabilidad de los servicios de red, facilita el despliegue dinámico de nuevos protocolos y servicios (como Aplicaciones de Red), y permite controlar los recursos que emplea cada uno de ellos [219].

Sin embargo, la utilización de esta arquitectura "programable" no convierte automáticamente a los asistentes en una plataforma de computación "pública", en el sentido de que cualquier usuario de la red pueda ejecutar sus propias rutinas (recordemos que, según el tercer principio de diseño, el despliegue dinámico de código es responsabilidad exclusiva del administrador de red).

Por tanto no tiene sentido utilizar mecanismos de "cápsulas activas" (paquetes con datos y código móvil), sino que el código de los servicios que pueden desplegarse en la red se almacena en uno o más servidores de código seguros. De esta forma, cuando se desea desplegar un nuevo servicio de red, el Entorno de Ejecución de los asistentes puede descargar su código del servidor, y cargar dinámicamente la Aplicación de Red, que podrá comenzar a procesar paquetes inmediatamente, sin necesidad de realizar comprobaciones de seguridad previas o en tiempo de ejecución sobre el código.

Gracias a las simplificaciones derivadas de la utilización de fuentes de código restringidas y fiables, la arquitectura SARA no pone ninguna restricción: ni a la descarga de código, por lo que es posible emplear cualquier protocolo de transferencia de ficheros estándar (e.g. ftp, http, https), ni a la carga dinámica

de código, lo que permite utilizar cualquier tipo de lenguaje portable (e.g. Java, lenguajes de *script*). En realidad podría emplearse cualquier tipo de lenguaje con librerías dinámicas (e.g. C) ya que, gracias a la utilización de asistentes, un operador puede desplegar una plataforma de computación homogénea y de bajo coste en su red, incluso si está formada por *routers* de diferentes modelos y fabricantes.

3.3.2. Desvío de paquetes de señalización

Además de los aspectos puramente técnicos (e.g. lenguajes portables, carga dinámica de código) no hay que olvidar que, típicamente, el despliegue de un nuevo servicio de red requiere la re-configuración de todos los sistemas finales que desean utilizarlo. Por tanto, una vez definida una plataforma de procesamiento estándar, que simplifica el desarrollo de servicios de red portables, es necesario aliviar el problema del despliegue propiamente dicho.

En este sentido, esta tesis doctoral propone la utilización de paquetes de señalización, (o "paquetes activos" en la terminología de Redes Activas[2]), que permitan a los sistemas finales localizar dinámicamente a los nodos de red programables que ofrecen un determinado servicio. De esta forma se evita que los sistemas finales tengan que estar pre-configurados con la localización de, al menos, uno de los nodos que ofrecen el servicio de red, ya que, además de ser una fuente de errores y de complejidad adicional, este tipo de configuración manual puede generar redes superpuestas sub-óptimas, al no adaptarse a los cambios en la topología subyacente o en los nodos de red que la forman.

El mecanismo propuesto funciona de la siguiente manera: los sistemas finales, puesto que no conocen a priori la situación de los nodos de red que ofrecen el servicio de red deseado, envían paquetes de señalización directamente al destino final, solicitando el servicio de red que desean emplear. Estos paquetes atravesarán la red como cualquier otro datagrama IP, pero al llegar a un nodo de red programable, el *router* lo desviará a un asistente para que sea procesado por el servicio de red solicitado[3].

En este punto, dependiendo del servicio de red (e.g. *proxy* Web/SOCKS),

[2] En cierto sentido, la función de un paquete de señalización que contenga órdenes de control es semejante a la de los "paquetes activos", puesto que altera el comportamiento de la red sobre un conjunto de flujos de datos

[3] Aunque, dependiendo del servicio, también podría ser analizado y procesado por el plano de control del *router* directamente

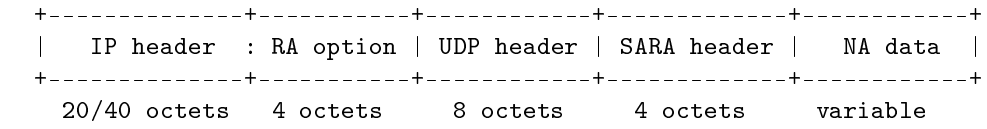


Figura 3.2: Formato de un paquete de señalización SARA

el asistente podría responder al cliente con su dirección IP para que éste le enviase el flujo de datos directamente para su procesamiento. Sin embargo, en otros servicios de red (e.g. *multicast* fiable), los sistemas finales no necesitan conocer la situación de los nodos de red programables, sino que les basta con que éstos procesen sus paquetes. Por tanto, en este tipo de aplicaciones, los paquetes de señalización sirven para configurar el tratamiento de los flujos de datos de los nodos de red programables situados en la ruta al destino. En ese caso las aplicaciones de red, después de procesar el paquete de señalización, lo re-inyectarán en el *router* para que siguiese su camino hacia el destino.

Dada la gran utilidad de este mecanismo, tanto para los servicios de red "transparentes", como para localizar a los nodos de red programables en los servicios con direccionamiento explícito, el desvío de los paquetes de señalización al asistente es la única modificación del *router* obligatoria dentro de la arquitectura SARA. Por tanto es imprescindible que este mecanismo pueda ser implementando sin que sea necesario efectuar cambios significativos en el *router*.

En realidad el desvío de paquetes necesita dos mecanismos básicos: la detección de los paquetes de señalización que atraviesan el *router*, y el desvío propiamente dicho de los paquetes al asistente.

Para poder realizar la primera tarea eficientemente, con la tecnología disponible en los *routers* actuales, se ha optado por emplear la opción *Router Alert*, obligatoria para los *routers* IPv4 [188] e IPv6 [135], que les indica que ese paquete contiene información de señalización[4], a tratar en el *Slow-path* del *router*.

Por tanto los paquetes de señalización SARA (figura 3.2) son datagramas IPv4 o IPv6 con la opción *Router Alert* activa, que contienen una cabecera UDP, seguida de la cabecera SARA, y los datos de la Aplicación de Red[5]. Para diferenciar los paquetes de señalización SARA de otro tipo de mensajes de señalización (e.g. RSVP), alguno de los puertos UDP debe ser 3323.

[4]De hecho, en IPv6 uno de los valores (0x0002) de la opción está reservado para paquetes activos.

[5]Network Application, en inglés

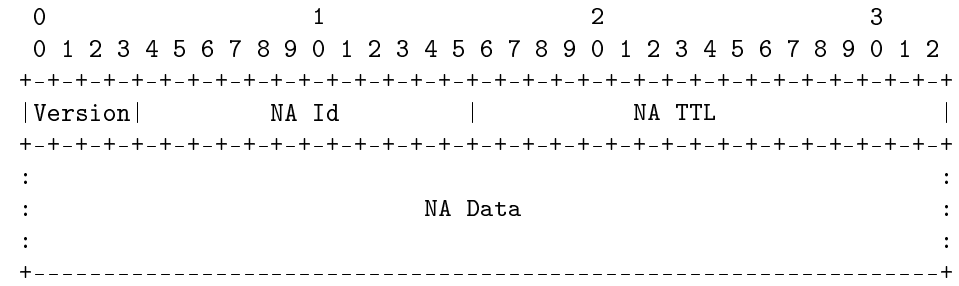


Figura 3.3: Formato de la cabecera SARA

En cuanto a la cabecera SARA (figura 3.3) está formada por un campo `version` para permitir futuras extensiones, un identificador de Aplicación de Red (`NA Id`) para que el Entorno de Ejecución sepa a qué servicio de red debe entregarle dicho paquete, un campo de `NA TTL` para actualizar el tiempo de vida de la aplicación de red y su *soft-state* asociado, y por último los datos de aplicación.

Por tanto, los *routers* actuales pueden detectar fácilmente los paquetes de señalización, ya que siempre tienen que comprobar la existencia de la opción *Router Alert* en cada datagrama IP que encaminan, y simplemente habría que modificar ligeramente las rutinas de tratamiento de dicha opción para identificar a los paquetes de señalización SARA.

Una vez localizados, los paquetes de señalización tienen que desviarse a un asistente, y para ello la solución más sencilla consiste en reenviarlo normalmente, encapsulado en una trama de nivel de enlace, como si el asistente fuese el siguiente salto en la ruta del datagrama a su destino. Para implementar este mecanismo bastaría con saltarse la consulta de la tabla de rutas, o sobreescribir en la meta-información asociada al paquete, la dirección del siguiente salto, reemplazándola con la dirección del asistente.

Por tanto la implementación de ambos mecanismos es muy sencilla puesto que son muy poco intrusivos con el plano de datos del *router*. El único aspecto que puede incrementar mínimamente la complejidad es el reenvío de paquetes de señalización desde los asistentes.

Al igual que cuando fueron desviados, los paquetes de señalización reenviados por los asistentes llegarán al *router* encapsulados en tramas de nivel 2 (aunque a nivel IP seguirán estando dirigidos al sistema final destino). Por tanto basta con que el *router* los reenvíe como si fueran un datagrama IP normal, aunque

tengan la opción *Router Alert* que los identifica como mensajes de señalización. Esto significa que en el proceso de identificación deben ignorarse los paquetes de señalización recibidos por los puertos a los que está conectado el *cluster* de asistentes, o de lo contrario se producirá un bucle de paquetes de señalización reenviados por los asistentes y desviados una y otra vez por el *router*.

3.3.3. Cooperación *router*-asistente

El desvío transparente de paquetes de señalización puede considerarse el primer mecanismo de cooperación entre el *router* y los asistentes, ya que posibilita el despliegue de servicios transparentes, o basados en una red superpuesta de *proxies* que se comunican entre sí, empleando la red IP subyacente meramente como un mecanismo de transporte. Sin embargo, una de las principales ventajas del procesamiento dentro de la red consiste, justamente, en poder conocer el estado de la misma inmediatamente, y así reaccionar a cambios que se producen dentro de la misma. Por esta razón ciertos servicios de red necesitan acceder al estado del *router* mediante algún mecanismo de cooperación *router*-asistente adicional.

La utilización del *Simple Network Management Protocol* (SNMP) es una solución obvia para este problema, puesto que en la actualidad es el mecanismo de gestión más extendido y todos los *routers* comerciales lo soportan, con lo que puede emplearse directamente sin que se requiera ninguna modificación adicional.

Sin embargo SNMP fue diseñado para que los agentes fueran sondeados periódicamente desde un número limitado de estaciones de gestión, por lo que en el caso que varias aplicaciones necesitasen acceder continuadamente al estado del *router*, los asistentes podrían generar un número de consultas SNMP excesivo, que podrían perjudicar al rendimiento de la CPU central de *router*, y por ende al funcionamiento normal del mismo.

Una solución parcial a este problema [147] consiste en añadir a los asistentes una cache con las consultas SNMP recientes, para que todas las aplicaciones puedan compartir la MIB del *router* localmente.

Lamentablemente SNMP tiene otro problema de difícil solución: la MIB estándar de los *routers* sólo ofrece una abstracción general del dispositivo, por lo que mediante SNMP no se puede obtener información detallada sobre el

estado del *router*, ni es posible configurar el plano de datos del *router* con la flexibilidad que necesitan los servicios de red actuales[6] [215].

Por esa razón, y aunque el uso de SNMP para consultar información general del *router* puede ser suficiente para algunas aplicaciones, es necesario definir un mecanismo de cooperación adicional, que permita acceder al estado detallado del *router* de manera eficiente, así como configurar el plano del datos del mismo.

Esta tesis define un mecanismo alternativo, denominado *Router-Assistant Protocol* (RAP), gracias al cual, todas las funciones del plano de datos del *router* pueden estar disponibles desde un asistente externo, incluyendo: el acceso a la información de estado de bajo nivel, la re-configuración del plano de datos del *router*, o la capacidad de extraer/re-inyectar paquetes en cualquier punto del proceso de reenvío.

Aunque la descripción completa de RAP se encuentra en el capítulo 4, es obvio que un protocolo de estas características requiere un soporte significativo por parte del *router*, a diferencia del resto de los mecanismos de cooperación definidos en SARA. Por esta razón, en lugar de desarrollar un modelo de *router* abstracto desde cero, RAP está basado en el protocolo de control ForCES, actualmente en fase de diseño, y es compatible con la arquitectura de nodo de red ForCES, de forma que cualquier *router* futuro que soportase el protocolo ForCES podría ampliarse fácilmente con asistentes externos.

3.3.4. Desvío transparente de flujos de datos

Una de las características más interesantes del protocolo RAP es la capacidad de configurar el plano de datos del *router* para que este desvíe al asistente, no sólo los paquetes de señalización, sino además flujos de datos convencionales. Es más, empleando RAP como encapsulación, es posible enviar al asistente, además del paquete en sí, la meta-información asociada al mismo, de forma que el servicio de red que ejecuta en el asistente pueda conocer, por ejemplo, por que puerto del *router* se ha recibido el paquete.

Una vez procesados, los paquetes de datos pueden ser reenviados por los asistentes normalmente, enviándoselos al *router* encapsulados en tramas de nivel de enlace para que este los encamine hacia su destino. Sin embargo, con RAP

[6] De hecho, hoy en día SNMP sólo se utiliza para obtener el estado de los dispositivos de red, mientras que para su configuración suelen emplearse otros mecanismos mucho más flexibles, como la interfaz de línea de comandos (CLI).

también es posible re-inyectar un paquete en cualquier punto del plano de datos del *router*. Por ejemplo, una aplicación de *multicast* fiable puede retransmitir un paquete *multicast* por un subconjunto de las interfaces del *router*, indicando los puertos de salida en la meta-información asociada al datagrama, que está encapsulada dentro de un mensaje RAP.

Sin embargo, la utilización de un *cluster* de asistentes, y de políticas de reparto de carga en SARA afecta en gran medida al desvío de flujos, dado que es razonable suponer que los servicios de red que traten flujos específicos también guardarán información por flujo, y por tanto todos los paquetes de un flujo deberían desviarse siempre al mismo asistente.

Para ilustrar este punto supongamos que se desea implementar un NAT escalable, empleando un nodo constituido por múltiples asistentes. Para ello cada asistente se encargará de un subconjunto de los flujos que atraviesan el *router* y almacenará la tabla de traducción de sus flujos. Por tanto, el desvío de flujos requerirá: primero identificar los datagramas con direcciones privadas, y luego desviar dichos datagramas al asistente encargado del flujo al que pertenecen.

La solución más inmediata consistiría en almacenar en el *router* una tabla con todos los flujos desviados, indicando a qué asistente está asignado cada uno. Sin embargo con un gran número de flujos, la búsqueda en dicha tabla por cada paquete reenviado por el *router* sería inviable. Una alternativa que no requiere conocer todos los flujos es la utilización de una función *hash* [146] que, dado un identificador de flujo determinado, siempre devuelve a qué asistente deben desviarse los paquetes con el mismo identificador, y que, estadísticamente, es capaz de repartir los flujos equitativamente entre todos los asistentes.

Un problema de esta solución es que un cambio en el número de asistentes, por ejemplo si falla uno de ellos, requiere reemplazar la función *hash*, y por tanto la mayoría de los flujos previos se re-asignarían a un asistente diferente. Para solucionar este problema se han propuesto técnicas de *Robust Hashing* [206] que permiten mantener las asignaciones previas ante un cambio en el número de asistentes, pero a costa de realizar tantas operaciones *hash* como asistentes estén activos.

El problema del *Robust Hashing* es que las funciones de *hash* realizan operaciones relativamente complejas y, dado que a altas velocidades el tiempo de procesamiento por paquete es bastante reducido, el número de operaciones de *hash* que puede ejecutar el *router* limita el tamaño máximo del *cluster*. Por esta

razón, en el capítulo 5 de esta tesis se definen varios algoritmos análogos a los de *Robust Hashing*, pero que permite la asignación persistente de flujos a asistentes empleando un número significativamente menor de operaciones *hash*.

3.3.5. *Cluster* de Asistentes

Una alternativa a la carencia de soporte *hardware* especializado para el procesamiento de paquetes a altas velocidades consiste en explotar el paralelismo de las arquitecturas distribuidas, y en particular las granjas de servidores replicados. De este modo es posible incrementar la capacidad del procesamiento del nodo de red programable añadiendo varios asistentes que ejecutan simultáneamente el mismo servicio de red, repartiéndose equitativamente los paquetes de señalización y los flujos de datos entre ellos.

Sin embargo, para poder aprovechar todo el potencial de las aplicaciones de red replicadas en varios asistentes, es imprescindible utilizar una política de reparto de carga adecuada al servicio de red desplegado. Lo cual es un desafío importante debido a la heterogeneidad de las aplicaciones de red que pueden ejecutar simultáneamente en el *cluster*. Por tanto, a diferencia de las granjas de servidores mono-aplicación tradicionales (como los *clusters* de servidores web estudiados en el Capítulo 2), en este caso será necesario emplear varios tipos de algoritmos de reparto de carga simultáneamente, dependiendo de las características de cada aplicación.

Por ejemplo en los servicios de red con direccionamiento explícito tipo *proxy*, que requieren una gran capacidad de procesamiento (e.g. pasarelas entre diferentes formatos de vídeo), la elección del mejor servidor sólo se realizará una vez por sesión multimedia, y por tanto el algoritmo de reparto de carga puede ser bastante complejo, pero es esencial que se seleccione el asistente menos cargado al que desviarle los paquetes de señalización de la sesión. En el otro extremo de los requisitos se encuentran las aplicaciones de red sin estado pero en las que el *router* debe elegir por cada datagrama el asistente donde será procesado. En este caso es indispensable utilizar algún algoritmo de reparto de carga de bajo coste como *Weighted Round Robin* para servicios sin estado por flujo, o el *Fast Robust Hashing* antes mencionado para realizar asignaciones de flujos a asistentes permanentes.

Por tanto, dada la heterogeneidad de las técnicas de reparto de carga que pueden emplearse, así como la necesidad de conocer el estado real de los asis-

tentes para poder seleccionar realmente al servidor menos cargado, es necesario emplear algún protocolo dinámico para el reparto de carga, similar a Rserpool (ver apartado 2.6.9). De hecho, la enorme diferencia entre los requisitos de las políticas de reparto de carga para cada tipo de servicios es tal, que en realidad se proponen dos mecanismos totalmente diferentes.

Tal como se detallará en el capítulo 4, los servicios de red transparentes emplean RAP para configurar el reparto de carga dentro del plano de datos del *router*, mientras que para los servicios con direccionamiento explícito es preferible emplear un mecanismo de reparto de carga más dinámico y flexible. Para ello se propone un nuevo mecanismo, denominado *eXtensible Service Discovery Framework* (XSDF), que permite la utilización de las políticas de selección de carga más adecuadas para los diferentes servicios de red que pueden cohabitar en el cluster de asistentes. En el capítulo 6 se definirá XSDF y cómo puede aplicarse dentro de la arquitectura SARA.

3.3.6. Seguridad

La seguridad ha sido un tópico muy importante dentro de la investigación en redes activas [6], especialmente en lo referido al estudio de los riesgos y posibles soluciones de la utilización de código móvil. Sin embargo la aproximación pragmática de SARA, basada en utilizar repositorios de código seguros, que sólo distribuyen aplicaciones previamente autorizadas por el administrador de la red, simplifica en gran medida el problema de la fiabilidad del código de los servicios de red.

Aún así la seguridad en redes activas no se limita solamente a este ámbito, y como en cualquier otro sistema distribuido es necesario analizar las implicaciones de seguridad de la arquitectura. De hecho, el estudio de la seguridad en SARA ha sido objeto de un profundo análisis en otro trabajo doctoral [4], donde se detallan diferentes mecanismos para ofrecer servicios de red programable seguros, incluso cuando se realiza procesamiento transparente tanto a flujos de datos como a paquetes de señalización. Por tanto esta tesis doctoral no ahondará en ningún aspecto de seguridad, sino que remitirá al lector a [4].

3.4. Prototipos SARA

Además del diseño arquitectural, dentro de este trabajo de investigación se han desarrollado dos prototipos de nodo de red programable que emplean la arquitectura SARA: una implementación basada en un *router* comercial, y otra basada en un *router software*. En ambos casos se emplea un asistente Linux, que incluye un Entorno de Ejecución, y varias Aplicaciones de Red de ejemplo.

Tanto el Entorno de Ejecución como las aplicaciones de red están programadas con el lenguaje de programación multi-plataforma Java, lo que simplifica el desarrollo de servicios de red independientes de la plataforma del *router*/asistente, y permite la carga dinámica del código de las aplicaciones. Dicho código puede residir tanto en el sistema de ficheros local como en un servidor de código remoto, por ejemplo un servidor web o FTP.

Además de los Gestores de Código y de Aplicación, el Entorno de Ejecución contiene un cliente SNMP que le permite consultar la MIB de cualquier *router* que disponga de un agente SNMP. Para reducir el número de consultas SNMP enviadas al *router*, el Entorno de Ejecución implementa una cache que almacena las respuestas más recientes del *router*, de forma que esta información pueda ser compartida por varias Aplicaciones de Red, aunque también es posible saltarse la cache del EE para obtener el estado actual del *router*.

Las Aplicaciones de Red pueden acceder a cualquier recurso del asistente incluyendo librería estándar de la *Java Virtual Machine* (JVM). Por tanto pueden abrir un *socket* TCP o UDP, y comenzar a comunicarse directamente con sistemas finales, o con otros nodos de red programable para formar una red superpuesta.

Por último, el Entorno de Ejecución permite la recepción transparente de paquetes de señalización, así como el envío de cualquier tipo de datagrama IP (incluso con una dirección origen diferente a la del asistente). Para ello se emplean *sockets* "en crudo"[7], a los que se puede acceder desde el Entorno de Ejecución mediante el *Java Native Interface* (JNI).

[7] Raw Sockets, en inglés.

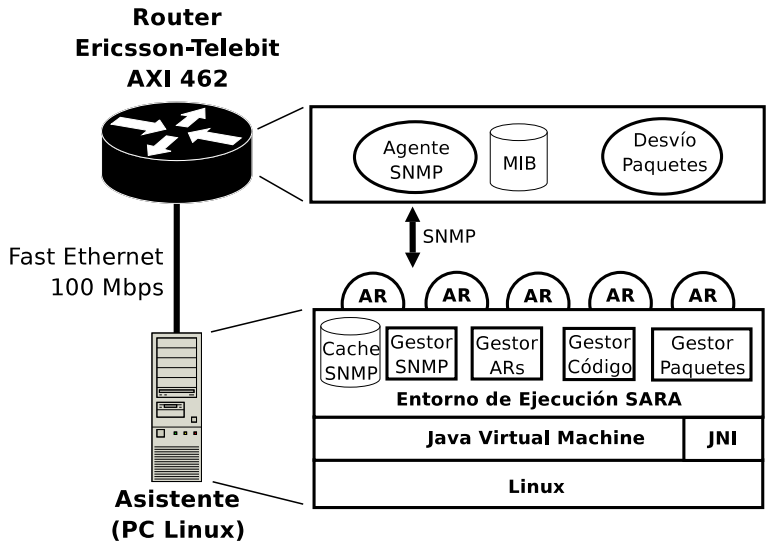


Figura 3.4: Prototipo SARA-Ericsson Telebit AXI462.

3.4.1. Routers Comerciales: Ericsson-Telebit AXI462, Cisco

Quizá la mejor forma de demostrar la viabilidad de la arquitectura SARA propuesta sea el primer prototipo desarrollado, que está basado en el *router* comercial Ericsson Telebit AXI 462 [80]. Tal como muestra la figura 3.4, permite el desvío de paquetes de señalización al asistente, así como el acceso a su información de estado a través de SNMP.

El desvío de paquetes de señalización ha sido la única funcionalidad que ha requerido cambios en el *software* de control estándar del *router*, y se limitó a introducir un nuevo procedimiento en el tratamiento de los paquetes con la opción IP de *Router Alert*, para que estos fueran reenviados al asistente.

Dicho desarrollo fue realizado por los ingenieros de Telebit, y su esfuerzo se estimó en menos de un hombre-mes, lo que puede dar una idea de lo sencillo que puede ser adaptar un modelo de *router* comercial actual a la parte básica de la arquitectura SARA, es decir, para conseguir el desvío transparente y procesamiento externo de paquetes de señalización.

De hecho en cualquier *router* Cisco bastaría con incluir el fragmento de configuración mostrado en la figura 3.5 para desviar los paquetes de señalización a un asistente (`192.168.1.100`), y por tanto podría soportar la parte básica de SARA, sin que fuese necesario modificar la IOS del mismo.

```
ip access-list extended sara_divert
permit  ip any any option router-alert
permit udp any any eq 3323

route-map sara_route
match ip address sara_divert
set ip next hop 192.168.1.100
```

Figura 3.5: Ejemplo de configuración IOS para el desvío de paquetes de señalización SARA

3.4.2. *Router* Linux

Aunque el asistente puede funcionar por sí mismo como un *router software*, se ha desarrollado otro prototipo SARA, basado en un *router* Linux, que es capaz de desviar flujos de datos convencionales al asistente (figura 3.6). Para ello ni siquiera ha sido necesario modificar la pila TCP/IP del *kernel* del *router*, sino que bastó con utilizar las funcionalidades de red avanzadas de Linux. En particular se han empleado los comandos `iptables` e `iproute2` para configurar las reglas de filtrado del *kernel*, y reenviar los flujos seleccionados, empleando una tabla de rutas alternativa, que siempre apunta al asistente como el siguiente salto para los paquetes seleccionados.

En cuanto al asistente, la implementación original fue ampliada para permitir la captura transparente de los flujos desviados por el *router*. En este caso también se emplea las reglas de filtrado del *kernel*, lo que permite marcar cada paquete con un identificador de flujo y luego enviarlos al espacio de usuario a través de un socket `netlink` [210]. El Entorno de Ejecución Java, de nuevo mediante JNI, recibe esos paquetes junto con su meta-información asociada, de forma que, gracias al identificador de flujo asignado por las reglas del *kernel*, es capaz de entregarlos a la aplicación de red que solicitó el desvío de dicho flujo.

Para facilitar el procesamiento de los paquetes desviados, se ha desarrollado una librería que permite trabajar de forma sencilla y eficiente con las cabeceras de los protocolos de red más habituales, tales como IP, TCP, UDP o SARA.

Una vez procesados, los paquetes pueden reenviarse a través del *raw socket* del Entorno de Ejecución, que los encaminará, como cualquier otro datagrama IP, primero al *router*[8] y de ahí al destino final. Al igual que en el caso de los paquetes de señalización desviados, la única precaución adicional consiste en

[8]En los asistentes el siguiente salto de la ruta por defecto siempre es el *router*

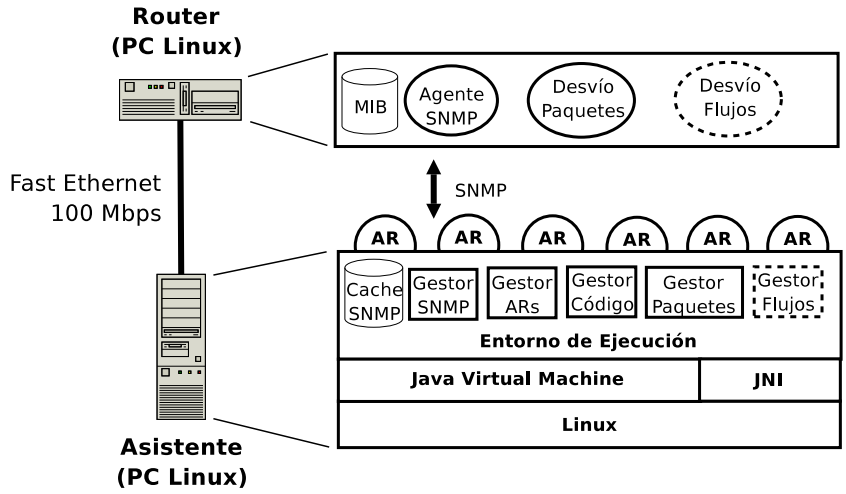


Figura 3.6: Prototipo SARA-*Router Software*.

ignorar, cuando se clasifican los flujos en el *router*, el puerto del asistente, para evitar que los paquetes vuelvan a ser desviados al asistente y se forme un bucle.

En las pruebas realizadas el proceso completo de desvío de flujos, procesamiento por la aplicación, en espacio de usuario, y reenvío al *router* añadía, como máximo, una latencia de 4 milisegundos sobre el proceso de encaminamiento normal del *router*.

3.4.3. Pruebas de rendimiento

A partir del segundo prototipo se realizaron diferentes pruebas para evaluar el rendimiento del procesamiento de paquetes del asistente, y como afectaban diferentes parámetros a dicho rendimiento.

La figura 3.7 muestra el escenario de pruebas. En todos los casos un cliente envía, a diferentes tasas, paquetes de señalización o flujos de datos a un servidor, que comprueba, mediante la tasa de recepción, el rendimiento del asistente que procesa los paquetes. Todos los PCs Linux (*kernel* 2.4.12) empleados son Pentium III 733 MHz con 128 MB de RAM y tarjetas Fast Ethernet PCI Intel EtherExpress Pro 100. Las medidas se han repetido, al menos, 6 veces, de modo que las gráficas muestran el intervalo de confianza al 95 % del valor medio de todas ellas.

La figura 3.8 permite evaluar el rendimiento del asistente al procesar paquetes pertenecientes a un flujo UDP, en función del tamaño de los datagramas.

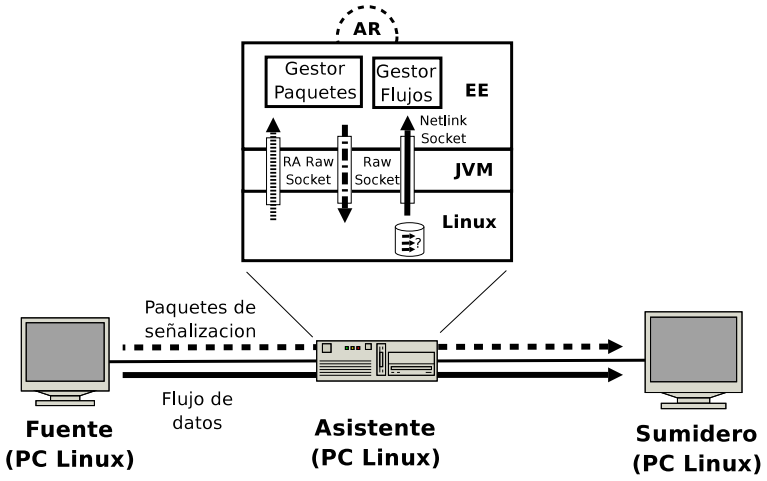


Figura 3.7: Topología de las pruebas sobre el asistente SARA

En realidad esta gráfica muestra el rendimiento máximo de la plataforma, puesto que la Aplicación de Red empleada tan sólo recibe los paquetes del flujo capturado y los reenvía al destino.

Como es de esperar, dado que los ejes representan la tasa de envío y recepción medida en paquetes por segundo (pps), el rendimiento máximo se consigue con los paquetes de menor tamaño (256 bytes), cayendo ligeramente según aumenta el tamaño de los paquetes. Sin embargo, el tamaño de los tramas no parece ser el factor limitante del sistema, puesto que, una vez saturado existen pequeñas diferencias de rendimiento, sobre todo si tenemos en cuenta que el salto de 512 a 1024 Bytes por paquete conlleva la duplicación del ancho de banda cursado.

Por tanto estas medidas sugieren que el rendimiento de la plataforma depende mucho más del número de cambios de contexto entre el *kernel* y el espacio de usuario (tramas recibidas) que del tamaño de los paquetes en sí, es decir, que del ancho de banda del flujo desviado.

La figura 3.9 permiter comparar la captura transparente de flujos, con el procesamiento de paquetes de señalización. En ambos casos la Aplicación de Red se limita a recibir y reenviar paquetes de 1024 Bytes sin realizar ningún otro tipo de procesamiento, por lo que, en igualdad de condiciones, el rendimiento en ambos casos es muy parecido.

Por último, la figura 3.10 permite estudiar la escalabilidad del mecanismo de clasificación de flujos empleado, ya que muestra el rendimiento del asistente al capturar paquetes de 1024 bytes, variando el número de reglas instaladas en

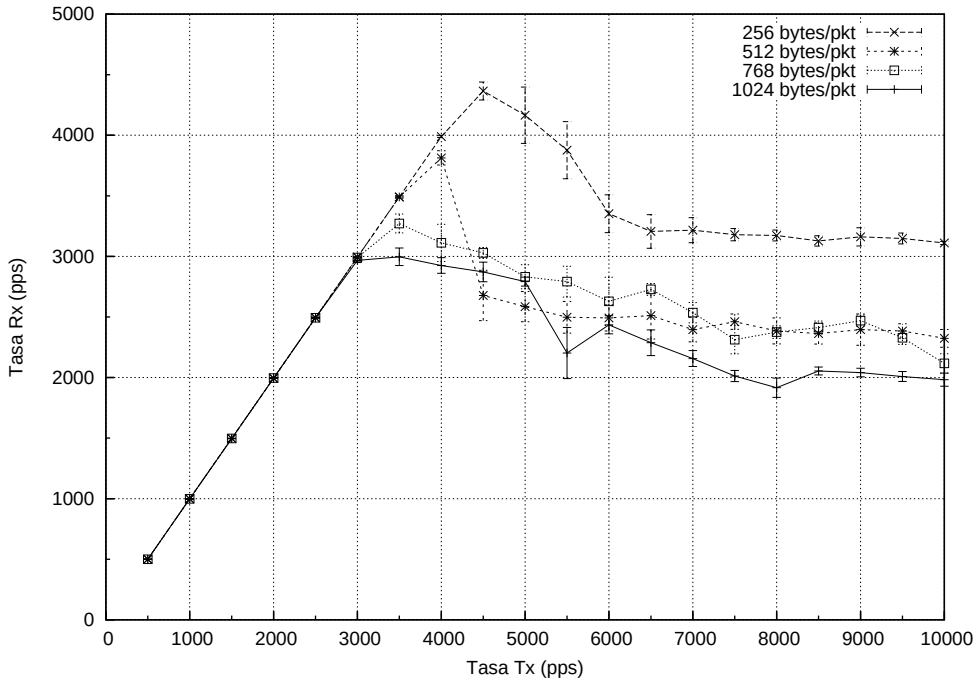


Figura 3.8: Rendimiento del procesamiento de flujos en función del tamaño de los paquetes

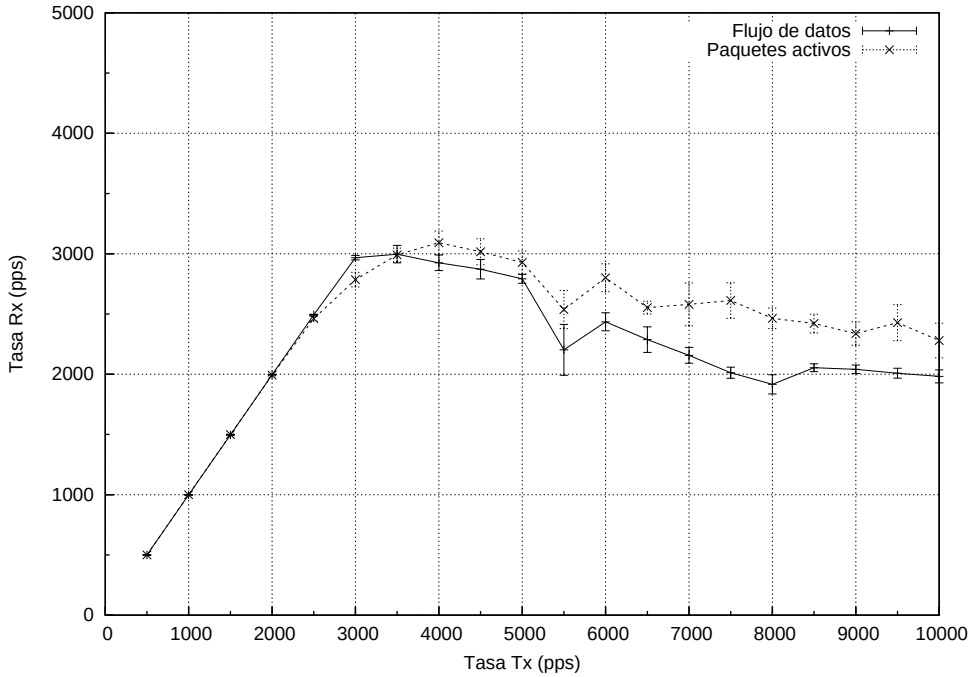


Figura 3.9: Rendimiento de los diferentes mecanismos de procesamiento de paquetes

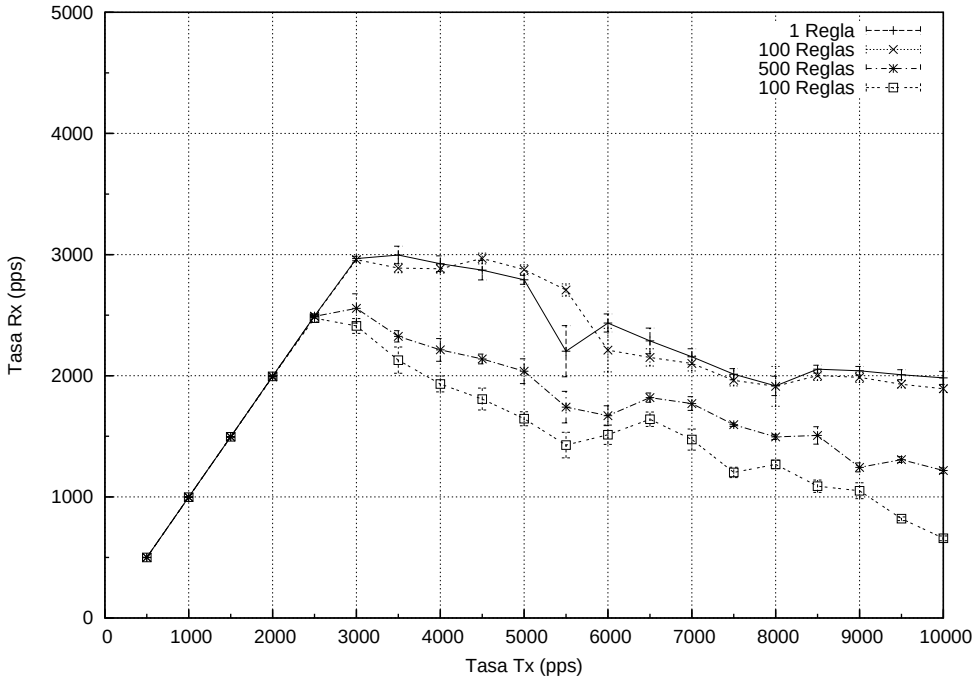


Figura 3.10: Rendimiento del procesamiento de flujos en función del número de reglas

el *kernel*.

Tal como puede observarse, la caída de rendimiento es insignificante entre 1 y 100 reglas, mientras que la utilización de reglas para diferenciar cientos o miles de flujos puede convertirse en un cuello de botella de la plataforma.

Por tanto, la utilización del desvío transparente y la captura de flujos debería limitarse a clases de tráfico (e.g. tráfico web), en lugar de intentar identificar flujos individuales, ya que emplear un gran número de reglas puede afectar gravemente al rendimiento del asistente.

Una solución a este problema consistiría en emplear varios asistentes, de forma que el *router* repartiese los flujos entre ellos. De este modo cada asistente podría definir reglas para procesar un conjunto reducido de flujos, con lo que el rendimiento global no se vería perjudicado.

Capítulo 4

Router-Assistant Protocol (RAP)

4.1. Requisitos de los Servicios de Red

Antes de diseñar el protocolo de control entre el *router* y los asistentes, es necesario analizar los requisitos de los servicios de red, que en última instancia serán los usuarios finales de dicho protocolo. Idealmente un nodo de red programable debería ser capaz de soportar cualquier protocolo o servicio de red, por tanto los requisitos deberían ser lo más amplios posible, sin centrarse en ningún tipo de aplicación en particular. Sin embargo, para ilustrar los requisitos identificados se utilizarán como ejemplo los siguientes servicios de red avanzados, que extienden las capacidades de los *routers* actuales:

- Extensión a IPv6. Empleando un asistente IPv6 sería posible añadir a un *router* IPv4, basado en *hardware*, capacidades de encaminamiento IPv6.
- *Multicast* fiable. El *router* se encargaría del encaminamiento de los datagramas *multicast* con datos, mientras que los asistentes procesarían los mensajes de señalización (e.g. ACK, NACK) de los clientes.
- NAT extensible. Delegando parte del procesamiento del NAT a un asistente, sería posible soportar nuevas aplicaciones o protocolos de transporte (e.g. SCTP).
- Exo-*firewall*. El *router* de acceso de un ISP podría extenderse con asistentes controlados por sus clientes para filtrar ataques de Denegación de

Servicio[1] antes de que saturen su enlace WAN, o en general para poder implementar políticas avanzadas de QoS en el enlace de entrada de la organización.

- Descubrimiento de *proxies* Web/SOCKS. Mediante un sencillo mecanismo de señalización que emplee paquetes SARA, los clientes podrían identificar qué *firewalls*/*proxies* hay en la red, y configurarse automáticamente para poder utilizarlos.

- Identificación de flujos basada en los datos de nivel de aplicación (e.g. para limitar el tráfico P2P). Cuando el *router* detecta un nuevo flujo podría enviar una copia del mismo al asistente hasta que fuese identificado, de modo que el asistente podría configurar en el *router* el filtro adecuado para ese flujo.

En general, basándose en el tipo de direccionamiento que pueden emplear los sistemas finales, los servicios de red pueden clasificarse en:

- **Transparentes**. Los sistemas finales desconocen qué nodos de la red están realizando el servicio empleado, o incluso que estén utilizando algún servicio de red, y por tanto envían los paquetes directamente al destino final (e.g. *firewall*, NAT, *multicast* fiable).

- **Con direccionamiento explícito**. En este tipo de servicios los sistemas finales son conscientes de la existencia de nodos de procesamiento intermedios, o por lo menos conocen al primero de ellos, ya que envían sus peticiones explícitamente a dicho nodo, que será el encargado de comunicarse con el destino final, u otros nodos de red intermedios (e.g. *proxy* Web/SOCKS, *redes superpuestas*).

Cada tipo de servicios necesita configurar de forma diferente el plano de datos del *router* al que está asociado: En el caso de los servicios de red transparentes es necesario desviar los flujos de datos adecuados para que sean procesados por los asistentes, y luego re-inyectar de nuevo ese tráfico en el *router* para que pueda alcanzar al destino final. Sin embargo, en el caso de los servicios de red con direccionamiento explícito basta con que los asistentes sean accesibles desde el exterior del nodo de red, y por tanto no es necesario aplicar ninguna medida especial.

[1] Denial of Service (DoS), en inglés.

Por otro lado, para localizar inicialmente a los nodos programables intermedios que realizan procesamiento dentro de la red, bastaría con soportar el mecanismo de señalización transparente definido en SARA, según el cual los sistemas finales pueden enviar mensajes de control directamente al destino que, al ser capturados por los nodos intermedios, informarán al cliente de su localización y de sus capacidades, para que este sea capaz de elegir al mejor de ellos.

Por tanto para soportar los dos tipos de transparencia (completa o sólo en la señalización inicial) es necesario que el *router* sea capaz de **desviar paquetes** determinados (e.g. marcados con la opción IP *Router Alert*) a los asistentes, y que posteriormente estos puedan **re-inyectar tráfico** en el plano de datos del *router*.

Analicemos cada una de estas funcionalidades por separado: En el caso del desvío de paquetes será necesario decidir qué paquetes deben ser desviados, y emplear un mecanismo para enviar al asistente tanto el paquete recibido como sus meta-datos asociados (e.g. la interfaz de entrada). Del mismo modo, dicho mecanismo también puede emplearse para que los asistentes re-inyecten los paquetes que han procesado en el mismo punto del plano de datos del *router* donde fueron desviados.

Para identificar a los paquetes que deben ser desviados, puede emplearse filtros basados en la información disponible en cualquiera de las cabeceras de protocolo del paquete, por lo que, en el caso general, pueden aplicarse a cualquier nivel de la torre de protocolos TCP/IP. Los filtros de desvío más importantes que han sido identificados en los *routers* actuales son:

- Nivel 1: Por interfaz física de entrada o salida. Este filtro permitiría a un ISP desviar todos los paquetes del interfaz WAN de un cliente a un asistente controlado por dicho cliente (e.g. exo-*firewall*).

- Nivel 2: Por identificador de tipo de protocolo en la trama. De esta forma sería posible ampliar un *router* IP con un asistente que encaminase otros protocolos de red (e.g. asistente IPv6).

- Nivel 3: Paquetes de señalización, por ejemplo los que contienen la opción *Router Alert* (e.g. descubrimiento de *proxies*), o según el identificador del protocolo de transporte (e.g. NAT extensible).

- Nivel 4[2]: Por flujo TCP/IP: <`dirección IP origen`, `dirección IP destino`, `protocolo`, `puerto origen`, `puerto destino`> o conjuntos de flujos. Por ejemplo desviar todo el tráfico web destinado al puerto TCP:80 a un asistente que implementa un *proxy* web transparente.

En cuanto al mecanismo para transportar los paquetes desviados del *router* al asistente y viceversa, podría emplearse el proceso de reenvío normal sobre tramas de nivel de enlace. Sin embargo algunos servicios de red requieren meta-datos adicionales, como el puerto físico o la interfaz por la que se ha recibido el paquete (e.g. *firewall*) o indicar explícitamente por qué interfaces debe enviarse el paquete (e.g. *multicast* fiable). Por tanto será necesario definir un mecanismo alternativo que permita **encapsular el paquete recibido junto con sus meta-datos asociados**.

Otro aspecto interesante del desvío de paquetes del *router* al asistente consiste en permitir dos semánticas de desvío: el **desvío con copia**, donde el paquete sigue procesándose normalmente en el *router*, pero una copia del mismo se envía al asistente, y el **desvío completo**, donde el paquete deja de ser procesado por el *router* para ser reenviado al asistente. La semántica de copia sería muy interesante para aplicaciones como los sistemas de detección de intrusiones[3], o para recoger estadísticas del tráfico. En ese caso sería recomendable desviar tan sólo una copia de los primeros 64 octetos del paquete, lo que permitiría observar las cabeceras más importantes del mismo sin comprometer la privacidad de los datos de los usuarios, y además reduciría [111] el consumo de ancho de banda en el enlace *router*-asistente.

Por último, una de las grandes ventajas de realizar procesamiento dentro de la red consiste en poder obtener inmediatamente información sobre el estado de la misma, y así actuar en consecuencia (e.g. reducir la calidad de un flujo de vídeo en caso de congestión en la red). Por tanto es necesario que el asistente pueda **acceder al estado detallado del router**. Tal como se comentó en el apartado 3.3.3, aunque SNMP puede servir para obtener una visión global del del *router*, sería deseable acceder con un mayor nivel de detalle a la información de estado y, por ejemplo, permitir a los servicios de red conocer el tamaño medio de la cola de una interfaz específica, u otra información que no está disponible

[2]Este tipo de filtros de desvío requiere cierto grado de soporte hardware por parte del router para la clasificación de flujos. Sin embargo, hoy en día la mayoría de los routers multiprocesador (e.g. Cisco 7500) disponen de esta característica, lo que les permite identificar cada flujo y así conocer el resultado de todas las ACLs que deberían aplicarse sobre sus paquetes con una sola búsqueda.

[3]Intrusion Detection System (IDS), en inglés.

en la MIB estándar de un *router*.

4.2. RAP y ForCES

Además de los requisitos enunciados en el apartado anterior, el protocolo de control *router*-asistente debe ser capaz de **descubrir las capacidades del *router*** en detalle para poder identificar qué funciones del servicio de red pueden procesarse directamente en el *router* (lo que será indudablemente más eficiente), y qué funciones no están soportadas por el *router* y deben delegarse en el asistente. Por ejemplo emplear el soporte *hardware* del *router* para hacer NAT sobre los flujos TCP/UDP, pero desviar los paquetes SCTP al asistente. Además, el protocolo debería permitir la configuración de las funcionalidades del *router* de las que depende el servicio de red desplegado, remotamente, desde el asistente.

Tal como se analizó en la sección 2.4.4, el protocolo *Forwarding and Control Element Separation* (ForCES), que está siendo desarrollado por el IETF, tiene exactamente esos requisitos: descubrir las características de los Bloques Funcionales (LFB) de los Elementos de Reenvío (FE) que forman el plano de datos de un *router*, y permitir la configuración remota de dichos bloques desde el Elemento de Control (CE).

Recordando la nomenclatura ForCES, un *router* ForCES multi-procesador está formado por varios Elementos de Reenvío (tarjetas de línea), controlados por un Elemento de Control (CPU central). Por otro lado, dado que en los asistentes se ejecuta tanto el plano de datos como el plano de control de los servicios de red desplegados, cada asistente podría considerarse como un Elemento de Red (NE) independiente y, como tal estaría formado por los Elementos de Reenvío (las Aplicaciones de Red que procesan paquetes) y un Elemento de Control (repartido entre la parte de control de las Aplicaciones y el Entorno de Ejecución que las gestiona).

Si intentásemos integrar SARA directamente dentro la arquitectura ForCES, habría que considerar que el nodo de red estaría formado por los FEs del *router* y las aplicaciones de red de los asistentes, mientras que el plano de control estaría distribuido entre el CE del *router* y los Entornos de Ejecución de los asistentes.

Sin embargo, aunque ForCES permite la existencia de múltiples CEs dentro de un nodo de red, dicha característica está orientada a arquitecturas de alta disponibilidad, con CEs redundantes, de forma que los FEs deben estar asociados

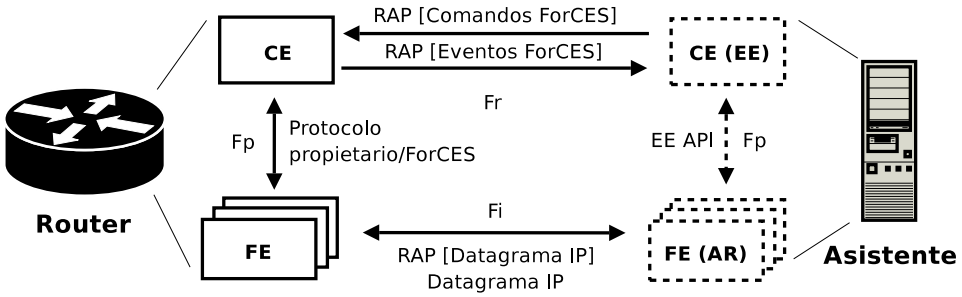


Figura 4.1: SARA dentro de la arquitectura ForCES

con ambos CEs, para que puedan pasar de uno a otro en caso de fallo. Por el contrario, en este caso cada CE controla grupos de FEs absolutamente disjuntos (FEs del *router* o ARs de un asistente), y no tiene sentido que los FEs de las tarjetas de línea del *router* sean conscientes de la existencia de los CE de los asistentes, estén asociados con ellos y, por ejemplo, les envíen paquetes de *heartbeat* periódicamente.

Por consiguiente RAP es un protocolo que permite que el *router* y cada asistente sean Elementos de Red totalmente independientes, pero que puedan cooperar entre sí, sin necesidad que los FEs del *router* estén asociados con los CEs de los Asistentes, y viceversa. Por el contrario, cuando un servicio de red desea configurar el plano de datos del *router*, por ejemplo para desviar el tráfico de una interfaz, el CE del asistente no intenta configurar los FE del *router* directamente, sino que emplea RAP para enviar dicha petición al CE del *router*, que será el encargado de aplicarla a sus FEs (figura 4.1).

Por tanto, el protocolo RAP formaría parte de la interfaz `Fr` de la arquitectura ForCES, definido entre el CE del *router* y el CE de un asistente. De ahí la necesidad de definir un nuevo protocolo, puesto que, tal como se explicó en el apartado 2.4.4, el grupo de trabajo ForCES sólo tiene como objetivo definir el protocolo CE-FE de la interfaz `Fp`.

Llegados a éste punto sería legítimo cuestionar la elección de la arquitectura ForCES para modelar un nodo de red SARA, puesto que si ForCES aún está en proceso de diseño, obviamente es imposible que un *router* legado pueda emplear dicha arquitectura. Sin embargo, no hay que olvidar que a día de hoy existe un gran número de plataformas comerciales basadas en arquitecturas distribuidas y que, por tanto, la existencia de protocolos de control propietarios para coordinar todos los elementos que las forman es un hecho. Por lo tanto el grupo de trabajo ForCES no intenta desarrollar un nuevo tipo de arquitec-

`divert ( filter:Filter, copy:boolean ): FlowHandle`
`PortFilter ( inIfs:int[] )`
`EtherFilter ( srcMAC:byte[6], dstMAC:byte[6], l3Proto:int )`

`IPv4Filter ( srcAddr:byte[4], srcMask:byte[4],`
`dstAddr:byte[4], dstMask:byte[4], l4Proto:int )`
`IPv6Filter ( srcAddr:byte[16], srcPrefix:int,`
`dstAddr:byte[16], dstPrefix:int, l4Proto:int )`

`TCPFilter ( srcAddr:byte[], srcPort:int,`
`dstAddr:byte[], dstPort:int )`
`UDPFilter ( srcAddr:byte[], srcPort:int,`
`dstAddr:byte[], dstPort:int )`

Cuadro 4.1: API de desvío de flujos del Entorno de Ejecución en UML

tura de *routers* multi-procesador, sino que ForCES define una arquitectura y unos protocolos estándar para permitir la inter-operabilidad entre elementos de diferentes fabricantes. En este sentido, ForCES proporciona a RAP un modelo de *router* estándar, independiente del fabricante, y que por tanto, evita definir un protocolo RAP diferente para cada arquitectura de *router*, lo cual es extremadamente útil, independientemente de si el *router* utiliza internamente los protocolos ForCES o no.

De este modo, para emplear un enfoque lo más general posible en el diseño de RAP, basaremos éste en los comandos del protocolo ForCES, de manera que el CE del *router* puede hacer de *proxy* de los comandos ForCES del asistente, bien reenviándoselos a sus FEs directamente si el *router* emplea ForCES, o bien traduciendo estos comandos estándar al protocolo de control propietario que soporte el *router* legado.

Sin embargo, obligar a que las Aplicaciones de Red sean capaces de obtener dinámicamente la topología de los FEs del *router*, y que a partir de la misma fuesen capaces de generar los comandos ForCES necesarios para configurarlo, complicaría extremadamente el desarrollo de servicios de red multi-plataforma, que en la mayoría de los casos requieren un nivel de abstracción del *router* muy superior. Por tanto, aunque la posibilidad de enviar comandos ForCES directamente puede ser interesante para algunas aplicaciones de red muy específicas, es preferible que el Entorno de Ejecución defina un API con las operaciones más usadas por los servicios de red, como la consulta del estado del *router* o los mecanismos de desvío o copia parcial de flujos, basados en los filtros definidos en el apartado anterior, y que sea el Entorno de Ejecución el que genere las peticiones ForCES adecuadas para los FEs y LFBs del *router* asociado. Por ejemplo,

la Tabla 4.1 muestra parte del API que permitiría a las Aplicaciones de Red desviar paquetes de datos (`divert()`) que coincidiesen con el filtro especificado (`TCPFilter()`).

Adicionalmente, además de las transacciones de configuración del tipo petición/respuesta, ForCES, y en general el resto de protocolos de control, también permiten que los FEs generen notificaciones asíncronas para informar al CE de eventos importantes, como por ejemplo la desconexión de un interfaz. Por tanto el CE del *router* también puede funcionar como un *proxy* en el sentido opuesto y enviar los eventos de sus FEs a los asistentes interesados en ellos.

En cuanto al desvío de flujos, no sería muy apropiado emplear este mecanismo de *proxy* a través del plano de control del *router*, sino que resulta mucho más eficiente que los FEs del *router* envíen los paquetes desviados directamente a los FEs de los asistentes, a través de la interfaz `Fi` definida en la arquitectura ForCES. Por tanto, la única peculiaridad que introduce SARA respecto al modelo ForCES habitual consiste en que el *router* SARA tiene dos tipos de puertos FE-FE, uno que conecta el FE con la *switching fabric*, y por tanto con el resto de los FEs del *router*, y otro que emplea alguna de sus interfaces externas para comunicarse con los FEs de los asistentes anexos.

Esto significa que RAP debe permitir, no sólo el transporte de los comandos de control ForCES, sino además la encapsulación de paquetes de datos y su meta-información asociada. Por tanto, RAP debe emplear un mecanismo de codificación que cumpla los siguientes requisitos:

- Debe ser extremadamente extensible para que pueda ampliarse fácilmente, por ejemplo para definir nuevos atributos de meta-información, incluir elementos opcionales, o para añadir más operaciones al protocolo.
- También tiene que ser muy eficiente en espacio, o de lo contrario, la encapsulación de los paquetes desviados junto con sus meta-datos asociados podría consumir demasiado ancho de banda del enlace *router*-asistente.
- Por último, no hay que olvidar que el rendimiento del plano de datos es esencial, y por tanto los mensajes RAP deben ser generados y procesados a altas velocidades.

Para lograr que RAP cumpla todas estas características es necesario emplear un mecanismo de codificación adecuado, que sea simultáneamente ligero,

extensible, fácil de procesar, y que permita encapsular datos binarios (i.e. los datagramas desviados) de manera eficiente.

4.3. *eXtensible Binary Encoding* (XBE32)

Hoy en día existe un gran número de mecanismos de codificación que pueden emplearse para definir protocolos de comunicaciones, sin embargo cada uno de ellos ha sido diseñado basándose en un conjunto de requisitos diferente, por lo que cada uno tiene unas características particulares, y está orientado a un dominio de aplicación diferente.

Por ejemplo, el *eXternal Data Representation* (XDR) [226], utilizado por RPC[4], tiene una sintaxis muy sencilla, similar a las estructuras de C, y además es muy eficiente y fácil de procesar ya que permite codificar datos binarios de una manera similar a como se almacenan en memoria. Sin embargo no es nada extensible, ya que es necesario conocer a priori la estructura de los mensajes para poder procesar su contenido.

Por el contrario, el *eXtensible Meta-Language* (XML) es el paradigma de la flexibilidad, por lo que se ha convertido una elección muy popular entre los diseñadores de aplicaciones y protocolos [113], ya que se han desarrollado un gran número de herramientas para trabajar con él. Sin embargo, XML puede ser extremadamente pesado [141] para el intercambio de datos binarios, ya que está orientado a la transmisión de información textual.

Abstract Syntax Notation One (ASN.1) es otro mecanismo de codificación flexible, pero está mucho más orientado a la definición de protocolos de comunicaciones binarios. En realidad ASN.1 [119, 120, 121, 122] es un complejo lenguaje para la definición de información, mientras que para la serialización de los datos definidos en ASN.1 pueden emplearse diferentes reglas de codificación[5] [123, 124], que son extremadamente extensibles, y permiten generar mensajes muy compactos. Sin embargo tanto la generación como el tratamiento de estos mensajes puede resultar bastante pesado, ya que están alineados a 8-bits y no emplean campos de longitud fija.

Por tanto, ninguno de los mecanismos de codificación analizados satisfacía por completo las necesidades de extensibilidad, rendimiento y eficiencia del pro-

[4]Remote Procedure Call, en inglés
[5]Encoding Rules, en inglés.

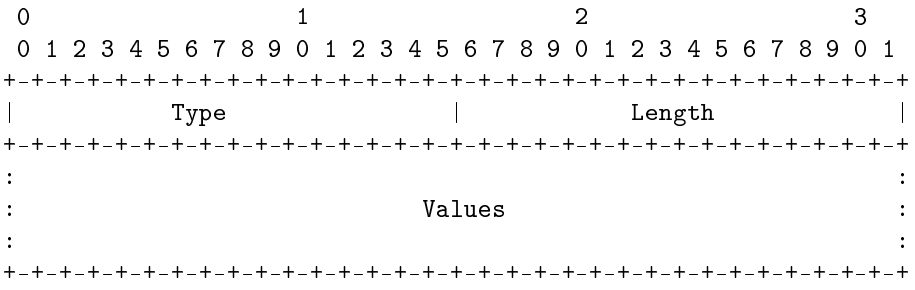


Figura 4.2: Formato de una TLV XBE32

tocolo RAP. Por esta razón se ha desarrollado el *eXtensible Binary Encoding* (XBE32), un mecanismo sencillo de codificación binaria basado en estructuras Tipo-Longitud-Valor[6] (TLV), que están alineadas a 32 bits para que puedan ser procesadas fácilmente (figura 4.2).

Un mensaje XBE32 está formado por Elementos que pueden identificarse por el Tipo de la TLV subyacente, mediante una cadena de texto, o con un identificador de 32 bits. Los Elementos XBE32 pueden ser de dos tipos: Atributos y Elementos Complejos. Un Atributo encapsula diferentes tipos de datos binarios (i.e. vectores de enteros, booleanos, números de coma flotante, cadenas de caracteres), mientras que un Elemento Complejo no contiene datos simples, sino Atributos u otros Elementos Complejos, lo que permite definir estructuras jerárquicas arbitrariamente grandes. Además, al estar orientado a la definición de protocolos, permite implementar fácilmente un procedimiento de control de versiones, ya que cada TLV indica qué operaciones debe realizarse cuando se encuentra un Elemento desconocido. Por ejemplo ignorarlo y seguir procesando el mensaje si el Elemento es opcional, o notificar al origen que no se ha comprendido su significado si el Elemento es obligatorio.

Este apartado sólo pretende ser una breve introducción del mecanismo de codificación XBE32; para más información puede consultar la especificación completa en [240].

Se ha optado por emplear XBE32 como mecanismo de codificación de los mensajes RAP, ya que es una solución de compromiso entre los tres requisitos definidos: es altamente extensible, muy compacto para la codificación de datos binarios, y tanto la generación como el procesamiento es muy eficiente en sistemas con arquitecturas de 32 bits.

[6] Type-Length-Value, en inglés

```
<error>
  <code>123456789</code>
  <desc>
    <text>Invalid Password</text>
    <lang>en</lang>
  </desc>
</error>
```

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x1000 (Complex)    | Length = 0x0000 (unspecified) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x2000  (Name Id)   |          Length = 12          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'e'      |      'r'      |      'r'      |      'o'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'r'      |      0x00     |      0x00     |     0x00      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x4001  (code)      |          Length = 8           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    0x07       |      0x5B     |      0xCD     |      0x15     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x0801  (desc)      |          Length = 32          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x2101  (text)      |          Length = 20          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'I'      |      'n'      |      'v'      |      'a'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'l'      |      'i'      |      'd'      |      ' '      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'P'      |      'a'      |      's'      |      's'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'w'      |      'o'      |      'r'      |      'd'      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 0x2102  (lang)      |          Length = 6           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'e'      |      'n'      |      0x00     |     0x00      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Type = 0x0000 (End-of-data)  |          Length = 4           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 4.3: Ejemplo de codificación XBE32

4.4. Especificación del protocolo RAP

El *Router-Assistant Protocol* (RAP) en realidad se descompone en dos protocolos dentro del marco de referencia ForCES: un protocolo de control entre CEs (interfaz `Fr`) y un protocolo de interconexión entre FEs (interfaz `Fi`). En ambos casos es necesario especificar un nuevo protocolo, puesto que el único interfaz considerado dentro del grupo de trabajo ForCES es el `Fp` (CE-FE), y por tanto el "protocolo ForCES" que está siendo estandarizado sólo tendrá ese ámbito de aplicación.

Por tanto, aunque formalmente RAP podría descomponerse en dos protocolos disjuntos, en la práctica tienen tantas características similares que es posible definir una estructura común para los mensajes. En particular todos los mensajes RAP están formados por un Elemento XBE32 Complejo, que a su vez contiene otros Elementos complejos: una cabecera común (`Header Element`) y una o más operaciones (`RAP Operation Element`), tal como se muestra en la figura 4.4.

Tanto en el caso de los mensajes de configuración CE-CE, como la transmisión de paquetes FE-FE es necesario identificar a los extremos de la comunicación, de modo que el Elemento XBE32 de cabecera contiene dos atributos (`source`, `destination`) que identifican al CE o FE que genera y al que está dirigido dicho mensaje (e.g. el CE del *router* o el FE de un asistente). Para ello se emplean los identificadores de 32 bits para los FE/CE definidos en el protocolo ForCES [73].

Por tanto, en RAP el *router* y los Entornos de Ejecución de los asistentes se diferencian por sus identificadores de CE, mientras que las Aplicaciones de Red de cada asistente emplearán identificadores de FE únicos, de modo que el EE lo pueda utilizar para entregar los paquetes desviados a la Aplicación de Red adecuada.

Además de estos atributos, la cabecera RAP contiene un tercer atributo (`xid`) que permite correlar peticiones y respuestas, ya que la respuesta a una petición RAP debe incluir el mismo `xid` especificado en la petición. De este modo es posible enviar múltiples peticiones RAP antes de recibir la respuesta anterior, lo que permite obtener un mayor rendimiento en la comunicación. Este atributo es opcional puesto que la transmisión de paquetes, y la notificación de eventos, son totalmente asíncronas y no tienen una semántica de petición-respuesta. Por tanto el atributo `xid` sólo aparecerá en los mensajes RAP de control enviados

```
  RAPv1 Message Element:
  Element name : rapv1
  XBE32 Type   : 0x0001

+-----------------------------------------------------------------+
:                          Header Element                         :
+-----------------------------------------------------------------+
:                     RAP Operation Element #1                    :
+.................................................................+
:                     RAP Operation Element #N                    :
+-----------------------------------------------------------------+


Header Element:
  Element Name : header
  XBE32 Type   : 0x0010

+-----------------------------------------------------------------+
|                            Source ID                            |
+-----------------------------------------------------------------+
|                          Destination ID                         |
+-----------------------------------------------------------------+
|                    [ Transaction Identifier ]                   |
+-----------------------------------------------------------------+

Source Identifier Attribute:
  Attribute Name : source
  XBE32 Type     : 0x4011
  Value Type     : opaque4
  Value Length   : 32 bits

Destination Identifier Attribute:
  Attribute Name : destination
  XBE32 Type     : 0x4012
  Value Type     : opaque4
  Value Length   : 32 bits

Transaction Identifier Attribute:
  Attribute Name : xid
  XBE32 Type     : 0x4013
  Value Type     : opaque4
  Value Length   : 32 bits
```

Figura 4.4: Formato de los mensajes RAP y de su cabecera

entre el CE del *router* y los CEs de los asistentes.

Además de por este atributo, los mensajes RAP de cada interfaz (`Fi`/`Fr`) se diferencian esencialmente por las operaciones que define cada uno de ellos.

4.4.1. Comandos de Control

Aunque en principio el CE del *router* podría actuar como un *proxy* transparente, y simplemente des-encapsularía el mensaje ForCES de la operación RAP, lo traduciría al protocolo de control interno, y la re-enviaría al FE especificado, el mecanismo de proxy propuesto permite al CE del *router* implementar políticas de seguridad y de virtualización sobre los mensajes ForCES enviados por los asistentes. Por ejemplo, permitiría asociar asistentes a FEs determinados, o limitar los comandos de configuración permitidos para cada LFB.

En el protocolo ForCES [73] se han propuesto dos familias de operaciones: las consultas, que sirven para obtener la configuración o la información de estado de los FEs o LFBs, y los comandos de configuración, que permiten aplicar operaciones básicas (`ADD`, `DEL`, `UPDATE`, `DEL_ALL`, `CANCEL`) para modificar el comportamiento de los FEs/LFBs del plano de datos del *router*. Por tanto se han definido cuatro operaciones RAP (dos peticiones y dos repuestas) para poder realizar todas las operaciones ForCES permitidas remotamente, desde un asistente externo.

Las cuatro operaciones comparten la misma estructura, ya que los campos definidos replican a los de la cabecera de los mensajes ForCES [73] (exceptuando `feId` que reemplaza a los campos `Source ID` y `Destination ID`, puesto que uno de ellos siempre será el `CE ID` del *router*, y el campo de `Flags` que sólo aparece en las peticiones). Por tanto, un asistente puede encapsular múltiples comandos ForCES en un único mensaje RAP, y el CE del *router* generará todos los mensajes de configuración a partir de las operaciones contenidas en el mensaje RAP que ha recibido y, en el caso de un *router* ForCES, los enviara a los FEs adecuados mediante la asociación establecida previamente con ellos.

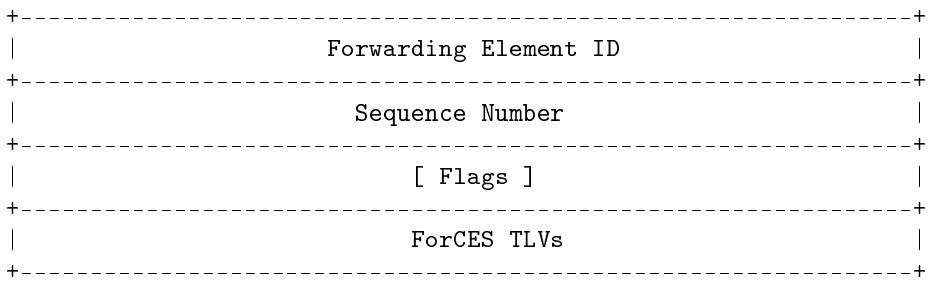
Además de este conjunto de operaciones, en ForCES hay varias familias de comandos adicionales pero no se han definido en RAP ya que sólo tienen sentido localmente, dentro del *router*, como las operaciones de asociación entre el CE y sus FEs, o los *heartbeats* periódicos. La figura 4.6 muestra un hipotético intercambio de mensajes RAP/ForCES, incluyendo el establecimiento de

```
ForCES Command Elements:
  Element Name : queryCommand
  XBE32 Type   : 0x0020

  Element Name : queryResponse
  XBE32 Type   : 0x0021

  Element Name : configCommand
  XBE32 Type   : 0x0030

  Element Name : configResponse
  XBE32 Type   : 0x0031
```




```
Forwarding Element ID:
  Attribute Name : feId
  XBE32 Type     : 0x4021
  Value Type     : opaque4
  Value Length   : 32 bits

SequenceNumber
  Attribute Name : seqNum
  XBE32 Type     : 0x4121
  Value Type     : int32
  Value Length   : 32 bits

Flags
  Attribute Name : flags
  XBE32 Type     : 0x4022
  Value Type     : opaque4
  Value Length   : 32 bits

ForCES TLV Attribute:
  Attribute Name : forcesTLV
  XBE32 Type     : 0x1010
  Value Type     : opaque
  Value Length   : variable
```

Figura 4.5: Formato de las operaciones RAP que encapsulan comandos ForCES

la asociación entre el CE y un FE del *router* ForCES, donde se intercalan los comandos RAP enviados por un asistente externo, con el envío periódico de *heartbeats*.

4.4.2. Subscripción a Eventos

Tal como se mencionó anteriormente, ForCES también estandariza un mecanismo para que los FE generen notificaciones asíncronas e informen al CE de determinados eventos. Por tanto la arquitectura de *proxies*-CE original puede ampliarse para incluir un mecanismo de notificación, que permita hacer llegar dichos eventos hasta los asistentes interesados en ellos.

En este caso, el CE del *router* también actúa como un *proxy*, pero en sentido inverso. Los servicios de red se subscriben a los eventos de los FEs/LFBs que controlan, y el CE del *router* será el encargado de enviar los eventos que reciba de sus FEs a todos los asistentes subscritos a ellos. Además, cuando el CE u otros asistentes realicen cambios significativos en la configuración de un FE/LFB, el CE también generará eventos para indicar a los asistentes afectados que la configuración del *router* ha cambiado.

La figura 4.8 muestra las operaciones para suscribirse y de-suscribirse a las notificaciones asíncronas generadas por el *router*, así como para recibir las notificaciones de eventos.

Las operaciones para suscribirse y de-suscribirse tienen exactamente el mismo formato que las operaciones de configuración y consulta detalladas en el apartado anterior, ya que en ForCES estas funciones están integradas dentro de las operaciones de configuración. Sin embargo en RAP se ha decidido diferenciarlas a nivel de familia de comandos, puesto que requieren un procesamiento especial por parte del CE, para que éste construya su propia lista de asistentes subscritos, a los que hacer llegar los eventos de los FEs.

En cuanto a la notificación de eventos, se ha optado por simplificar la cabecera del mensaje ForCES para reducir la sobrecarga en la generación de eventos, ya que sólo es necesario el campo `FE Identifier` para identificar a la fuente del evento, mientras que el LFB específico y el tipo de evento se especifican dentro de la TLV del mensaje ForCES.

Como protocolo de transporte para el intercambio de mensajes RAP de configuración, consulta, o suscripción enviados entre el *router* y un asistente, puede

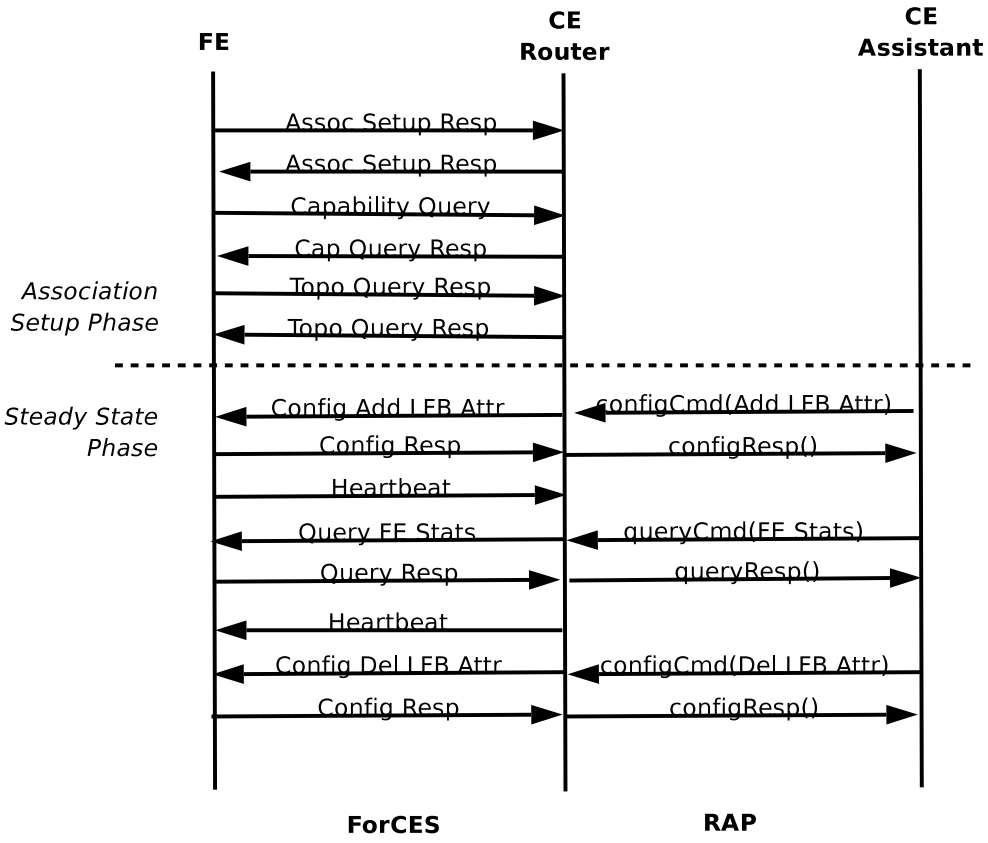


Figura 4.6: Intercambio de mensajes RAP/ForCES

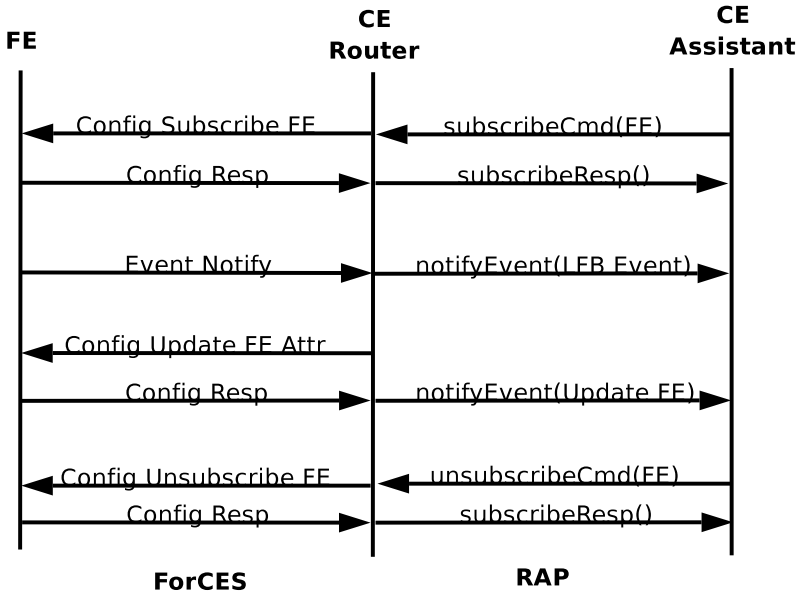


Figura 4.7: Eventos RAP/ForCES

```
Subscribe/Unsubscribe Command Elements:
  Element Name : subscribeCommand
  XBE32 Type   : 0x0040

  Element Name : subscribeResponse
  XBE32 Type   : 0x0041

  Element Name : unsubscribeCommand
  XBE32 Type   : 0x0042

  Element Name : unsubscribeResponse
  XBE32 Type   : 0x0043
```

```
+---------------------------------------------------------------+
|                      Forwarding Element ID                    |
+---------------------------------------------------------------+
|                        Sequence Number                        |
+---------------------------------------------------------------+
|                            [ Flags ]                          |
+---------------------------------------------------------------+
|                     ForCES Configuration TLVs                 |
+---------------------------------------------------------------+
```

```
Notification Event Element:
  Attribute Name : notifyEvent
  XBE32 Type     : 0x0050
```

```
+---------------------------------------------------------------+
:                      Forwarding Element ID                    :
+---------------------------------------------------------------+
:                  ForCES Event Notification TLVs               :
+---------------------------------------------------------------+
```

Figura 4.8: Formato de las operaciones RAP para (de)suscribirse, y notificar eventos

emplearse alguno de los protocolos TML [17, 144] definidos para el protocolo ForCES PL, o en general cualquier protocolo de transporte fiable como TCP o SCTP. Opcionalmente para el canal de notificación de eventos es preferible utilizar UDP, lo que permitiría el uso de direcciones *multicast* cuando varios asistentes estén interesados en los mismos eventos del *router*.

4.4.3. Envío de paquetes *router*-asistente

La transmisión de paquetes entre los FEs del *router* y del asistente requiere un formato de operación RAP (figura 4.9) más complejo que en el caso de los mensajes de configuración, puesto que además del contenido de la trama en sí, es necesario adjuntar la meta-información asociada a dicho paquete, como por ejemplo a qué flujo pertenece el datagrama.

Además, como el *router* puede ser capaz de desviar diferentes tipos de paquetes a los asistentes, es necesario emplear algún mecanismo para identificar que tipo de paquete contiene cada mensaje RAP. Por tanto también se ha definido un conjunto de atributos para transportar el contenido de los paquetes en sí, de forma que el campo `Type` de la TLV XBE32 identifica qué tipo de paquete está almacenado en el campo `Value`.

Los meta-datos asociados a un paquete son imprescindibles para algunos servicios de red, que necesitan conocer esa información para procesar el paquete, como por ejemplo un *firewall* necesita saber por qué interfaz del *router* fue recibido el paquete. Además, este mecanismo permite a los asistentes controlar la re-inyección de paquetes, por ejemplo para especificar las interfaces de salida de un datagrama *multicast*. Para ello es necesario definir una serie de atributos para encapsular dicha información, basándose en la librería de meta-datos definida en ForCES [264].

Para intentar minimizar el ancho de banda utilizado en el enlace *router*-asistente, podría ser adecuado enviar los mensajes RAP de datos directamente sobre el nivel de enlace empleado (e.g. Ethernet), o transportar dichos paquetes encapsulados en IP/UDP, lo que introduciría una sobrecarga de 28 bytes por paquete pero simplificaría el desarrollo del protocolo y permitiría que los asistentes estuviesen a varios saltos del *router*.

```
Header Element:
  Element Name : packet
  XBE32 Type   : 0x0080

+-----------------------------------------------------------------+
:                            Metadata                             :
+-----------------------------------------------------------------+
:                           Packet Data                           :
+-----------------------------------------------------------------+

Ethernet Frame Attribute:
  Attribute Name : etherFrame
  XBE32 Type     : 0x1082
  Value Type     : opaque
  Value Length   : variable

IPv4 Datagram Attribute:
  Attribute Name : ipv4Pkt
  XBE32 Type     : 0x1084
  Value Type     : opaque
  Value Length   : variable

...

Metadata Element:
  Element Name : metadata
  XBE32 Type   : 0x0081

+-----------------------------------------------------------------+
|                      Metadata Attribute #1                      |
+.................................................................+
|                      Metadata Attribute #N                      |
+-----------------------------------------------------------------+

Input Interface Metadata Attribute:
  Attribute Name : inIf
  XBE32 Type     : 0x4081
  Value Type     : opaque4
  Value Length   : 32 bits

Output Interface Metadata Attribute:
  Attribute Name : outIf
  XBE32 Type     : 0x4082
  Value Type     : opaque4
  Value Length   : variable

...
```

Figura 4.9: Operación RAP para el transporte de paquetes entre FEs y algunos Meta-datos definidos

4.5. Soporte Multi-Asistente y Reparto de Carga

La extensión de la arquitectura SARA a múltiples asistentes precisa un tratamiento especial dentro del protocolo RAP, puesto que es necesario considerar los escenarios dónde un mismo servicio de red ejecuta en múltiples asistentes, y por tanto varios asistentes pueden intentar configurar simultáneamente el *router* para especificar qué flujos deben desviarse a cada uno de ellos.

En este caso, se propone evitar, en la medida de lo posible, que varios asistentes configuren simultáneamente el plano de datos del *router*. Por tanto es preferible que un sólo asistente realice el grueso de las operaciones de configuración. A dicho asistente se le denomina "Asistente Designado" y se escogerá de entre todos los asistentes que soporten un servicio de red determinado, aquel que tenga el menor identificador de Entorno de Ejecución, esto es, el menor identificador de CE.

Una vez que el asistente designado haya configurado el plano de datos del *router* de acuerdo a las necesidades del servicio de red, el resto de los asistentes realizará los ajustes pertinentes a la configuración del reparto de carga inicial para que puedan recibir los paquetes desviados. A partir de ese momento los asistentes pueden coordinarse gracias al mecanismo de eventos definido para RAP, de forma que cualquier cambio en la configuración del *router* realizada por el CE del *router* o por cualquiera de los asistentes será notificada al resto.

Otro aspecto fundamental del *cluster* de asistentes es el reparto de carga. En ese sentido es necesario diferenciar entre los dos tipos de servicios de red, puesto que para cada uno de ellos el proceso de reparto de carga se realiza de formas muy diferentes entre sí. Los servicios transparentes dependen, permanentemente, del desvío de flujos, mientras que en los servicios con direccionamiento explícito la selección del mejor asistente ocurre puntualmente.

En el caso del desvío de flujos permanente, el *router* debe decidir a que asistente debe desviar cada paquete, a velocidad de línea, y por tanto el mecanismo de reparto de carga debe ser extremadamente eficiente. Por otro lado, en los servicios de red con direccionamiento explícito (e.g. *proxies*) la selección del mejor asistente se realiza una sola vez por sesión de datos, por lo que es preferible emplear un mecanismo de reparto de carga más complejo, pero que realmente seleccione al mejor asistente disponible. Es decir, en el primer caso el proceso de reparto de carga debe realizarse en el plano de datos del *router*, mientras que en segundo caso la selección del mejor asistente puede realizarse en el plano de

control, que es mucho más flexible.

4.5.1. Reparto de Carga en el Plano de Datos del *Router*

Dado que un servicio de red transparente necesita configurar el mecanismo de reparto de carga adecuado en el *router*, es razonable emplear el protocolo RAP para configurar su plano de datos, de forma que la selección del mejor asistente sea una etapa más del procesamiento de los paquetes desviados.

En este escenario el asistente designado realizará todas las operaciones de configuración necesarias (e.g. desviar los paquetes con direcciones privadas en un dispositivo NAT), incluyendo la instanciación de un LFB de Reparto de Carga que procese los paquetes que van a ser desviados a los asistentes. Dicho LFB será el encargado de elegir para cada paquete, a qué asistente debe desviarse, de acuerdo a la política de reparto de carga configurada.

Una vez que el plano de datos del *router* haya sido configurado correctamente, el resto de los asistentes que forman el *cluster* tan sólo tendrán que añadir sus entradas al LFB de Reparto de Carga, para que puedan comenzar a recibir la parte proporcional de los flujos asociados al servicio de red.

Dado que el rendimiento en el plano de datos es esencial, sólo será posible emplear políticas de reparto sencillas y escalables. En particular, y en función del servicio de red que ejecuta en el cluster de asistentes, será necesario definir dos tipos de mecanismos de selección: persistentes y no persistentes (i.e. sin estado).

En los servicios de red donde cada paquete puede tratarse independientemente de los demás (e.g. un *firewall* estático que filtra las conexiones TCP entrantes) se pueden emplear políticas de selección no persistentes, es decir cada paquete de un flujo podría enviarse a cualquiera de los asistentes del cluster. Para este tipo de aplicaciones, la política de selección más apropiada es, sin duda, *Weighted Round Robin* ya que realiza un reparto de carga óptimo y su sobrecarga al procesado de paquetes es mínima.

En el otro extremo de los requisitos se encuentran los servicios de red que necesitan almacenar estado por flujo (e.g. un NAT). En este caso, para que el cluster pueda escalar, es necesario que el estado esté distribuido entre todos los asistentes, de modo que los asistentes sean lo más independientes entre sí que sea posible. Por tanto el *router* debe ser capaz de desviar todos los paquetes

de un flujo determinado al asistente que almacena su estado. Además, el *router* debe ser capaz de realizar esta operación a velocidad de línea y soportar miles de flujos simultáneos.

Estos requisitos impiden que un *router* legado pueda utilizar una tabla de flujos que indique a qué asistente está asignado, puesto que implicaría realizar cambios en su arquitectura interna, y además, si no dispone de soporte *hardware*, podría limitar la escalabilidad de todo el sistema. En su lugar, el capítulo 5 de este trabajo propone una conjunto de técnicas alternativas, denominado *Fast Robust Hashing*, que permiten realizar esta asignación flujo-asistente a altas velocidades, e independientemente del número de flujos procesados y del tamaño del cluster de asistentes.

4.5.2. Reparto de Carga en el Plano de Control del *Router*

A diferencia de los servicios de red transparentes, cuando se emplea direccionamiento explícito, el *router* no tiene que reconocer y desviar flujos de datos, sino que el cliente final envía sus paquetes directamente a un asistente.

Por tanto, la selección de asistente se realiza por sesión de datos en lugar de por cada paquete, de modo que la política de reparto de carga se reduce a elegir al mejor asistente al inicio de la sesión. Dado que esa elección sólo se realiza una vez, pero puede tener un gran impacto el rendimiento del resto de la sesión de datos, es esencial que la selección del mejor asistente sea muy precias, lo que significa que la política de selección debe ajustarse perfectamente a las características del servicio de red distribuido. Así que, debido a la heterogeneidad de los servicios de red, es necesario disponer de un amplio abanico de algoritmos de selección, con diferentes grados de complejidad para poder elegir el más adecuado para cada tipo de servicio, a diferencia del caso anterior donde había un número de algoritmos muy reducido puesto que el rendimiento era el aspecto fundamental.

Dado que en este caso la flexibilidad es claramente más importante que el rendimiento, el proceso de selección puede implementarse en el propio plano de control del *router*. Es más, dado que los clientes son conscientes de la existencia de un *proxy* intermedio, sería incluso posible que la elección la realizase directamente el cliente final. De hecho, la única diferencia consiste en que, en el primer modo de funcionamiento el cliente sólo conocerá al asistente elegido por el *router*, mientras que en el segundo modo será el cliente el que seleccione el "mejor" asistente de todo el *cluster*, con la ventaja que supone que el cliente

conozca *proxies* alternativos en caso de que falle el seleccionado inicialmente. En cualquier caso, e independientemente del sistema que las aplique, es necesario realizar dos acciones: descubrir a los asistentes que soportan un determinado servicio, y elegir al mejor de ellos en función de su información de estado.

Curiosamente, la mayoría de la literatura sobre reparto de carga ha obviado el problema de la obtención del estado del *cluster* y se ha centrado en los algoritmos de selección. Sin embargo, en este caso, el proceso de descubrimiento de los asistentes es esencial desde el punto de vista de los clientes y nada trivial, si se desea evitar la pre-configuración de los clientes con la información sobre los asistentes, puesto que esta pre-condición complica extremadamente la gestión y el despliegue de nuevos servicios de red.

En su lugar se propone un mecanismo automático para localizar los nodos de red programable que se encuentran entre el cliente y el destino. Esencialmente consiste en enviar paquetes de señalización al destino, pero marcados con la opción IP de *Router Alert* para que sean capturados por los *routers*, solicitando un servicio de red determinado (e.g. un *proxy* SOCKS). De este modo, esta petición llegaría al plano de control del *router* y este podría responder al cliente indicando la información de todos sus asistentes, o bien eligiendo directamente a uno de ellos.

En cualquier caso es necesario algún mecanismo dinámico para que el *router*, o un sistema final, pueda conocer el estado real de los asistentes. Obviamente la primera alternativa de diseño consiste en añadir esta funcionalidad a RAP. Sin embargo hay que recordar que todas las operaciones de ForCES/RAP se encargan de gestionar el plano de datos del *router*, mientras que en este escenario el reparto de carga se realiza por completo en el plano de control. Por esta razón se ha optado por diseñar un mecanismo adicional, separado de RAP, que permite al *router* obtener información actualizada sobre el estado de los asistentes y de los servicios de red que soportan, incluyendo información específica sobre las políticas de reparto de carga más adecuadas para cada uno de ellos.

Aunque en principio sería posible diseñar un protocolo muy simple enfocado a este problema en particular, se optó por ampliar la investigación al problema de reparto de carga de servicios general, de forma que la idea de integrar las tareas de descubrimiento y reparto de carga en un único proceso pudiese extenderse a cualquier otro ámbito de aplicación. Esta línea de trabajo es otra de las aportaciones de la tesis, y ha tenido como resultado el diseño del *eXtensible Service Discovery Framework* (XSDF), un entorno completo de descubrimiento

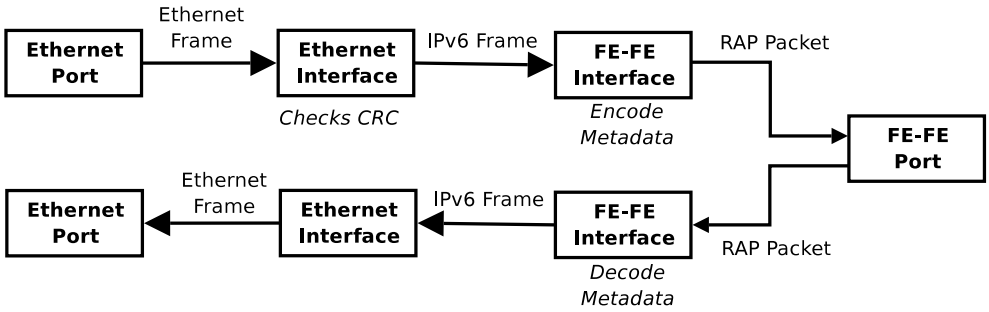


Figura 4.10: Asistente IPv6. Modelo ForCES de los FEs del *router*.

de servicios y reparto de carga, que será explicado en detalle en el capítulo 6.

4.6. Ejemplo de uso: Asistente IPv6

Para ilustrar el uso de RAP, y en general de la arquitectura SARA, van a analizarse tres servicios de red avanzados que podrían implementarse fácilmente en un nodo de red programable SARA. El primero de ellos consiste en dotar a un *router* IPv4 con la capacidad de encaminamiento IPv6 mediante el uso de un asistente[7].

A grandes rasgos, el *router* desviaría los datagramas IPv6 al asistente, y éste sería el encargado de realizar su encaminamiento, de forma que luego lo re-inyectaría en el *router*, indicándole la interfaz de salida. Además, el asistente se encargaría de ejecutar los protocolos de encaminamiento IPv6, así como la gestión de las direcciones de nivel de enlace mediante el mecanismo conocido como *Neighbor Discovery* [167].

Para desviar los datagramas IPv6, el asistente debería añadir una regla empleando el API del Entorno de Ejecución. Dicha regla permitiría desviar al FE del asistente todas las tramas de nivel 2 (e.g. Ethernet) con el identificador de nivel de red `0x86DD`, correspondiente a los datagramas IPv6. A partir de ese filtro el Entorno de Ejecución generaría los comandos ForCES para que los FE del *router* estableciesen el *data-path* mostrado en la figura 4.10 para las tramas IPv6, mientras que el resto de los datagramas IPv4 seguirían su procesamiento habitual.

Dichos comandos de configuración se enviarían al CE del *router* en opera-

[7]Este tipo de aplicaciones sería especialmente útil en *routers* legados de elevado coste, en los que se ha hecho una importante inversión en interfaces de red de diversas tecnologías

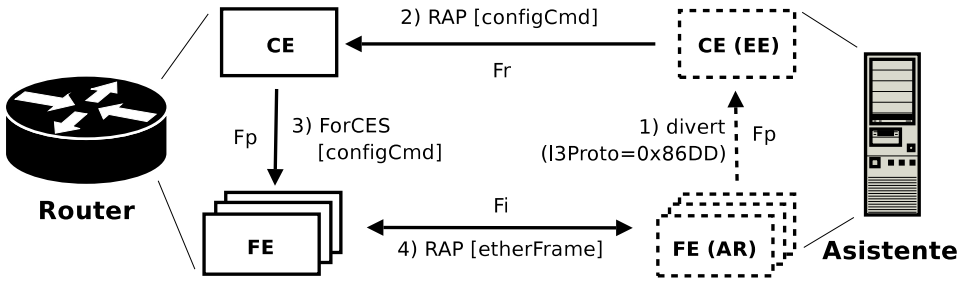


Figura 4.11: Asistente IPv6. Utilización del protocolo RAP

ciones `configCommand`, dentro de mensajes RAP. El CE del *router* al recibir los comandos del CE del asistente, comprobaría que son correctos, los traduciría en caso necesario, y los re-enviaría a sus FEs, tal como se muestra en la figura 4.11.

De esta forma todas las tramas que contuviesen datagramas IPv6 serían desviadas al asistente, dentro de mensajes RAP que incluirían los meta-datos asociados a las mismas, por lo que el asistente sería capaz de conocer, por ejemplo, la interfaz del *router* por donde se ha recibido dicho datagrama. Una vez que la aplicación IPv6 del asistente realizase el proceso de *route lookup*, el paquete se devolvería al *router*, encapsulado de nuevo en un mensaje RAP, y se especificaría el FE/Puerto de salida por el que debe ser enviada para llegar al siguiente salto.

En cuanto a los mensajes de los protocolos de encaminamiento IPv6 o de *Neighbor Discovery*, estos también emplean datagramas IPv6, por tanto, gracias a la regla añadida inicialmente, también serían desviados por el *router* al asistente. La aplicación de red IPv6 los procesaría y, en caso de ser necesario, enviaría los mensajes de encaminamiento o las peticiones/respuestas de *Neighbor Discovery* como datagramas IPv6 convencionales, de manera totalmente transparente para el *router* IPv4.

4.7. Ejemplo de uso: NAT Extensible y Escalable

El segundo ejemplo también consistiría en extender las capacidades de un *router* legado, pero en este caso el servicio de traducción de direcciones de red (NAT[8]). Típicamente, muchos *routers* de altas prestaciones implementan parte de este proceso en *hardware*, y soportan los protocolos de transporte más usuales (i.e. TCP, UDP) e incluso aplicaciones que necesitan un tratamiento específico

[8]Network Address Translation, en inglés

(e.g. FTP, H.323, SIP).

Sin embargo, el proceso de actualización para soportar nuevos aplicaciones o protocolos de transporte (e.g. SCTP) puede resultar extremadamente lento, sobre todo si tenemos en cuenta la velocidad a la que aparecen nuevas aplicaciones de red en Internet. Por esta razón se propone delegar parte del procesamiento del NAT al *cluster* de asistentes, de modo que el *router* legado con soporte *hardware* se encargue de los protocolos de transporte habituales, mientras que el procesamiento especializado de los datos de nivel de aplicación o de protocolos no soportados se produzca en los asistentes, basados en procesadores de propósito general.

Inicialmente, los asistentes emplearían RAP para descubrir las capacidades NAT del *router* legado y así conocer el subconjunto de funcionalidades no soportadas que deben proporcionar ellos (o el servicio de NAT completo si el *router* no lo soporta). Supongamos que en este caso se tiene un *router* de altas prestaciones que implementa en *hardware* un NAT para TCP, UDP e ICMP, pero se desea soportar también flujos SCTP mediante varios asistentes externos para suplir la carencia de *hardware* especializado.

Una vez identificadas las carencias del *router*, los asistentes ejecutarían la aplicación de NAT para SCTP, y el asistente delegado (aquel con menor identificador de CE) configuraría el *router* para desviar todos los flujos SCTP del interfaz WAN de salida (i.e. `divert(EtherFilter(l3Proto=0x84))`), de manera similar al ejemplo anterior. Sin embargo, al disponer de varios asistentes, sería necesario instanciar un LFB adicional para repartir los flujos SCTP desviados entre todos ellos. De este modo, una vez que el asistente delegado hubiese construido el *data-path* de la figura 4.12, todos los asistentes NAT añadirían su dirección al LFB de `Load Sharing` para que el *router* comenzase a entregarles los flujos SCTP desviados (aunque en este caso no sería necesario encapsular los datagramas en paquetes RAP, sino que bastaría con reenviarlos normalmente al asistente seleccionado).

Como el NAT es un servicio de red con estado que necesita persistencia, esto es que todos los paquetes de un flujo SCTP siempre vayan al mismo asistente, será necesario utilizar un algoritmo de *Fast Robust Hashing* que, tal como veremos en el capítulo 5, permite asignar flujos a los asistentes de manera permanente, a alta velocidad y sin limitación en el número de flujos, puesto que no emplea una tabla de flujos, sino que se basa en aplicar una o más operaciones *hash* sobre las cabeceras del paquete (e.g. las direcciones IP origen y destino)

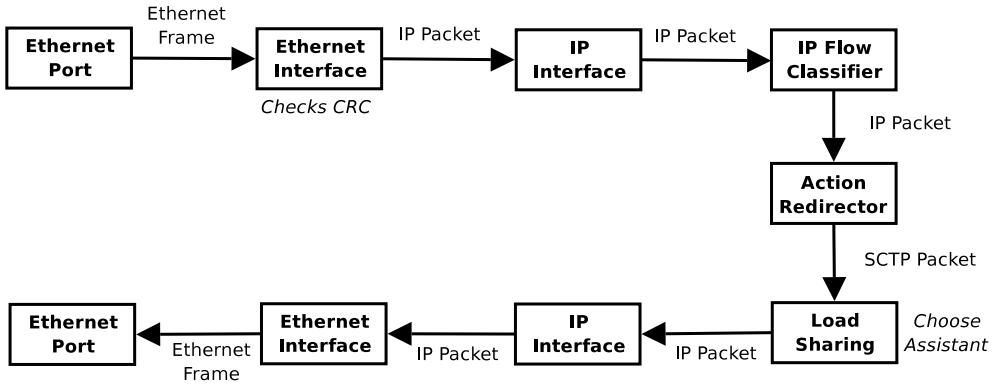


Figura 4.12: NAT Extensible. Modelo ForCES de los FEs del *router*.

para localizar al asistente encargado de procesar dicho flujo. Además, este mecanismo permite que la tabla de flujos SCTP se reparta equitativamente entre todos los asistentes del *cluster*, de modo que el servicio de NAT sería escalable con el número de asistentes del *cluster*.

De este modo, como todos los paquetes de un flujo SCTP siempre tienen las mismas direcciones origen y destino, los paquetes de cada flujo siempre serán desviados al mismo asistente. Éste consultaría su tabla de flujos SCTP para cambiar el puerto y la dirección origen privada (reemplazándola con su propia dirección IP pública), y lo reenviaría normalmente al destino.

Por tanto, los paquetes de respuesta llegarían directamente al asistente (sin que el *router* tuviese que desviarlos), que utilizaría el puerto destino para recuperar la dirección y puerto originales de su tabla de flujos SCTP, y así terminaría de traducir el paquete antes de reenviarlo normalmente al sistema final en la red privada.

4.8. Ejemplo de uso: *Multicast* fiable

Otro ejemplo muy interesante por sus requisitos de escalabilidad es el *multicast* fiable, que es uno de los mejores ejemplos de aplicaciones con procesamiento dentro de la red. Es conocido que para conseguir un servicio de *multicast* fiable capaz de escalar a miles de usuarios, es necesario limitar la cantidad de mensajes de señalización que se generan, y el número de retransmisiones, y por tanto es preferible realizar ese procesamiento inteligente en el interior de la red en lugar de extremo a extremo.

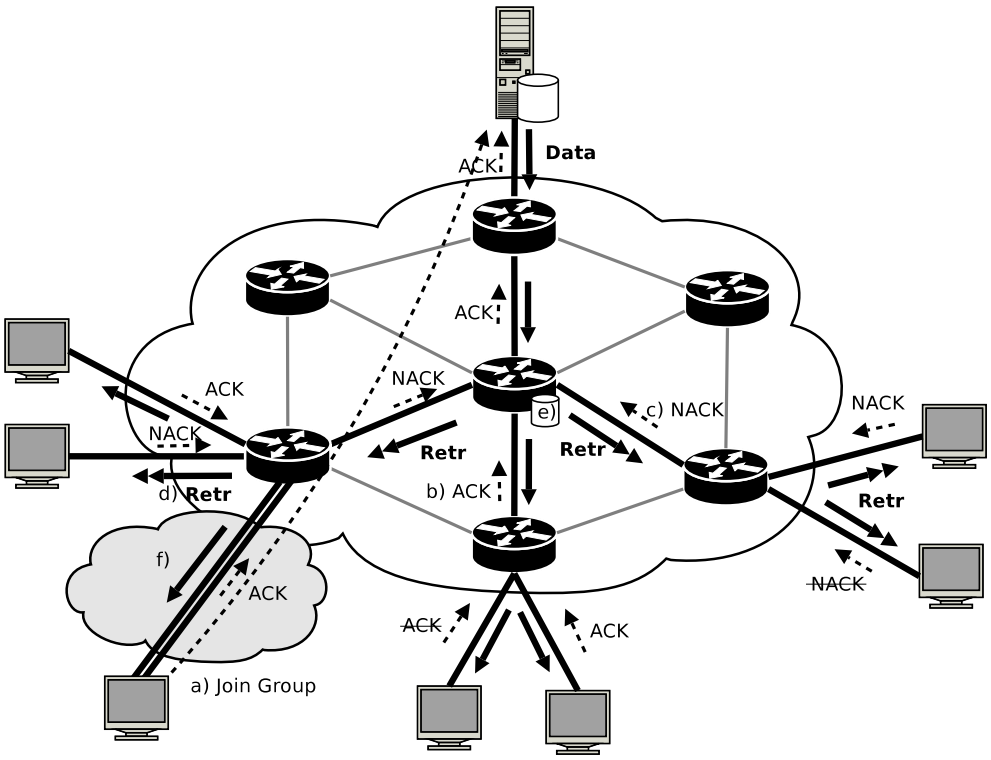


Figura 4.13: Mecanismos de *Multicast* Fiable: a) Gestión del árbol *multicast*, b) Agregación de ACKs, c) Filtrado de NACKs, d) Retransmisión selectiva, e) Retransmisión local, f) *Proxy multicast/unicast*

En la figura 4.13 se muestra un escenario de *multicast* fiable con una sola fuente y varios receptores, así como algunos mecanismos que emplean los protocolos de *multicast* fiable, como por ejemplo RMANP [34], para mejorar la escalabilidad del mismo:

- La construcción de un árbol *multicast* eficiente es una tarea esencial para poder aprovechar al máximo la transmisión *multicast*. Con la cooperación de los nodos programables intermedios, la complejidad de gestionar el árbol y los grupos de usuarios se reduce considerablemente.

- El mayor problema del *multicast* fiable es la implosión de los mensajes de confirmación (ACK o NACK) dirigidos a la fuente. Para evitar este efecto es necesario agregar la información proveniente de las ramas inferiores del árbol (Agregación de ACKs) y filtrar las peticiones de retransmisión

duplicadas (Filtrado de NACKs).

- En *multicast*, las retransmisiones reducen en gran medida el rendimiento ya que frenan la tasa de envío de la fuente, y aunque normalmente la perdida afecta a muy pocos usuarios del árbol, la retransmisión de la fuente llega de nuevo a todos los clientes. Una alternativa consiste en retransmitir selectivamente sólo por los enlaces donde se ha producido la pérdida, o realizar *buffering* en los nodos intermedios y retransmitir las tramas perdidas localmente, sin molestar a la fuente.

- Actualmente la mayoría de las organizaciones no disponen de una infraestructura *multicast*. Por tanto, para desplegar un servicio *multicast* fiable global sería necesario emplear *proxies* para atravesar redes *unicast*.

SARA es una plataforma ideal para desplegar un servicio de *multicast* fiable, puesto que permite implementar todos estos mecanismos eficientemente. Por ejemplo, gracias al procesamiento transparente de los paquetes de señalización podría construirse un árbol de distribución óptimo, que minimice los recursos de red empleados, incluso sin conocer a priori la topología de la red. Los clientes pueden enviar paquetes "activos" directamente a la raíz para asociarse al grupo *multicast*, de manera que los nodos programables intermedios que reciban paquetes de señalización por varias interfaces podrían cargar la aplicación de *multicast* fiable bajo demanda.

Además, este mecanismo no tendría problemas de escalabilidad gracias a la utilización del *cluster* de asistentes. La gestión de cada sesión de *multicast* fiable se puede asignar al asistente menos cargado de cada nodo de red programable que forma parte del árbol, de modo que, una vez construido el árbol, los mensajes de señalización de los clientes se enviarían de asistente a asistente hasta llegar a la fuente, y por tanto podrían filtrarse/agregarse fácilmente en los nodos de distribución. Además, aún empleando direccionamiento explícito, los paquetes de señalización podrían incluir la opción de *Router Alert*, para permitir que el árbol se adaptase a los cambios en la topología de la red dinámicamente, simplemente instanciando el servicio de red *multicast* en los nuevos nodos de distribución.

Así, al delegar toda la señalización en los asistentes, el rendimiento del *router* no se vería afectado, ya que este se limitaría a encaminar los datagramas *multicast* que contienen los datos generados por la fuente. Por el contrario, el asistente se encargaría de la señalización, y sólo procesaría paquetes de datos en

dos casos: en los nodos de red frontera con redes que no soportasen *multicast*, y para implementar retransmisión selectiva.

Para implementar un *proxy multicast/unicast*, el asistente tendría que desviar los datagramas dirigidos a la dirección *multicast* de la sesión de datos y encapsularlos en datagramas *unicast* para que puedan atravesar redes sin soporte *multicast*.

Finalmente, en los nodos con enlaces poco fiables (e.g. satélite) podría hacerse *buffering* de la sesión de datos para poder retransmitir localmente las tramas perdidas. Para ello, el asistente puede configurar al *router* para recibir una copia de los datagramas *multicast* (en lugar de desviarlos como en el caso anterior). De este modo, cada vez que recibiese NACKs, podría buscar el paquete perdido en su caché local y re-inyectarlo en el *router*, indicando las interfaces de salida para que fuese reenviado sólo por las ramas que lo hubieran solicitado.

Capítulo 5

Fast Robust Hashing

5.1. Introducción

Tal como se analizó en capítulos anteriores, hay ciertos servicios de red, como los *firewalls* o los NATs, que no pueden procesar los paquetes independientemente entre sí, sino que trabajan a nivel de flujo de paquetes.

Por lo tanto, en arquitecturas distribuidas como SARA donde un servicio de red puede ejecutarse simultáneamente en varias CPUs, de modo que cada CPU trata solamente una parte de los flujos desviados por el *router*, es necesario algún mecanismo de asignación flujo-asistente de forma que dado un paquete, el *router* pueda desviarlo al asistente encargado del flujo al que pertenece.

La política de asignación más sencilla es la estática, donde todos los paquetes recibidos por una interfaz (o grupo de interfaces) del *router* se asignan a un asistente determinado. Sin embargo este mecanismo no permite explotar toda la capacidad del *cluster* de asistentes, ya que puede ocurrir que algún asistente esté libre mientras que otros se encuentran saturadas por interfaces muy ocupadas.

Por tanto es razonable evitar este tipo de asignación interfaz-asistente estática y diseñar arquitecturas totalmente distribuidas, donde los paquetes de cualquier interfaz pudieran ser procesados por cualquier asistente disponible. Este tipo de arquitecturas serían menos costosas que las descritas anteriormente, puesto que los asistentes no tendrían que sobre-dimensionarse para soportar servicios a la velocidad de línea de la interfaz a la que estuviesen asignados, sino que la carga de cada interfaz podría distribuirse entre varios asistentes.

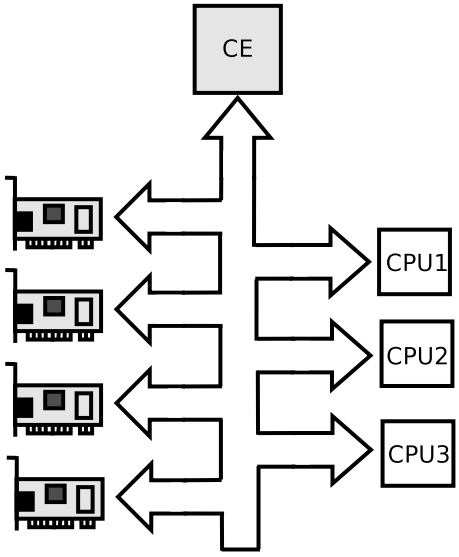


Figura 5.1: Arquitectura multi-procesador de *router* frontera

Sin embargo, esta necesidad de políticas dinámicas de asignación flujo-CPU no es exclusiva de la arquitectura SARA, sino que puede generalizarse a cualquier *router* multi-procesador que tenga procesadores de red especializados en la ejecución de servicios de red. En la figura 5.1 se muestra un diseño simplificado de esta arquitectura.

Por tanto, en el caso general puede considerarse que un nodo de red está formado por varias tarjetas de línea, que contienen las interfaces físicas, y que son capaces de realizar el procesamiento de nivel de enlace (e.g. *checksum*) a velocidad de línea. Por otro lado el *router* dispone de una granja de n CPUs (o asistentes en el caso de la arquitectura SARA), que ejecutan las rutinas de encaminamiento y el procesamiento de los niveles superiores a IP. Por último hay una CPU central de propósito general que ejecuta los demonios de encaminamiento, los protocolos de señalización y gestiona el resto de los componentes del *router* . Todos estos elementos están interconectados por un *backplane* ó *switching fabric* que permite transportar los paquetes desde las tarjetas de línea a las CPUs de servicios y viceversa, además de los comandos de configuración del Elemento de Control (CE) central (en [140] hay un estudio más amplio acerca de este tipo de arquitecturas).

Como esta arquitectura general engloba a SARA, pero además permite extender los resultados de este trabajo a otros tipos de arquitecturas de *router* distribuidas, se ha optado por emplear esta terminología general en el resto del capítulo. Sin embargo todos los razonamientos pueden aplicarse punto por pun-

to a la arquitectura SARA. Para ello basta con reemplazar el término "CPU de servicios" por "asistente".

La contención en el acceso a memoria es un gran problema en el diseño de *routers*, por ello sería muy deseable que cada CPU de encaminamiento o de servicios tuviese su propia área de memoria privada, del mismo modo que los asistentes SARA son servidores totalmente desacoplados, que no comparten información de estado entre ellos.

Sin embargo, esta decisión de diseño tiene una gran repercusión sobre los servicios de red que requieren almacenar estado por flujo, ya que esto implica que la información de un flujo determinado residirá en el área de memoria privada de una CPU y, por tanto, todos los paquetes de dicho flujo deberían ser procesados por esa CPU, o de otro modo el flujo podría ser tratado incorrectamente, o incluso interrumpirse. Por ejemplo un *firewall* con estado descartaría todos los paquetes externos si el registro sobre la sesión permitida estuviese en una CPU diferente a la que procesa el resto de los paquetes de dicho flujo.

Este requisito adicional de persistencia puede sugerir la utilización de alguna política de balanceo de carga para asignar flujos a CPUs y almacenar dicha asignación en una tabla de flujos, compartida por todos las tarjetas de línea. Sin embargo, diseñar una tabla con cientos de miles de entradas y compartida por múltiples procesadores que la consultan por cada paquete recibido es un gran desafío [260], y que sin duda requerirá soporte *hardware* por parte del *router*.

Por tanto, otro tipo de mecanismo rápido, escalable y distribuido para la asignación persistente de flujos a CPUs podría ser muy útil para el desarrollo de servicios de red con procesamiento por flujo, incluido el reparto de los flujos desviados entre los asistentes de la arquitectura SARA. De este modo el estado de los flujos podría estar distribuido entre todas las CPUs disponibles y se evitaría utilizar una tabla de flujos compartida en el *router*, que podría convertirse fácilmente en el cuello de botella de todo el sistema.

5.2. Técnicas de *Robust Hashing*

Un esquema de asignación basado en funciones *hash* parece ideal para arquitecturas de procesamiento por paquete, ya que no requiere una tabla de flujos compartida, sino tan sólo realizar una operación *hash* sobre el identificador de flujo[1] del paquete, que devuelve que CPU se encarga de todos los paquetes pertenecientes a ese flujo.

Sin embargo este esquema tan simple falla cuando el número de CPUs disponibles cambia, por ejemplo cuando una CPU se estropea. En ese caso el rango de los valores devueltos por la función *hash* debería reducirse en una unidad para evitar el uso de la CPU desactivada. Pero cambiar la función *hash* significa que las asignaciones previas de flujos, con CPUs obtenidas con la anterior función *hash*, serán diferentes a las que defina la nueva función [237] y, por tanto, hay una alta probabilidad de que el estado previo de los flujos resida en una CPU diferente a la seleccionada por la nueva función *hash*, lo que puede significar la pérdida de los flujos que se re-asignen.

La técnica de *Robust Hashing* [140] fue diseñada para evitar ese problema. Su objetivo consiste en que, siempre que una CPU se desactiva, solamente los flujos que estaban ligados a esa CPU se re-asignen equitativamente entre el resto de CPUs activas. La solución es bastante simple: en lugar de realizar una única operación *hash*, que devuelve la CPU escogida, cada CPU tiene una función de *hash* asociada, que indica la afinidad de la CPU a los flujos. En ese caso es necesario evaluar las funciones *hash* de todas las CPUs y se escoge la CPU que tenga la mayor[2] afinidad.

Cuando una CPU se desactiva, su función de *hash* deja de evaluarse (y por tanto no puede ser seleccionada). De esta forma sólo los flujos que estaban asociados con dicha CPU se re-asignan, puesto que, por definición, el resto de los flujos tendrán una afinidad mayor con alguna de las CPUs activas restantes.

El término *Robust Hash* fue acuñado por Ross en [206] dentro del dominio de las caches Web. Dada una URL (o el nombre del servidor Web) los clientes empleaban el algoritmo de *Robust Hash* para escoger entre los diferentes *proxies*-

[1] En este capítulo los términos flujo e identificador de flujo se emplean de manera general, puesto que las técnicas descritas son independientes del identificador de flujo empleado (e.g. en IP se suele emplear la tupla-5 <dir. IP origen, dir. IP destino, protocolo, puerto origen, puerto destino>)

[2] En realidad el algoritmo original seleccionaba el valor menor, pero cualquier otro tipo de elección determinista también sirve.

cache disponibles. De este modo cada objeto tan sólo reside en uno de los *proxies*, y así el tamaño efectivo de cache es la suma de todas las caches de los *proxies*, mientras que con otras políticas de reparto de carga, los objetos más populares tienden a replicarse en todas las caches. Posteriormente, el algoritmo de *Highest Random Weight* [237] de Thaler y Ravishankar amplió el mecanismo de *Robust Hashing* para soportar caches heterogéneas mediante la aplicación de un factor de peso a las funciones *hash* de cada *proxy*, y de esta forma variar el porcentaje de objetos asignados a cada uno de ellos.

Kencl y Le Boudec fueron los primeros en aplicar el mecanismo de *Robust Hashing* para asignar flujos a CPUs de *routers*. En [140] se analizan diferentes arquitecturas de *routers* multi-procesador, y describen un método para realizar balanceo de carga dinámico, ajustando los pesos asociados a cada CPU en función de la carga del último intervalo. Aunque esta técnica permite "ecualizar" la carga entre todas las CPUs disponibles, también provoca pequeñas re-asignaciones de los flujos debido al continuo ajuste en los pesos. En este sentido, los autores estudiaron los efectos de este mecanismo en el re-ordenamiento de paquetes (dos paquetes consecutivos del mismo flujos pueden reordenarse si se asignan a CPUs diferentes), y observaron que el porcentaje de paquetes re-ordenados era razonablemente pequeño.

Sin embargo, tal como hemos visto, incluso estas pequeñas re-asignaciones puede ser muy perjudiciales en el caso de realizar procesamiento de flujos con estado. Por tanto, en este tipo de servicios de red es preferible mantener asignaciones estáticas, aunque la carga no se reparta de una forma totalmente equitativa.

Una desventaja de todas estas técnicas es que para realizar cada asignación es necesario realizar una operación *hash* por cada CPU de servicios disponible. Aunque a primera vista la complejidad lineal de estos algoritmos no parece demasiado preocupante, hay que recordar que las funciones *hash* tienen cierta complejidad, y que las tarjetas de línea deben realizar todos este procesamiento por paquete, a velocidad de línea.

Este capítulo presenta dos nuevos algoritmos de asignación, diseñados para reducir el número de operaciones *hash* por paquete, pero manteniendo las propiedades de las técnicas "clásicas" de *Robust Hashing*: los algoritmos *Small Robust Hash* y *Big Robust Hash*.

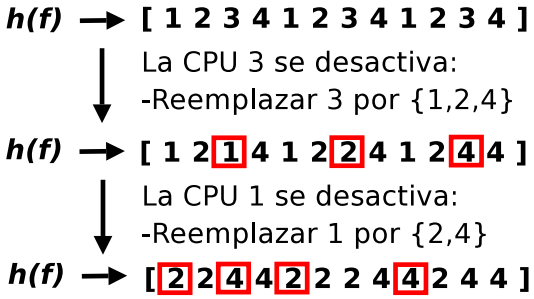


Figura 5.2: Evolución del Vector de Asignación en el algoritmo *Small Robust Hash* ($n = 4$, las CPUs 3 y 1 se desactivan)

5.3. *Small Robust Hash*

El objetivo del algoritmo *Small Robust Hash* es asignar un paquete a una CPU con una única operación *hash*. Para ello emplea una sola función de *hash* que no cambia, incluso cuando se desactiva alguna CPU, para poder mantener las asignaciones previas.

El algoritmo funciona de la siguiente manera: cuando llega un paquete, se obtiene su identificador de flujo f, y se emplea como argumento de la función de *hash* $h(f)$. Sin embargo, a diferencia de la técnica de *hashing* básica, $h(f)$ no devuelve el identificador de CPU en sí, sino una posición dentro del denominado "Vector de Asignación" de longitud fija, que contiene los identificadores de las CPUs disponibles.

Asumiendo que todos los índices devueltos por la función de *hash* son equiprobables para conseguir reparto de carga probabilístico [237], todas las CPUs activas deben tener el mismo número de entradas en el Vector de Asignación.

Cuando una CPU del *router* se desactiva, el CE central pide en paralelo a todas las tarjetas de línea que actualicen su copia local del Vector de Asignación. Este proceso de actualización terminará cuando todas las entradas de la CPU desactivada se sobre-escriban con los identificadores de las CPUs restantes empleando *round-robin*, tal como se muestra en la figura 5.2.

Por ejemplo, para el Vector de Asignación de la figura 5.2 y empleando $h(f) = f \mod 12$ como función de *hash*[3] sencilla, todos los paquetes con el identificador de flujo 17 serán asignados a la CPU 2, incluso cuando las CPUs 1 y 3 se desactivan. Por otro lado, los flujos con identificador 14 empezarían

[3]Esta función requiere que los índices del Vector de Asignación pertenezcan al rango $[0, 11]$.

siendo procesados por la CPU 3, pero luego se re-asignarían a la CPU 1 y luego a la CPU 4 cuando las CPUs 3 y 1 se desactiven.

5.3.1. Requisitos de Memoria

Para obtener un reparto de carga equitativo de flujos-CPUs, el algoritmo *Small Robust Hash* necesita un Vector de Asignación de longitud m tal que, dado cualquier número de CPUs desactivadas, siempre debe haber el mismo número de entradas para cada una de las CPUs restantes.

Este requisito es bastante exigente, puesto que, si n es el número total de CPUs, m debe ser múltiplo de n, $(n-1)$, $\ldots 2$. Es decir, de todos los posibles estados de CPUs activas. De otro modo, si m no es múltiplo del número de CPUs activas en ese momento, no será posible reescribir todas las entradas de la última CPU desactivada mediante un *round-robin* totalmente justo, lo que originaría cierto grado de desigualdad entre las CPUs activas.

$O(n!)$ puede ser una cota superior para la complejidad en memoria del Vector de Asignación, dado que $n!$, por definición, es múltiplo de todos los números en $[2, n]$. Sin embargo, normalmente el valor de m puede ser menor. Por ejemplo si m es múltiplo de 32, también lo será de 16, 8, 4 y 2. Por tanto m es el Mínimo Común Múltiplo de todos los números naturales entre 2 y n.

Una posible implementación, para inicializar el Vector de Asignación del algoritmo *Small Robust Hash*, consiste en calcular m realizando una *Criba de Eratóstenes* para encontrar el conjunto de los números primos $\mathcal{P} \subset [2, n]$. Después, para cada uno de ellos ($p_j \in \mathcal{P}$) se calculan sus potencias secuencialmente, y se escoge la mayor de ellas (i_{max}), tal que $p_j^{i_{max}} \leq n$, multiplicándolas hasta calcular m. Luego:

$$m = \prod_{\forall p_j \in \mathcal{P}} p_j^{i_{max}} \quad ; \quad i_{max} = max(i \in \mathbb{N} \mid p_j^i \leq n)$$

Número de CPUs (n)	**4**	**8**	**16**	**32**
Small Robust Hash (m)	12	840	$7 \cdot 10^5$	$14 \cdot 10^{12}$
Bits por posición	2	3	4	5
Tamaño (Bytes)	3	315	$3{,}6 \cdot 10^5$	$8{,}75 \cdot 10^{12}$

Cuadro 5.1: Requisitos de memoria del algoritmo de *Small Robust Hashing*

La primera fila de la tabla 5.1, muestra la longitud (m) del Vector de Asignación para diferentes valores de n. Tal como puede observarse, aunque $m \leq n!$, aun así es un valor enorme excepto para un número reducido de CPUs. Tal como hemos visto podría emplearse un valor de m mucho menor escogiendo sólo ciertos estados entre n y 2, pero en los casos no contemplados se produciría un reparto de flujos no equitativo.

5.3.2. Rendimiento

El algoritmo *Small Robust Hash* posee un rendimiento óptimo, dado que siempre realiza la asignación paquete-CPU con una única operación *hash*, y un acceso a memoria, independientemente del número de CPUs desactivadas. Además, cuando se desactiva una CPU, todas sus entradas en Vector de Asignación pueden reescribirse con complejidad lineal $O(m)$.

Sin embargo los requisitos de memoria del algoritmo restringen su uso a arquitecturas con un número reducido de CPUs, de ahí su nombre. Para arquitecturas con un gran número de CPUs podría emplearse el algoritmo de *Big Robust Hash* ya que, tal como veremos en el siguiente apartado, sus requisitos de memoria son mucho más modestos.

5.4. *Big Robust Hash*

Al igual que la técnica anterior, el algoritmo *Big Robust Hash* también emplea funciones *hash*, sobre el identificador de flujo del paquete, para conseguir una posición de memoria en el Vector de Asignación, que contiene el identificador de la CPU seleccionada. Sin embargo, este algoritmo no emplea una única función *hash* sino varias, cada una asociada a un Vector de Asignación diferente. Todos los Vectores de Asignación empleados forman la llamada "Matriz de Asignación".

Cuando todas las CPUs están disponibles, este algoritmo se comporta exactamente igual que el *Small Robust Hash*: solamente hay un Vector de Asignación (pero sólo tiene una entrada por CPU), y también existe una única función *hash* que devuelve posiciones dentro del vector. Las diferencias sólo aparecen cuando alguna CPU se inutiliza: el algoritmo de *Big Robust Hash* no emplea un único vector muy largo, sino varios.

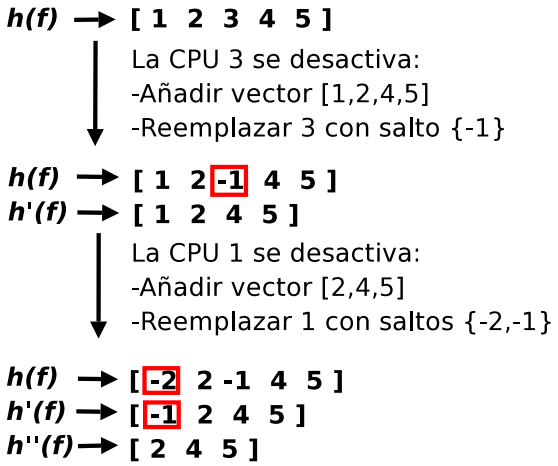


Figura 5.3: Evolución de la Matriz de Asignación del algoritmo de *Big Robust Hash* ($n = 5$, las CPUs 3 y 1 se desactivan)

Cuando el CE anuncia que una CPU se ha desactivado, se añade un nuevo Vector de Asignación, que contiene las posiciones de las CPUs restantes, y su función *hash* asociada. Después, la entrada de la CPU desactivada en el vector inicial se reemplaza con un puntero (denominado *hop* o "salto") al segundo vector, tal como se muestra en el primer paso de la figura 5.3.

Una vez actualizada la Matriz de Asignación, el proceso de asignación funciona de la siguiente manera: dado f, el identificador de flujo del paquete, la función *hash* inicial $h(f)$ devuelve una posición del primer Vector de Asignación. Si dicha entrada contiene una CPU activa, la asignación se ha completado. Pero si se encuentra un salto al segundo vector, es necesario aplicar otra función *hash* $h'(f)$ para elegir una posición en dicho vector. En el caso general, este proceso debe repetirse hasta encontrar una CPU disponible.

Aunque este mecanismo emplea varias funciones *hash*, no hay pérdida de flujos dado que siempre se aplican en el mismo orden y el proceso es determinista. Los flujos asignados a una CPU activa lo serán siempre por la misma función *hash* ya que su posición en el primer vector se mantiene sin cambios. Por otro lado, los flujos asignados a una CPU inactiva encontrarán la posición de salto en la primera operación *hash*, y la segunda función *hash* los asignará a una de las CPUs disponibles en el último vector. Por tanto, mientras que todas las entradas sean equiprobables las restantes CPUs absorberán la misma cantidad de flujos pertenecientes a la CPU inactiva.

Si una segunda CPU se desactiva, se añadirá otra función *hash* $h''(f)$ y su

Vector de Asignación asociado. Las entradas de la CPU inactiva en el resto de los vectores se renombrarán con posiciones de salto que apunten directamente al nuevo vector (ver segundo paso de la figura 5.3).

Nótese que el nuevo vector no reemplaza a uno anterior, sino que se añade al final de la Matriz de Asignación. Los vectores intermedios y sus funciones *hash* asociadas son necesarios para evitar la pérdida de los flujos que se asignaron al emplearlas, justo antes de que se desactivase la última CPU. Sin embargo, si varias CPUs se desactivan simultáneamente, los vectores intermedios no son necesarios, y es posible reescribir todas las entradas inactivas con saltos al nuevo vector, que sólo contiene entradas de las CPUs activas restantes.

Los siguientes ejemplos ilustran el algoritmo de *Big Robust Hash* basándose en las Matrices de Asignación mostradas en la figura 5.3, y una familia de funciones *hash* muy simples basadas en el operador modulo. $(h(f) = f \mod i) : h(f) = f \mod 5, h'(f) = f \mod 4, \ldots$. Por ejemplo, el flujo 9 se procesará siempre en la CPU 5 dado que, cuando se evalúa la primera función de *hash* $h(9)$, siempre encuentra la CPU 5. Por otro lado, una vez desactivadas las CPUs 3 y 1, los paquetes con el identificador de flujo 12 se asignan empleando las tres funciones *hash*. $h(12)$ encuentra un salto -1 al segundo Vector de Asignación, y aplicando $h'(12)$ sobre ese vector se obtiene otro salto al tercero. Por último $h''(12)$ asignará esos paquetes a la CPU 2. Sin embargo los paquetes del flujo 10 se clasificarán empleando solamente dos operaciones *hash*, porque el salto -2 encontrado en el primer Vector de Asignación redirige la búsqueda directamente al último, que siempre asigna directamente $(h''(10) \mapsto$ CPU 4).

5.4.1. Requisitos de memoria

Como la Matriz de Asignación se inicializa con un Vector de Asignación de longitud n, y se añade uno nuevo cada vez que se desactiva una CPU, el máximo número de Vectores de Asignación es $n + 1$. Entonces, dado que la Matriz de Asignación es triangular, tan sólo se necesita memoria para almacenar $\frac{n(n+1)}{2}$ entradas.

Número de CPUs (n)	**4**	**8**	**16**	**32**
Big Robust Hash ($\frac{n(n+1)}{2}$)	10	36	136	528
Bits por posición	3	4	5	6
Tamaño (Bytes)	4	18	85	396

Cuadro 5.2: Requisitos de memoria del algoritmo de *Big Robust Hashing*

La tabla 5.2 muestra, para diferentes valores de n, el tamaño máximo de la Matriz de Asignación que necesita el algoritmo de *Big Robust Hash*. Esos valores son lo suficientemente pequeños como para poder almacenar la Matriz de Asignación en la memoria SRAM de datos de cualquier procesador de red comercial [8, 118]. Por tanto, tal como estudiaremos en el apartado 5.6.3, la latencia de acceso a memoria no parece ser un problema importante, frente a la reducción en el número de operaciones *hash* que ofrece esta técnica.

5.4.2. Rendimiento

Si $k \in [0, n]$ es el número de CPUs inactivas, los algoritmos de *Robust Hash clásicos* evalúan $n - k$ funciones *hash* para realizar cada asignación paquete-CPU, esto es, una operación *hash* por cada CPU disponible. Sin embargo, esto significa que la mayoría del tiempo, cuando ninguna CPUs está desactivada, cada tarjeta de línea debe evaluar n funciones *hash* por paquete. Por el contrario, en el caso más habitual, cuando todas las CPUs están funcionando, los algoritmos de *Fast Robust Hash* presentados en este trabajo tan sólo necesitan una única operación *hash* y un acceso a memoria SRAM por paquete.

A diferencia de las técnicas de *Robust Hash* anteriores, donde todas las operaciones *hash* son independientes y pueden ejecutarse en paralelo, las operaciones del algoritmo *Big Robust Hash* deben realizarse consecutivamente, dado que el salto al Vector de Asignación que debe emplearse viene dado por la operación *hash* anterior. Sin embargo, la ejecución en paralelo requiere soporte *hardware*, mientras que los procesadores actuales para *routers* (e.g. procesadores de red) tienden a ser diseños multi-hilo, que permiten el procesamiento de paquetes mientras se accede a memoria, ya que intentan optimizar el proceso de *route lookup* basado en *Tries* [41], que también es un algoritmo serie con accesos a memoria consecutivos.

El estudio del rendimiento del algoritmo de *Big Robust Hash* es menos evidente que los anteriores dado que, a diferencia de las variantes *Small* y "clásicas", cada paquete no se asigna después de un número de operaciones *hash* fijo, sino que cada paquete puede necesitar un número de operaciones diferente.

Obviamente, el máximo número de operaciones *hash* por paquete es $k + 1$, puesto que ese es el número de Vectores de Asignación que hay con k CPUs inactivas. Incluso empleando este límite superior tan amplio, en el peor caso el algoritmo de *Big Robust Hash* supera al "clásico" (que debe realizar $n - k$

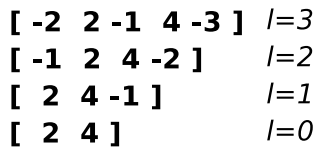


Figura 5.4: Ejemplo de Matriz de Asignación del algoritmo *Big Robust Hash* ($n = 5$, $k = 3$)

operaciones), a no ser que más de la mitad de las CPUs estén desactivadas.

Mediante otro análisis superficial se puede obtener un límite superior para el número medio de operaciones *hash*. Este proceso no puede tener complejidad lineal puesto que las posiciones de salto en la Matriz de Asignación sirven de "atajo" para encontrar una CPU disponible (cuanto más abajo se encuentre el Vector de Asignación, es más probable encontrar una CPU activa). Además, en media, los saltos desde cualquier Vector de Asignación se saltan la mitad de las filas de la Matriz de Asignación que hay entre ese vector y el último. Por tanto la complejidad debería ser, como máximo, logarítmica ($O(\log k)$).

La complejidad para actualizar la Matriz de Asignación cuando una CPU se desactiva es lineal $O(k)$ dado que tan sólo es necesario modificar una entrada en cada Vector de Asignación. Además, si las posiciones de las CPUs están ordenadas, ni siquiera es necesario buscar la CPU inactiva en cada vector, puesto que todas las entradas modificadas están, o en la misma columna que en el Vector superior, o en la columna anterior.

El resto de esta sección efectúa un estudio más minucioso, para poder obtener mejores límites del rendimiento del proceso de asignación. Para ello, cada vector de la Matriz de Asignación se ha denominado "nivel", y se ha numerado en sentido ascendente, desde el último vector al primero, tal como se muestra en la figura 5.4. De esta forma el número de cada nivel (l) también indica el número de CPUs inactivas/saltos que hay en ese nivel.

A partir de esta definición, estamos interesados en la variable de probabilidad $H(l)$, que define el número de operaciones *hash* que se necesitan para encontrar una CPU activa, si se empieza a buscar con la función *hash*/Vector de Asignación de nivel l. Otras funciones muy útiles para caracterizar un nivel l son:

$P_{CPUs}(l)$: Probabilidad de encontrar una CPU disponible en el nivel l:

$$P_{CPUs}(l) = \frac{n-k}{n-k+l} \tag{5.1}$$

$P_{hops}(l)$: Probabilidad de encontrar a alguno de los "saltos" en el nivel l:

$$P_{hops}(l) = 1 - P_{CPUs}(l) = \frac{l}{n-k+l} \tag{5.2}$$

Todos los saltos en un nivel son equiprobables (por la propiedad de balanceo de carga de las funciones *hash*). Por tanto, puede definirse $P_{hop}(l)$ como la probabilidad de encontrar un salto determinado en el nivel l:

$$P_{hop}(l) = \frac{P_{hops}(l)}{l} = \frac{1}{n-k+l} \tag{5.3}$$

Como vimos anteriormente, en el peor caso, una tarjeta de línea encontrará una CPU disponible tras evaluar las $k+1$ funciones *hash*. Sin embargo, eso significa que todas las operaciones *hash*, exceptuando la última, han encontrado un salto -1 al vector inmediatamente inferior. Así que, como tan sólo puede haber un salto -1 en cada nivel, la probabilidad de realizar $k+1$ operaciones *hash* es:

$$\begin{aligned} P(H(k) = k+1) &= P_{hop}(k) \cdot P_{hop}(k-1) \cdots P_{hop}(1) \\ &= \frac{1}{n} \cdot \frac{1}{n-1} \cdots \frac{1}{n-k+2} \cdot \frac{1}{n-k+1} = \frac{1}{\frac{n!}{(n-k)!}} = \frac{(n-k)!}{n!} \end{aligned} \tag{5.4}$$

Por tanto, cuando n es alto, el caso peor es muy poco probable. Así pues, estamos más interesados en $\bar{H}(k)$, esto es, el número medio de operaciones *hash* para encontrar una CPU activa, comenzando desde el vector superior de la Matriz de Asignación ($l = k$).

El nivel inferior de la Matriz de Asignación ($l = 0$) únicamente contiene CPUs activas, por tanto sólo se necesita una operación *hash*:

$$H(0) = 1 \Longrightarrow \bar{H}(0) = 1 \tag{5.5}$$

En los niveles superiores ($l > 0$), el número medio de operaciones *hash* es: sólo una si se encuentra una CPU ($P_{CPUs}(l)$), o una más el número medio de operaciones desde un nivel inferior ($\bar{H}(l-i)$) si se encuentra el salto a dicho nivel. En cada nivel hay l saltos equiprobables, *cada uno apuntando a un nivel*

inferior diferente (observe el nivel $l = 3$ de la figura 5.4), por tanto:

$$\begin{aligned}
\bar{H}(1) &= P_{CPUs}(1) \cdot 1 + P_{hops}(1)(1 + \bar{H}(0)) \\
\bar{H}(2) &= P_{CPUs}(2) \cdot 1 + \frac{P_{hops}(2)}{2}(1 + \bar{H}(1)) + \frac{P_{hops}(2)}{2}(1 + \bar{H}(0)) \\
\bar{H}(3) &= P_{CPUs}(3) \cdot 1 + \frac{P_{hops}(3)}{3}(1 + \bar{H}(2)) + \frac{P_{hops}(3)}{3}(1 + \bar{H}(1)) + \\
&\quad \frac{P_{hops}(3)}{3}(1 + \bar{H}(0)) \\
&\vdots
\end{aligned}$$

Es fácil ver que la formula general es:

$$\begin{aligned}
\bar{H}(l) &= P_{CPUs}(l) + \frac{P_{hops}(l)}{l}(l + \sum_{i=0}^{l-1} \bar{H}(i)) \\
&= \underbrace{P_{CPUs}(l) + P_{hops}(l)}_{1} + \frac{P_{hops}(l)}{l} \sum_{i=0}^{l-1} \bar{H}(i) = 1 + P_{hop}(l) \sum_{i=0}^{l-1} \bar{H}(i)
\end{aligned} \tag{5.6}$$

Esta formulación recursiva es bastante incómoda dado que $\bar{H}(l)$ depende de los $\bar{H}(i)$ de cada uno de los niveles inferiores. Sin embargo es fácil de simplificar, particularizándola para el nivel $l - 1$:

$$\begin{aligned}
&\bar{H}(l-1) = 1 + P_{hop}(l-1)(\bar{H}(0) + ... + \bar{H}(l-2)) \Rightarrow \\
&\bar{H}(0) + ... + \bar{H}(l-2) = \frac{\bar{H}(l-1) - 1}{P_{hop}(l-1)}
\end{aligned} \tag{5.7}$$

Extendiendo los elementos del sumatorio de la ecuación (5.6), y reemplazando la mayoría de ellos con la ecuación (5.7):

$$\begin{aligned}
\bar{H}(l) &= 1 + P_{hop}(l)(\bar{H}(0) + ... + \bar{H}(l-2) + \bar{H}(l-1)) \\
&= 1 + P_{hop}(l)(\frac{\bar{H}(l-1) - 1}{P_{hop}(l-1)} + \bar{H}(l-1)) \\
&= 1 + \frac{P_{hop}(l)}{P_{hop}(l-1)}(\bar{H}(l-1) - 1) + P_{hop}(l)\bar{H}(l-1)
\end{aligned} \tag{5.8}$$

Pero si tenemos en cuenta que:

$$\frac{P_{hop}(l)}{P_{hop}(l-1)} = \frac{\frac{1}{n-k+l}}{\frac{1}{n-k+l-1}} = 1 - \frac{1}{n-k+l} = 1 - P_{hop}(l) \tag{5.9}$$

De las ecuaciones (5.8) y (5.9) obtenemos:

$$\begin{aligned}\bar{H}(l) &= 1 + (1 - P_{hop}(l))(\bar{H}(l-1) - 1) + P_{hop}(l)\bar{H}(l-1) \\ &= 1 + \bar{H}(l-1) - 1 - P_{hop}(l)\bar{H}(l-1) + P_{hop}(l) + P_{hop}(l)\bar{H}(l-1) \\ &= \bar{H}(l-1) + P_{hop}(l)\end{aligned} \tag{5.10}$$

Extendiendo los elementos recursivos de la formula:

$$\begin{aligned}\bar{H}(l) &= \bar{H}(l-1) + P_{hop}(l) \\ &= (\bar{H}(l-2) + P_{hop}(l-1)) + P_{hop}(l) \\ &= (\bar{H}(l-3) + P_{hop}(l-2)) + P_{hop}(l-1) + P_{hop}(l) \\ &= (\bar{H}(0) + P_{hop}(1)) + P_{hop}(2) + \cdots + P_{hop}(l)\end{aligned}$$

Mediante (5.5), esta serie puede resumirse como:

$$\bar{H}(k) = 1 + \sum_{l=1}^{k} P_{hop}(l) = 1 + \sum_{l=1}^{k} \frac{1}{n-k+l} \tag{5.11}$$

Para obtener una expresión más compacta y asintótica en n, podemos reemplazar el sumatorio con una integral puesto que si $n \to \infty$ la diferencia entre los términos del sumatorio es pequeña ($\frac{1}{n-k+l} \to 0$), y por tanto el resultado puede aproximarse con una función continua:

$$\begin{aligned}\bar{H}(k) &= 1 + \int_1^{k+1} \frac{1}{n-k+l} dl = 1 + \ln(n-k+l)\Big|_1^{k+1} \\ &= 1 + \ln(n-k+k+1) - \ln(n-k+1)\end{aligned}$$

Y por último obtenemos una estimación del número medio de operaciones *hash* necesarias por paquete:

$$\bar{H}(k) = 1 + \ln\left(\frac{n+1}{n-k+1}\right) \tag{5.12}$$

Además del número medio, podemos intentar caracterizar mejor la distribución del número de operaciones *hash* por paquete, estudiando su función de probabilidad $P_{hash}(h,k)$:

$P_{hash}(h,l)$: Probabilidad de realizar h operaciones *hash* para encontrar una

CPU disponible, comenzando en el nivel l.

La única forma de realizar una única operación *hash* es encontrar alguna de las CPUs activas en el nivel actual, por tanto:

$$P_{hash}(1,l) = P_{CPUs}(l)$$

Para emplear 2 *hashes*, la primera consulta debe devolver uno de los l saltos de ese nivel, y la siguiente consulta en el nivel inferior $i \in [0, l-1]$ debe ser un éxito. Por tanto, la probabilidad total es la suma de todos los saltos posibles:

$$P_{hash}(2,l) = \sum_{i=0}^{l-1} P_{hop}(l) \cdot P_{hash}(1,i)$$

El razonamiento para 3 *hashes* es el mismo, aunque en este caso el salto al último vector no es posible dado que no tiene ningún salto y no sería posible realizar las 2 operaciones *hash* restantes. Por tanto, ahora $i \in [1, l-1]$:

$$P_{hash}(3,l) = \sum_{i=1}^{l-1} P_{hop}(l) \cdot P_{hash}(2,i)$$

De modo que, como todos los saltos son equiprobables, $P_{hop}(l)$ puede salir del sumatorio y la expresión general es:

$$P_{hash}(h,l) = \begin{cases} P_{CPUs}(l) & ; h = 1 \\ P_{hop}(l) \sum_{i=h-2}^{l-1} P_{hash}(h-1,i) & ; h \geq 2 \end{cases} \tag{5.13}$$

Para validar estos resultados analíticos, la sección 5.6 compara la aproximación al número medio de operaciones *hash* y la función de probabilidad de las operaciones *hash* en el algoritmo de *Big Robust Hash* con medidas sobre una simulación basada en una captura de tráfico real.

5.5. Añadir CPUs en los algoritmos de *Robust Hash*

Los algoritmos de *Fast Robust Hash* descritos en las secciones previas poseen las características deseadas para asignar flujos de paquetes a procesadores: son

rápidos, escalables con el número de CPUs (al menos en el caso del *Big Robust Hash*), pueden distribuirse fácilmente entre varias tarjetas de línea, y mantienen las propiedades de los algoritmos de *Robust Hashing* clásicos: reparten el número de flujos equitativamente entre todas las CPUs disponibles y las asignaciones no cambian mientras la CPU seleccionada siga funcionando. Sin embargo, imponen dos restricciones:

- Todas las tarjetas de línea deben tener una copia idéntica de los Vectores de Asignación para evitar inconsistencias en la asignación.
- Los algoritmos pueden funcionar indefinidamente desactivando y recuperando CPUs, siempre y cuando no se exceda el número inicial de CPUs y por tanto no sea necesario ampliar el Vector de Asignación.

Aunque estos requisitos pueden ser demasiado restrictivos para un algoritmo de *Robust Hashing* aplicado a un escenario de caches Web, que puede ser altamente dinámico, estas restricciones son bastante razonables para la arquitectura de *router* definida, donde el procesador central gestiona todas las tarjetas de línea, y el número máximo de CPUs estaría limitado por la capacidad física de la *switching fabric/backplane*. Sin embargo, el último requisito merece un estudio detallado para cada uno de los algoritmos analizados en este capítulo.

Los algoritmos de *Robust Hashing* clásicos [140] permiten añadir/recuperar CPUs, incluso superando el número total inicial, ya que para añadir una CPU nueva basta con evaluar su función *hash* de afinidad.

En el caso de los algoritmos de *Fast Robust Hash*, hay dos casos diferenciados: la recuperación de CPUs inactivas, y la ampliación del *cluster* de CPUs por encima de su capacidad inicial.

Las CPUs desactivadas pueden recuperarse invirtiendo la última actualización, es decir, reemplazando las últimas posiciones del Vector de Asignación reescritas (o saltos) con la nueva CPU. Sin embargo, no es necesario devolver a la CPU recuperada exactamente las mismas entradas que tenía inicialmente, ya que probablemente el estado de sus flujos asignados anteriormente se haya perdido, o simplemente esté obsoleto.

En el algoritmo de *Small Robust Hashing*, una CPU nueva tiene que "robar" aleatoriamente $\frac{m}{n-k} - \frac{m}{n-k+1}$ entradas a cada una de las CPUs activas. En el caso del algoritmo de *Big Robust Hash*, para deshacer la última actualización

basta con borrar el último Vector de Asignación, y reemplazar el mayor salto (en valor absoluto) del resto de los vectores con la nueva CPU.

Sin embargo a la hora de añadir más CPUs que el número planeado inicialmente, en ambos algoritmos es necesario reconstruir el Vector/Matriz de Asignación. Este problema puede aliviarse de cierto modo, sobre-estimando la Matriz de Asignación inicial, dado que sólo implicaría un ligera degradación en el rendimiento del algoritmo de *Big Robust Hash*, que necesita muy pocas operaciones *hash*, incluso cuando un gran número de CPUs están inactivas (ver ecuación (5.12)).

En cualquier caso, siempre que se recupera una CPU o se añade una nueva, todas las técnicas de *Robust Hashing* (incluidas las *clásicas*) re-asignarán $\frac{1}{n}$ flujos a la nueva CPU, debido a la capacidad de balanceo de carga de las funciones *hash*. Dado a que esta re-asignación puede provocar la pérdida de flujos, la adición de una nueva CPU debería emplear el siguiente mecanismo para transferir el estado de dichos flujos a la nueva CPU:

1. Cada CPU activa aplica el nuevo procedimiento de *Robust Hashing* a su tabla de flujos para comprobar qué flujos van a ser procesados por la nueva CPU.

2. El estado de esos flujos se transfiere a la nueva CPU, y todos los paquetes pertenecientes a dichos flujos se desvían a dicha CPU para ser procesados.

3. Comienza a aplicarse el nuevo procedimiento de Robust Hashing en las tarjetas de línea para asignar los paquetes directamente a la nueva CPU, en lugar de ser redirigidos por las CPUs activas.

4. Pasado un tiempo, cuando las colas de las CPU activas no tienen paquetes para la nueva CPU, se pueden borrar las entradas de los flujos desviados, y el proceso de actualización finaliza, puesto que a partir de ese momento todos los flujos serán procesados directamente por la CPU apropiada.

Este procedimiento puede aplicarse a cualquier técnica de *Robust Hashing*, incluyendo a las *clásicas*, y evita la pérdida de paquetes y flujos cuando se añade una nueva CPU. La otra alternativa pasa por planificar la activación de una nueva CPU cuando el *router* se encuentre desconectado de la red, junto al resto de las tareas de mantenimiento. En cualquier caso, el aspecto fundamental para la fiabilidad del *router* es recuperarse rápidamente ante el fallo de alguna de sus CPUs, puesto que estas situaciones no pueden predecirse con antelación.

CPUs Inactivas (k)	**0**	**1**	**2**	**3**
Asignación de Flujos	0.997903	0.998132	0.998826	0.999510
Asignación de Paquetes	0.991746	0.994732	0.996846	0.998336

Cuadro 5.3: Índice de Justicia medio del algoritmo de *Small Robust Hash* (n=5, t=50ms)

5.6. Comparativa de las técnicas de *Robust Hashing*

Este apartado realizará una comparativa de los diferentes algoritmos de *Robust Hashing* utilizando, tanto resultados analíticos como simulaciones basadas en trazas de tráfico real. En particular, las simulaciones de los algoritmos de *Fast Robust Hash* se basan en una captura de paquetes de un enlace *Packet-over-SONET* OC-12c de la Universidad de Florida en Gainesville [180]. Dicha traza dura 90 segundos y contiene 7.7 millones de paquetes y 270 mil flujos diferenciados, con una tasa binaria media de 473 Mbps y 86194 paquetes por segundo de media.

El identificador de flujo de 32 bits se ha calculado empleando el operador *XOR* sobre las direcciones IP origen y destino, y los números de puerto, con el puerto mayor en los primeros 16 bits y el menor en los 16 bits restantes. Posteriormente se aplica una función *hash* de Fibonacci [146], con el módulo apropiado para acceder al Vector de Asignación en uso.

5.6.1. Asignación Equitativa

Aunque, tanto el identificador de flujo, como la función *hash* empleada son bastante sencillos, y la traza ha sido anonimizada[4], la asignación de flujos parece estar bien distribuida entre todas las CPUs disponibles.

La tabla 5.3 muestra el Índice de Justicia medio ($f(x) = \frac{[\sum x_i]^2}{n \sum x_i^2} \in [0,1]$) [127] de la asignación de flujos y de paquetes de la traza entre 5 CPUs, empleando el algoritmo de *Small Robust Hash*, y medido en intervalos de 50 ms. El algoritmo de *Big Robust Hash* tiene unos valores similares, por lo que la descripción de la misma es aplicable a ambas técnicas de *Fast Robust Hashing*.

Puede observarse que los flujos están asignados de manera equitativa, incluso cuando se desactivan algunas CPUs, mientras que la justicia en la asignación de

[4]Cada dirección IP nueva encontrada fue renombrada secuencialmente, empezando desde 10.0.0.1

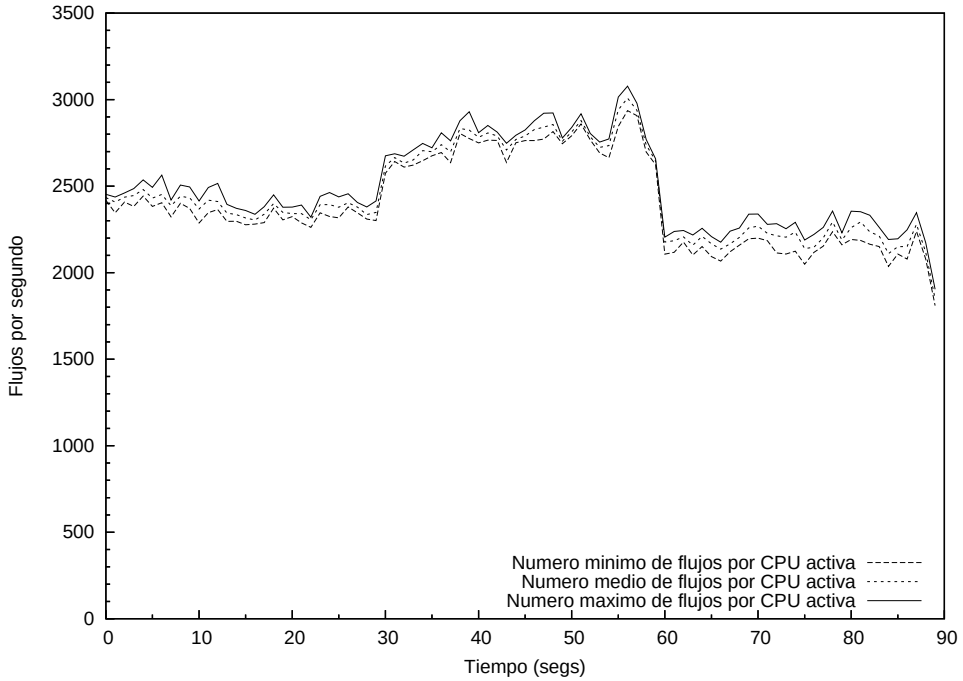


Figura 5.5: Asignación de flujos del algoritmo *Small Robust Hash* (n=5)

paquetes es un poco inferior. Este comportamiento se debe a que la distribución de los paquetes por flujo en Internet tiene una cola amplia[5] [189]. Esto es, aunque la mayoría de los flujos están formados por un pequeño número paquetes (< 20), la gran mayoría de los paquetes pertenecen a un reducido número de flujos de larga duración. Por tanto la asignación de esos flujos grandes entre CPUs puede acarrear cierto grado de reparto no equitativo, en cualquier tipo de técnica para la asignación de flujos a CPUs.

La figura 5.5 muestra, según la línea temporal de la traza, la asignación de flujos del algoritmo *Small Robust Hash*, para poder visualizar el impacto de la pérdida y la recuperación de CPUs. Inicialmente había 5 CPUs activas, pero 30 segundos más tarde una de las CPUs se ha desactivado con lo que se han perdido 1575 flujos activos en ese momento, aunque las 4 CPUs restantes han sido capaces de seguir procesando los nuevos flujos que llegan, aumentando su carga. A los 60 segundos la CPU inactiva se recupera, pero sin pérdida de flujos, gracias al mecanismo de recuperación presentado en el apartado 5.5. A partir de ese momento los flujos vuelven a asignarse equitativamente entre las 5 CPUs, disminuyendo su carga individual. Sin embargo no alcanzan los valores iniciales debido a la pérdida de flujos inicial debido a la perdida de estado de la CPU

[5] Heavy Tailed, en inglés

desactivada.

5.6.2. Rendimiento de las técnicas de *Robust Hashing*

La figura 5.6 compara el rendimiento de todos los algoritmos de *Robust Hashing* analizados en este capítulo: El *Robust Hash* clásico realiza $n-k$ operaciones *hash* por paquete. Por tanto no puede superar al algoritmo de *Big Robust Hash* a no ser que la gran mayoría de las CPUs estén inactivas. Por supuesto, el mejor rendimiento es el del algoritmo de *Small Robust Hash*, que necesita una única operación *hash* por paquete, independientemente del número de CPUs desactivadas. Sin embargo este resultado es puramente teórico, ya que los requisitos de memoria para 32 CPUs están más allá de cualquier implementación práctica.

El algoritmo de *Big Robust Hash* se encuentra en un terreno intermedio entre ambos, ya que sacrifica rendimiento por memoria (528 Bytes para 32 CPUs[6]). La figura 5.6 muestra la curva teórica (ecuación (5.12)) del número medio de operaciones *hash*, y el límite superior de $k+1$ operaciones, descrito en el apartado 5.4.2.

En cuanto a los resultados de la simulación del algoritmo de *Big Robust Hash*, se muestra el mínimo, primer cuartil, la media, segundo cuartil, y el número máximo de operaciones *hash* empleadas para la asignación de todos los paquetes de la traza. Puede observarse que solamente cuando hay muy pocas CPUs inactivas, algunos paquetes necesitan $k+1$ operaciones *hash* (el máximo teórico). Después de esa situación inicial, los valores máximos se quedan muy por debajo de la línea teórica, dado que la probabilidad del caso peor se hace cada vez más pequeña (ver ecuación (5.4)).

Tal como se esperaba, el número medio de operaciones aumenta incluso más lentamente que los valores máximos (e.g. con 31 Vectores de Asignación, es posible encontrar una de las 2 CPUs restantes tan sólo con 3.67 operaciones *hash* de media). Este comportamiento puede observarse mejor en la figura 5.7 que muestra, para valores representativos de k, las funciones de densidad del número de operaciones *hash* empleadas por el algoritmo de *Big Robust Hash*.

Además, la Tabla 5.4 muestra las probabilidades acumuladas de realizar h o menos operaciones *hash* por paquete, obtenidos a partir de la Ecuación (5.13) para diferentes valores de n y k. Por ejemplo, la sexta fila ($n = 32, k = 24$)

[6]En la figura 5.6 el eje X va de 0 a 30 CPUs inactivas, ya que las situaciones en las que todas las CPUs están desactivadas, o queda una única CPU no tienen demasiado sentido.

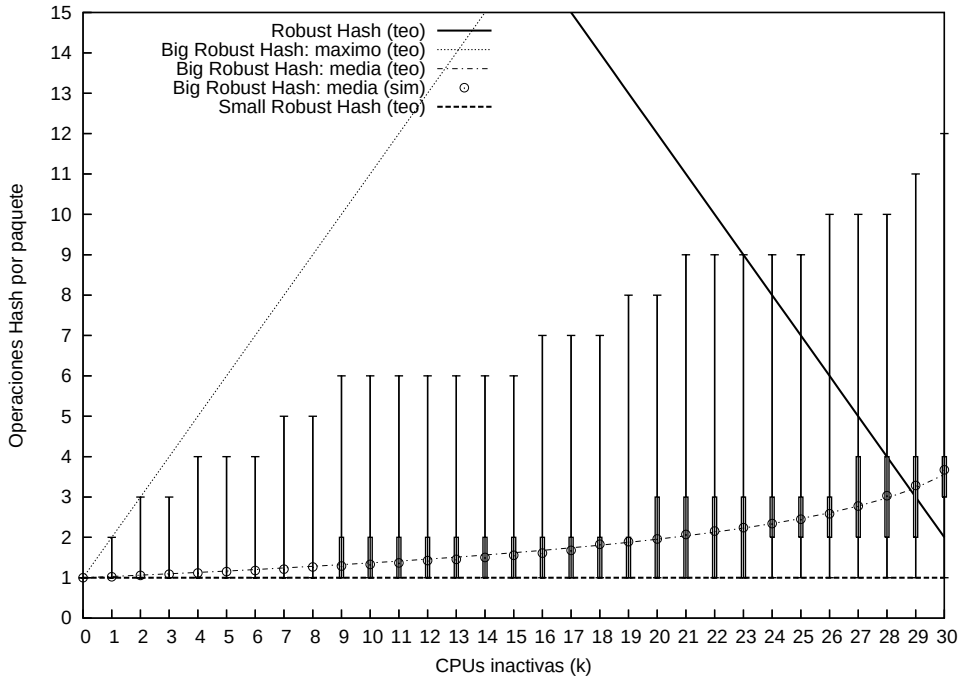


Figura 5.6: Comparación entre diferentes algoritmos de *Robust Hash* ($n = 32$)

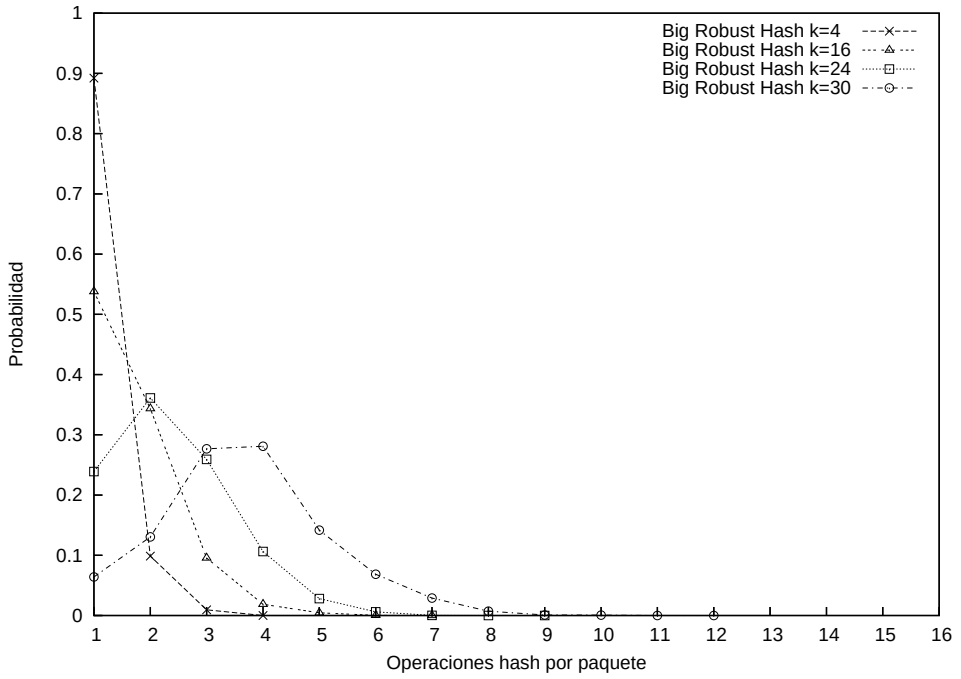


Figura 5.7: Funciones de Densidad para el algoritmo de *Big Robust Hash* ($n = 32$)

h	**1**	**2**	**3**	**4**	**5**
$n = 16, k = 4$	0.75	0.9738	0.9987	0.9999	1.0
$n = 16, k = 8$	0.5	0.8627	0.9771	0.9975	0.9998
$n = 16, k = 12$	0.25	0.6212	0.8694	0.9683	0.9944
$n = 32, k = 8$	0.75	0.9697	0.9978	0.9999	0.9999
$n = 32, k = 16$	0.5	0.8545	0.9720	0.9962	0.9996
$n = 32, k = 24$	0.25	0.6086	0.8531	0.9586	0.9909
$n = 64, k = 16$	0.75	0.9677	0.9973	0.9998	0.9999
$n = 64, k = 32$	0.5	0.8505	0.9694	0.9953	0.9994
$n = 64, k = 48$	0.25	0.6025	0.8449	0.9533	0.9887

Cuadro 5.4: Probabilidad acumulada del número de operaciones *hash* (h) para el algoritmo de *Big Robust Hash*

indica que cuando sólo 8 de las 32 CPUs se encuentran disponibles, el 99 % de los paquetes se asignaran con 5 o menos operaciones *hash* (frente a los 25 *hashes* del máximo teórico), y que la mayoría (60 %) tan sólo necesitarán 2 *hashes* (frente a los 9 *hashes* de las técnicas clásicas).

La Tabla 5.4 también permite ilustrar la escalabilidad del algoritmo, puesto que el número de operaciones *hash* no depende directamente del número de CPUs inactivas, sino de la fracción de CPUs disponibles frente al total. Por ejemplo, las probabilidades acumuladas cuando el 50 % de las CPUs están desactivadas (filas 2, 5 y 8) son casi idénticas, a pesar de que el tamaño del cluster se duplica en cada paso. Por tanto, aunque el aumento de k implica una ligera pérdida de rendimiento, claramente el factor principal es la relación $\frac{n-k}{n}$.

Por tanto estos resultados muestran que, comparado con las técnicas *clásicas* de *Robust Hash*, el algoritmo de *Big Robust Hash* puede realizar la gran mayoría de las asignaciones con un número reducido de operaciones *hash* por paquete, incluso cuando muchas CPUs se encuentran desactivadas. Es más, la probabilidad de emplear más operaciones *hash* que la media tiende rápidamente a cero.

5.6.3. Latencia de las técnicas de *Robust Hashing*

Otro aspecto muy interesante a tener en cuenta es la latencia introducida por estas técnicas durante el procesamiento de cada paquete, ya que se podría argumentar que, a diferencia de las técnicas de *Robust Hashing* anteriores, que sólo necesitaban ejecutar operaciones *hash*, los algoritmos de *Fast Robust Hashing* también necesitan acceder a memoria después de cada operación.

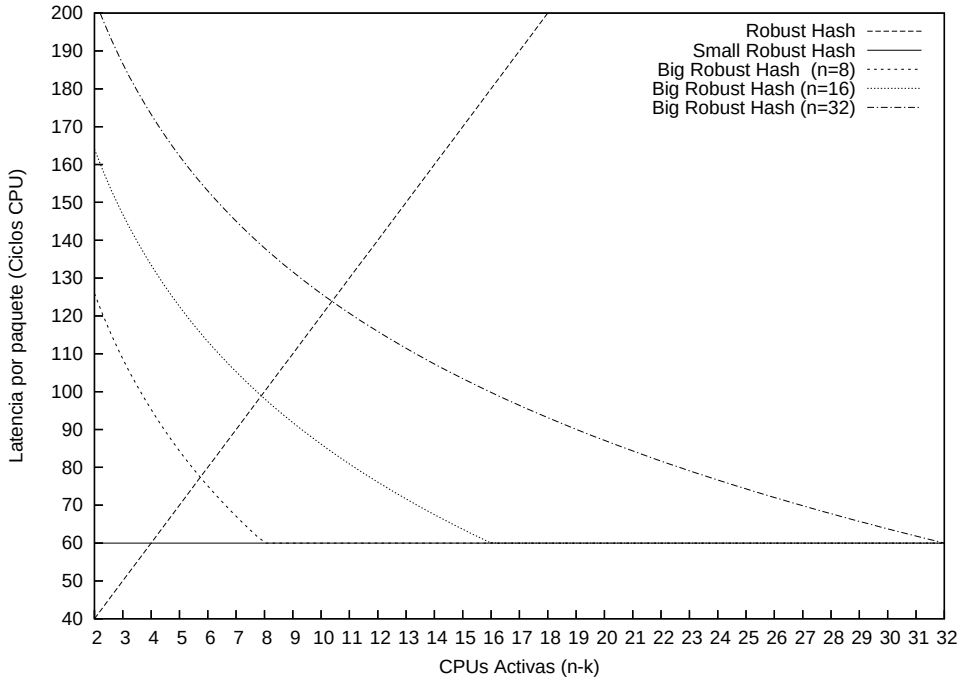


Figura 5.8: Latencia de los algoritmos de *Robust Hashing*

Para estudiar la latencia de cada algoritmo nos basaremos en la arquitectura de un procesador de red comercial, el IXP1200 de Intel [60], que dispone de seis micro-CPUs para el procesamiento de paquetes y de una unidad *hardware* especializada para realizar operaciones *hash*. Dicha unidad está segmentada (*pipeline*) lo que le permite realizar una operación *hash* cada 10 ciclos de reloj, aunque el resultado del primer *hash* tarda 30 ciclos de reloj en calcularse [82]. Por tanto, con el algoritmo *clásico* de *Robust Hashing*, para procesar cada paquete se enviarían $n-k$ operaciones a la unidad de *hash*, la primera tardaría 30 ciclos y el resto sólo 10:

$$L_{RH} = 30 + 10(n-k-1) = 20 + 10(n-k)\ ciclos$$

Sin embargo, con las técnicas de *Fast Robust Hashing*, es necesario acceder a memoria después de cada operación *hash* para consultar el Vector de Asignación. En el caso del IXP1200, si dicho vector se almacenase en memoria SRAM, el acceso a memoria tardaría 30 ciclos (40 ciclos con DRAM) [225]. Por tanto, la latencia del algoritmo *Small Robust Hash* es la suma de las latencias de una operación *hash* y un acceso a memoria:

$$L_{SRH} = 30 + 30 = 60\ ciclos$$

Por último, la latencia del algoritmo *Big Robust Hash* dependería del número de operaciones *hash* y los accesos a memoria posteriores hasta encontrar una CPU activa en la matriz de asignación. Por tanto la latencia media ($\bar{L}_{BRH}$) es:

$$\bar{L}_{BRH} = \bar{H}(k)(30+30) = 60(1+\ln(\frac{n+1}{n-k+1}))\ ciclos$$

La figura 5.8 permite comparar en líneas generales la latencia de los diferentes algoritmos de *Robust Hashing*. El crecimiento lineal del algoritmo *clásico* hace que su latencia supere a la de los algoritmos de *Fast Robust Hashing* en cuanto la suma de las operaciones *hash* necesarias supera la latencia de acceso a memoria SRAM, que en el caso del IXP1200 se situaría en 4 operaciones por paquete. En ese punto se llega a la latencia de una operación *hash* y un acceso a memoria, que caracteriza al algoritmo *Small Robust Hash*, y al estado inicial del algoritmo de *Big Robust Hash* cuando ninguna CPU se encuentra fuera de servicio. En este algoritmo, conforme el número de CPUs activas disminuye, el número medio de operaciones, y por tanto la latencia media por paquete, va aumentando hasta alcanzar el retardo del algoritmo *clásico*, o incluso puede superarlo cuando hay un gran número de CPUs inactivas (e.g. más de 8 CPUs inactivas sobre un total de 16).

Sin embargo, no hay que olvidar que, aunque la latencia del algoritmo *clásico* pueda ser inferior en algún caso a las variantes de *Fast Robust Hashing*, estas últimas realizan menos operaciones por paquete, y por tanto pueden procesar más paquetes en paralelo. Por ejemplo con 3 CPUs, la unidad de *hashing*, compartida por las 6 micro-CPUs del IXP1200, tendría que calcular 3 *hashes* por paquete para el algoritmo *clásico*, mientras que con *Fast Robust Hashing* podría procesar la *hash* de 3 paquetes diferentes. Además, los procesadores de red, y el IXP1200 en particular, tienen soporte multi-hilo *hardware* que les permite cambiar de una tarea a otra inmediatamente, sin realizar un cambio de contexto, por lo que durante el tiempo de acceso a memoria el procesador no se bloquearía, sino que podría seguir procesando otro paquete.

```
int mapping_vector[m];
int cpu_entries[n];
int available_cpus;

void init()
{
    available_cpus = n;

    int cpu_rr = 0;
    for (int i=0; i<m; i++) {
        mapping_vector[i] = cpu_rr;
        cpu_entries[cpu_rr]++;
        cpu_rr = (cpu_rr + 1) % n;
    }
}

int getCPU ( int flow_id )
{
    int index = hash(flow_id, m);
    int cpu = mapping_vector[index];

    return cpu;
}


void disableCPU ( int cpu_down )
{
    cpu_entries[cpu_down] = 0;
    available_cpus--;

    int cpu_rr = 0;
    for (int i=0; i<m; i++) {
        if (mapping_vector[i] == cpu_down) {
            while (cpu_entries[cpu] == 0) {
                cpu_rr = (cpu_rr + 1) % n;
            }
            mapping_vector[i] = cpu_rr;
            cpu_entries[cpu_rr]++;
        }
    }
}

void recoverCPU ( int cpu_up )
{
    int old_cpu_entries = m / available_cpus;
    available_cpus++;
    int new_cpu_entries = m / available_cpus;
    int steal_cpu_entries = old_cpu_entries - new_cpu_entries;
    while (steal_cpu_entries > 0) {
        int index = random(m);
        int cpu = mapping_vector[index];
        if (cpu_entries[cpu] > new_cpu_entries) {
            mapping_vector[index] = cpu_up;
            cpu_entries[cpu]--;
            steal_cpu_entries++;
        }
    }
}
```

Figura 5.9: Implementación del algoritmo *Small Robust Hash*

```
int mapping_matrix[n][n];
int k = 0;

void init()
{
    for (int i=0; i<n; i++) {
       mapping_matrix[i][0] = i;
    }
}

int getCPU ( int flow_id )
{
    int j = 0;
    int cpu;
    do {
       int index = hash(flow_id, n-j);
       cpu = mapping_matrix[index][j];
       if (cpu < 0) {
           j = j - cpu;
       }
    } while (cpu < 0);

    return cpu;
}

void disableCPU ( int cpu_down )
{
    k++;
    for (int j=0; j<k; j++) {
        for (int i=0; i<n-j; i++) {
            int hop = k-j;
            if (mapping_matrix[i][j] == cpu_down) {
                mapping_matrix[i][j] = hop;
            }
        }
    }
    int bottom_index = 0;
    for (int i=0; i<n-k; i++) {
        int cpu = mapping_matrix[i][k-1];
        if (cpu >= 0) {
            mapping_matrix[bottom_index][k] = cpu;
            bottom_index++;
        }
    }
}

void recoverCPU ( int cpu_up )
{
    for (int j=0; j<k; j++) {
        int hop_index = -1;
        do {
          hop_index++;
          int hop = mapping_matrix[hop_index][j];
        } while(hop != k-j);
        mapping_matrix[hop_index][j] = cpu_up;
    }
    k--;
}
```

Figura 5.10: Implementación del algoritmo *Big Robust Hash*

Capítulo 6

eXtensible Service Discovery Framework (XSDF)

6.1. Introducción

Tal como se analizó en capítulos anteriores, dada la heterogeneidad de los servicios de red que se pueden ejecutar en el *cluster* de asistentes de un nodo de red SARA, se necesita disponer de un gran número de técnicas y políticas de reparto de carga, de modo que cada aplicación de red pueda escoger la más adecuada a sus necesidades.

Si el capítulo anterior se ha centrado en desarrollar varias técnicas de reparto de carga extremadamente eficientes, que pudiesen emplearse directamente en el plano de datos de un *router* de altas prestaciones, este capítulo tratará del problema complementario, esto es, de políticas de reparto de carga flexibles para el plano de control del *router*.

En este caso, a día de hoy ya existe un gran número de políticas de reparto de carga [40] lo suficientemente variado para cubrir el amplio espectro de requisitos de los servicios de red actuales y, posiblemente, también de los futuros. Por tanto, en la actualidad parece que la única necesidad pendiente pasa por encontrar un mecanismo que permitiese a cada aplicación de red indicar al *router* cual es la política de reparto de carga más apropiada para ella, y además que fuese lo suficientemente extensible para poder definir un gran número de políticas de selección.

Sin embargo, a partir del estudio de las técnicas y protocolos de reparto de carga en *cluster* como Rserpool [238], se observó un aspecto muy interesante, común a todos los mecanismos analizados: en todos ellos se conocía a priori a todos los servidores que formaban parte del *cluster*, lo que esencialmente convierte al *cluster* en un entorno estático, donde añadir un nuevo servidor u otra aplicación requiere la intervención manual del administrador.

Por el contrario, la arquitectura SARA es altamente dinámica gracias a la carga de código bajo demanda, y por tanto requiere un mecanismo que permita al *router* localizar automáticamente a todos los asistentes que ejecutan una determinado servicio de red, desde el mismo momento en que se carga su código, o lo que es lo mismo, algún protocolo de descubrimiento de servicios como SLP [104].

Del estudio de SLP y demás mecanismos de descubrimiento de servicios se obtuvo otra conclusión interesante: a pesar de la enorme similitud entre las arquitecturas de descubrimiento de servicios y de reparto de carga [153], ninguno de los protocolos de descubrimiento de servicios definidos hasta ahora permite trabajar con *clusters* de servidores replicados, ya que no definen ningún mecanismo para elegir el mejor servidor de entre todos los que ofrecen un servicio determinado.

Por tanto, los protocolos de descubrimiento de servicios y los mecanismos de reparto de carga se complementan entre sí: lo primeros permiten descubrir los servidores que ofrecen un determinado servicio, mientras que los segundos permiten seleccionar uno de ellos según una política de reparto de carga determinada.

Sin embargo, aunque ningún mecanismo estudiado realiza el proceso completo, los dos grupos de protocolos tienen muchas características solapadas, como el registro centralizado de los servicios/servidores o las primitivas de consulta, por lo que al emplear varios protocolos se duplicarían un gran número de tareas.

Por esta razón, en esta tesis se propone un nuevo mecanismo que integra el descubrimiento de servicios y el reparto de carga en un único proceso, basado en las principales características y la arquitectura común de SLP [104] y Rserpool [238], los principales protocolos de cada área de trabajo.

El resultado de este proceso de integración se ha denominado *eXtensible Service Discovery Framework* (XSDF), y es el punto fundamental de este capítulo. Además, dada la novedad de este enfoque, XSDF no se restringe a su utilización

dentro de SARA, sino que se ha diseñado con vistas a resolver el problema general de descubrimiento de servicios en Internet, incluso para soportar políticas de Reparto de Carga Global entre múltiples sedes.

Además, en el diseño de XSDF se han incluido numerosas mejoras que intentan resolver los problemas de los protocolos y técnicas actuales de descubrimiento de servicios y reparto de carga. En los siguientes apartados se repasará brevemente cada uno de los mecanismos, que ya han sido descritos en el capítulo 2, identificando los principales problemas y limitaciones de cada uno de ellos.

6.1.1. Descubrimiento de Servicios: SLP

Tal como se estudió en el apartado 2.7, los múltiples beneficios del descubrimiento de servicios han fomentado el desarrollo de múltiples protocolos, como el *Service Location Protocol* (SLP) [104], un estándar del IETF que ha sido diseñado para encontrar servicios disponibles en el entorno de red local. Una de las principales diferencias entre SLP y otros protocolos reside en sus escenarios de aplicación. Mientras que la mayoría de los protocolos de descubrimiento de servicios están enfocados en pequeñas redes LAN sin administración, como una casa o una pequeña oficina, SLP ha sido diseñado cuidadosamente para poder escalar desde esas pequeñas LANs a grandes redes corporativas.

Esta escalabilidad se obtiene combinando diversos mecanismos. En redes pequeñas sin infraestructura estable, los Agentes de Usuario[1] (UA) localizan servicios enviando consultas *multicast* a los Agentes de Servicio[2] (SA) que residen en cualquier ordenador que ofrezca un servicio. Sin embargo cuando el número de usuarios y servicios aumenta, el tráfico *multicast* puede sobrecargar la red. En ese caso, es recomendable desplegar una entidad opcional de la arquitectura, denominada Agente de Directorio[3] (DA), que actúa como un repositorio centralizado de información de servicios, y donde los Agentes de Servicio publican sus servicios. Esto permite a los Agentes de Usuario enviar sus consultas directamente al Agente de Directorio empleando *unicast*. Por tanto, no es necesario que los Agentes SLP empleen consultas *multicast*, si no es para localizar inicialmente al Agente de Directorio (aunque también es posible emplear DHCP para configurar la localización del DA). Para poder escalar aún más, o cuando una organización está dividida en varios departamentos, es posible dividir a

[1] User Agent, en inglés
[2] Service Agent, en inglés
[3] Directory Agent, en inglés

conjuntos de usuarios y servicios en varios Grupos[4] independientes, que pueden tener su propio Agente de Directorio.

Además, SLP consigue esta escalabilidad excepcional sin sacrificar su simplicidad. Sin embargo, a veces tanta simplicidad se convierte en un problema, y un buen ejemplo de ello es el modelado de los servicios en SLP. En SLP los servicios se localizan mediante su URL, y opcionalmente pueden tener una lista de pares atributo-valor para poder describirlos en más detalle. Además, la URL también se emplea como identificador del servicio. En [101], Guttman estudia los problemas derivados de este modelo de servicio excesivamente simple:

1. Un servicio está atado a una única localización, y por tanto no puede tener varias direcciones, ni cambiar de dirección.

2. No puede accederse a un único servicio mediante varios protocolos de aplicación (e.g. `lpr`, `ipp`).

3. Las URLs no permiten indicar qué protocolos de transporte pueden emplearse para acceder al servicio (e.g. SCTP).

Para solventar estos problemas, Guttman sugiere [101] identificar a los servicios con una URN en lugar de una URL, que tan sólo contiene un UUID[5] [149], mientras que la información sobre los protocolos de transporte y aplicación soportados, su localización y otra información adicional (nombre, descripción, etc) se envían como atributos asociados al servicio.

Aunque esta modificación permite mantener el formato de los mensajes SLP, los Agentes de Usuario deben cambiar su comportamiento estándar cuando encuentran servicios de este tipo. Además, añadir tantos atributos (codificados en texto) a cada servicio puede suponer una gran sobrecarga, tal como se analizará en la sección 6.6.

6.1.2. Reparto de Carga en *Clusters*: Rserpool

Dado que replicar un servicio en múltiples servidores se ha convertido en un mecanismo bastante común para conseguir escalabilidad y alta disponibilidad, el reparto de carga entre servidores desacoplados ha sido un tema muy popular,

[4] Scopes, en inglés
[5] Universally Unique Identifier, en inglés

tanto en la literatura académica como en la comercial, y especialmente en el campo de los servidores web [38].

Los diseños iniciales con varios servidores web empleaban el DNS para balancear la carga entre ellos. Cuando los clientes pedían al servidor DNS autoritativo la dirección IP del servidor web, este devolvía cada vez la IP de un servidor diferente (*Round-Robin*). Sin embargo, la cache de los servidores DNS locales (o de los *proxies* web) producía un reparto de carga desigual entre servidores, especialmente en las situaciones de alta carga [161].

Como se considera que la técnica de DNS *Round-Robin* permite realizar un reparto de carga bastante tosco, muchas organizaciones han desplegado Conmutadores L4/L7 ("Balanceadores de Carga") como primera capa de sus *clusters* de servidores web para conseguir una mejor distribución de carga. Estos elementos se comportan como un NAT, ocultando los servidores finales, e implementan diferentes algoritmos de selección para escoger al mejor servidor basándose en métricas como la capacidad del servidor, su carga actual, el tiempo de respuesta o el número de conexiones de clientes. Además, algunos dispositivos capturan la cabecera HTTP de las peticiones y son capaces de realizar esta selección basándose en la URL de la petición u otra información sobre el servicio solicitado.

Un mecanismo alternativo para repartir la carga, es la selección de servidor en el cliente [78], que a diferencia de los anteriores [112], se ajusta perfectamente con la arquitectura extremo-a-extremo de Internet. Quizá el grupo de trabajo Rserpool del IETF es el mejor ejemplo de este tipo de técnicas. Rserpool define una arquitectura [238], muy similar a la de SLP [153], donde a un conjunto de servidores que proporcionan el mismo servicio se le denomina *Pool*. Para poder acceder a alguno de los servidores del *cluster*, el Usuario del *Pool*[6] (PU) pide al gestor del *cluster* -denominado Servidor ENRP- todos los Elementos del *Pool*[7](PE) disponibles, y el PU elige al mejor de ellos.

Dado que la política de selección depende en gran medida del tipo de servicio solicitado, Rserpool es extensible y permite diferentes mecanismos y métricas de reparto de carga. Además, dado que el cliente conoce varios servidores que ofrecen el mismo servicio, en caso de fallo puede seleccionar un servidor alternativo gracias al protocolo ASAP [231]. Dado que Rserpool está orientado a *clusters* con alta disponibilidad, es posible emplear varios servidores ENRP coordinados mediante el protocolo del mismo nombre [258].

[6]Pool User, en inglés
[7]Pool Element, en inglés

Rserpool emplea obligatoriamente SCTP como protocolo de transporte. Sin embargo, aún no se ha estudiado el efecto que puede tener sobre servicios interactivos el retardo introducido por establecimiento de la conexión SCTP para consultar el servidor ENRP. Por ejemplo, en la navegación Web la resolución de nombres DNS es mucho más rápida y, sin embargo, puede afectar en gran medida a la percepción del usuario sobre el rendimiento del servicio [220].

6.1.3. Reparto de Carga Global

La replicación de servicios puede ir un paso más allá y no sólo emplear múltiples servidores, sino también múltiples sedes repartidas por todo el mundo. Normalmente en este escenario se necesitan varias técnicas de reparto de carga: primero una política de reparto de carga global para seleccionar la sede y luego una política de reparto de carga local para elegir uno de los servidores de dicha sede.

Una forma bastante habitual para implementar el reparto de carga global consiste en emplear un servidor DNS autoritativo modificado. Cuando un cliente le envía una consulta DNS sobre la página web de la organización, este devuelve la dirección IP del balanceador de carga de la sede más cercana al usuario. Después, el balanceador de carga redirige la conexión del cliente al mejor servidor dentro de la sede. Sin embargo, dado que la conexión del cliente -y por tanto la petición HTTP- sólo se establece con el balanceador de carga de la sede escogida, la selección a nivel global no puede basarse en el contenido de la petición (e.g. la URL), sino meramente en la consulta DNS inicial, a no ser que se emplee una conexión triangular entre el cliente, la primera sede (que realiza la selección definitiva de sede basada en el contenido de la petición HTTP), y el balanceador de carga de la sede definitiva.

Otro problema de esta técnica consiste en encontrar la sede más "cercana" al cliente, pero desde la perspectiva del proveedor de servicios. Algunos heurísticas incluyen: la consulta de tablas de rutas BGP, el envió de *pings* desde todas las sedes al cliente o la recopilación de información estadística sobre conexiones anteriores. Ninguna de estas técnicas es sencilla y, lo que es peor, ni siquiera ofrecen garantías de proximidad efectiva, sobre todo si tenemos en cuenta que la dirección IP que se emplea como índice para el reparto de carga global no suele ser directamente la del cliente final sino la de su servidor de DNS local que, a pesar de lo que pueda parecer, en los ISPs suele estar bastante alejado [220] del cliente.

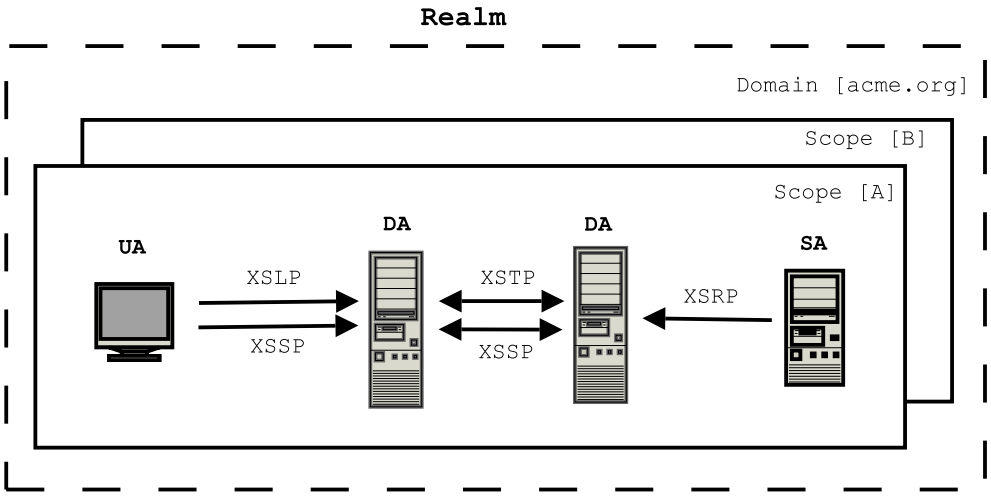


Figura 6.1: Vista general de la arquitectura de XSDF y sus protocolos.

6.2. *eXtensible Service Discovery Framework* (XSDF)

El *eXtensible Service Discovery Framework* (XSDF) ha sido diseñado para intentar resolver los problemas descritos en los apartados anteriores, puesto que integra en una arquitectura común tanto el descubrimiento de servicios como el reparto de carga. Como otros entornos de descubrimiento de servicios, y tal como se muestra en la figura 6.1, XSDF está compuesto por varios agentes y por los protocolos que emplean para comunicarse entre sí.

Al igual que en SLP, XSDF tan sólo se ocupa del descubrimiento de servicios (y adicionalmente de la Selección del mejor servicio). Una vez que un UA ha escogido un servicio, puede emplear cualquier otro protocolo para comunicarse con él. Por tanto, XSDF tan sólo permite descubrir la información de localización y los protocolos que pueden emplearse posteriormente.

En particular, XSDF hereda la misma arquitectura y nomenclatura de SLP, aunque hay varias diferencias entre ambos. Quizá la más visible sea que XSDF define varios protocolos muy simples, en lugar de uno sólo, como en el caso de SLP. Cada uno de estos protocolos cliente-servidor se encarga de una de las partes que forman el proceso de descubrimiento de servicios:

- **eXtensible Service Location Protocol (XSLP)** [241]: Los Agentes de Usuario (UA) emplean este protocolo para obtener información de los Agentes de Servicio (SA) o de los Agentes de Directorio (DA). XSLP es el protocolo principal de XSDF y permite tanto consultas *unicast* al DA,

como consultas *multicast* a todos los SAs, cuando no hay DAs desplegados en la red.

- **eXtensible Service Registration Protocol(XSRP)** [242]: Cuando un grupo de trabajo está gestionado por uno o más Agentes de Directorio, los Agentes de Servicio deben emplear XSRP para registrar la información de sus servicios en el DA. Por tanto, XSRP permite agregar toda la información de servicios en una cache central, lo que posibilita las búsquedas XSLP *unicast*.

- **eXtensible Service Subscription Protocol (XSSP)** [243]: Este protocolo permite a cualquier Agente XSDF suscribirse a los cambios en la información de servicios, mediante el registro de un canal de notificaciones. Todos los cambios que afecten a la información de servicios asociada se envían a dicho canal, que puede ser un Agente individual o una dirección *multicast* para distribuir los eventos a varios Agentes. Los Agentes de Directorio con grupos comunes emplean este protocolo para suscribirse a las operaciones XSRP que afectan a dichos grupos.

- **eXtensible Service Transfer Protocol (XSTP)** [244]: Para conseguir un servicio de descubrimiento de servicios escalable y de alta disponibilidad, cada grupo de trabajo puede estar gestionado por varios Agentes de Directorio. Estos DAs emplean el protocolo XSTP cuando se inicializan para obtener de otro DA toda la información de servicios disponible. Posteriormente también se emplea para sincronizar el repositorio de servicios. Para ello basta con reenviar todas las operaciones XSRP recibidas, que afecten al grupo de trabajo común, al resto de DAs, que se subscriben mediante XSSP y emplean un canal de notificaciones XSTP para encapsular dichos mensajes XSRP.

Aunque XSDF puede parecer más complejo que SLP o Rserpool, que sólo emplean uno y dos protocolos respectivamente, este autor considera que esta separación en cuatro protocolos redunda en una arquitectura más simple. Por ejemplo, tanto en SLP como en Rserpool, las consultas de los UAs/PUs y las operaciones de registro de los SAs/PEs están definidas en el mismo protocolo (SLP/ASAP), aunque cada tipo de operación tiene unas funcionalidades, requisitos de seguridad y agentes origen/destino muy diferentes entre sí. Al separar estas operaciones disjuntas en dos protocolos, es mucho más sencillo gestionar la infraestructura de descubrimiento de servicios. Por ejemplo, bastaría con filtrar todos los puertos XSDF en el *firewall* de la organización, excepto la consultas

XSLP dirigidas a sus DA públicos, para implementar una política de seguridad perimetral sencilla pero muy eficaz.

Además, la utilización de varios protocolos no implica que los Agentes tengan que ser más complejos, puesto que cada tipo de Agente tiene que implementar solamente la parte cliente o servidora adecuada y además, dependiendo del escenario de aplicación, algunos de los protocolos XSDF (i.e. XSRP, XSSP, XSTP) puede ser innecesario. Por ejemplo los Agentes de Usuario podrían limitarse a soportar solamente la parte cliente de XSLP, mientras que los Agentes de Directorio tienen que implementar XSLP y la parte cliente de XSRP. Por tanto, los protocolos más complejos: XSSP, XSTP y el servidor de XSRP tan sólo son necesarios cuando se emplean Agentes de Directorio, que además serán servidores especializados, a diferencia de los UAs y SAs, que pueden ejecutarse en cualquier dispositivo y por tanto deben simplificarse al máximo.

6.2.1. Modelado de los Servicios

En SLP un servicio se define con una URL, y una lista opcional de pares atributo-valor textuales (figura 6.4). Por el contrario, en Rserpool la información de los servicios tiene formato binario, ya que está formada por una o más direcciones de transporte, esto es, un puerto y una o más direcciones IP por cada protocolo de transporte soportado, además de ciertos datos para la política de selección.

El modelo de servicio en XSDF mezcla ambas aproximaciones en una única representación extensible de servicio (figura 6.2), que está dividida en cinco partes bien diferenciadas:

- **Identificador de Servicio**: Cada instancia de un servicio está identificada por un UUID de 128 bits [149], que debe ser único dentro del Dominio.
- **Estado del Servicio**: Contiene datos volátiles que no deberían almacenarse en la cache de los Agentes de Usuario durante demasiado tiempo (e.g. la carga de trabajo actual del servidor). Además, este apartado contiene campos con la versión del resto de la información, de forma que sea posible detectar si ha cambiado simplemente consultando esta parte de los datos.
- **Información principal del Servicio**: Dentro de estos datos se incluye el Tipo del Servicio, su Política de Selección y otra información importan-

```
<service>
  <id>8e9d7823-d5ac-497c-91d0-fb07ea0c3fb2</id>

  <serviceState>
    <metaInfo>
      <stateTimestamp>f85444f4eb</stateTimestamp>
    </metaInfo>
    <selectState>
      <workload>0</workload>
    </selectState>
  </serviceState>

  <serviceMainInfo>
    <serviceType>
      <type>printer</type>
    </serviceType>
    <alias>Alice's printer</alias>
    <selectInfo>
      <policies>Least Used (0x0002)</policies>
      <weight>14</weight>
    </selectInfo>
    <printer:color>false</printer:color>
    <printer:duplex>true</printer:duplex>
  </serviceMainInfo>

  <serviceLocation>
    <inet>
      <ipv4Addrs>169.254.85.139</ipv4Addrs>
      <ipv6Addrs>fe80::202:b3ff:fe3c:da7a</ipv6Addrs>
    </inet>
    <protocol>
      <name>ipp</name>
      <transPorts>tcp/631, sctp/631</transPorts>
    </protocol>
    <protocol>
      <name>lpr</name>
      <transPorts>tcp/515, sctp/515</transPorts>
    </protocol>
  </serviceLocation>

  <serviceAddInfo>
    <model>Acme Laser Printer 2000</model>
    <modelURL>http://www.acme.org/printers/lp2000.html</modelURL>
  </serviceAddInfo>
</service>
```

Figura 6.2: Un ejemplo del servicio XSDF de una impresora, codificado en XML

```
01000284 35110014  8e9d7823 d5ac497c    ....5... ..x#..I|
91d0fb07 ea0c3fb2  01100020 0111000f    ......?. ... ....
331a000c 000000f8  5444f4eb 0311000c    3.......  TD......
32320008 00000000  0120006c 0121000f    22......  . .l.!..
2812000b 7072696e  74657200 28140013    (...prin  ter.(...
416c6963 65277320  7072696e 74657200    Alice's   printer.
03210014 31330008  00020000 32350008    .!..13..  ....25..
0000000d 10000018  20020009 636f6c6f    ........   ...colo
72000000 30020005  00000000 10000018    r...0...  ........
2002000a 6475706c  65780000 30020005     ...dupl  ex..0...
ff000000 01300054  01310020 32150008    .....0.T  .1. 2...
a9fe558b 35160014  fe800000 00000000    ..U.5...  ........
0202b3ff fe3cda7a  01320018 28610007    .....<.z  .2..(a..
69707000 321a000c  00060277 00840277    ipp.2...  ...w...w
01320018 28610007  6c707200 321a000c    .2..(a..  lpr.2...
00060203 00840203  0140008c 2867001b    ........  .@..(g..
41636d65 204c6173  65722050 72696e74    Acme Las  er Print
65722032 30303000  2868002c 68747470    er 2000.  (h.,http
3a2f2f77 77772e61  636d652e 6f69672f    ://www.a  cme.org/
7072696e 74657273  2f6c7032 3030302e    printers  /lp2000.
68746d6c                                html
```

Figura 6.3: El servicio de impresora codificado con XBE32 (vistas hexadecimal y ASCII)

```
(service-id=8e9d7823-d5ac-497c-91d0-fb07ea0c3fb2),
(service-hi-name=Alice's printer),
(service-location-tcp=ipp://169.254.85.139:631,
                      ipp://[fe80::202:b3ff:fe3c:da7a]:631,
                      lpr://169.254.85.139:515,
                      lpr://[fe80::202:b3ff:fe3c:da7a]:515),
(service-location-sctp=ipp://169.254.85.139:631,
                       ipp://[fe80::202:b3ff:fe3c:da7a]:631,
                       lpr://169.254.85.139:515,
                       lpr://[fe80::202:b3ff:fe3c:da7a]:515),
(service-selection-policies=Least Used),
(service-workload=0),
(service-weight=14),
(service-model=Acme Laser Printer 2000),
(service-model-url=http://www.acme.org/printers/lp2000.html),
(color=false),
(duplex=true)
```

Figura 6.4: El servicio de impresora codificado con atributos SLPv2

te dependiente del servicio, como por ejemplo si una impresora soporta impresión a doble cara o a color.

- **Información de Localización del Servicio**: Para poder comunicarse con un servicio de red, los clientes deben saber dónde pueden localizarle, y qué protocolos pueden emplear para acceder al servicio. La información de localización incluye esos datos, tales como las direcciones IPv4/IPv6 del servidor, los puertos de nivel de transporte o qué protocolos de nivel de aplicación están soportados.
- **Información Adicional del Servicio**: Mientras que con los datos anteriores ya es posible seleccionar y acceder a un servicio, la información adicional contiene datos destinados a los usuarios del servicio, como una descripción textual del mismo, o los datos de contacto del administrador del servicio.

Cuando un UA pide información de servicios puede especificar en qué parte de datos está interesado. Esto permite que los UAs no interactivos tan sólo descarguen inicialmente el Estado, la Información Principal y de Localización del servicio, para seleccionar el mejor servidor, pero que después les baste con actualizar el Estado para ver si el resto de la información ha cambiado. Y todo ello sin descargar la Información Adicional, que no es necesaria para acceder al servicio, pero que, al contener mucha información textual para los usuarios, será mucho más voluminosa que el resto. Por el contrario, como en SLP hay una única lista de atributos, es necesario [101] descargar toda la información del servicio para poder acceder al mismo, que puede incluir datos como una larga descripción textual del mismo.

Los mensajes XSDF, y por tanto la información de servicios, pueden transmitirse por la red empleando cualquier mecanismo de codificación extensible y válido para datos estructurados jerárquicamente, como por ejemplo XML (figura 6.2). Sin embargo, se ha preferido emplear la codificación binaria XBE32 [240], definida en el apartado 4.3 ya que es más compacta, está alineada a 32 bits, y su procesamiento es más eficiente que XML o las reglas de codificación de ASN.1. Por ejemplo, la figura 6.3 muestra cómo se codifica con XBE32 el mismo servicio de impresora de la figura 6.2 (887 bytes) en tan sólo 324 bytes, sin aplicar ningún tipo de algoritmo de compresión, sino utilizando simplemente una codificación binaria (XBE32) en lugar de textual, como en el caso de XML o SLP (e.g. el servicio SLP de la figura 6.4 ocupa 586 bytes).

Este modelo de servicio es tan extensible que incluso permite representar a

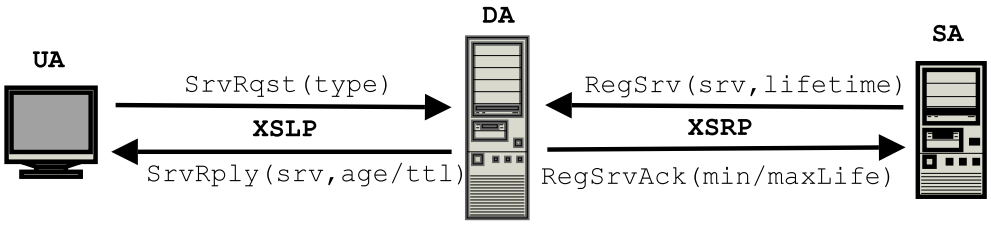


Figura 6.5: Flujo de información entre las caches de XSDF.

los propios Agentes XSDF como servicios. Por el contrario, tanto en SLP como en Rserpool se han definido mensajes y procedimiento específicos para localizar a los Agentes de Directorio y a los Servidores ENRP, puesto que necesitan definir información adicional que no puede incluirse dentro de la información de los servicios. Por tanto en una red XSDF el descubrimiento de servicios es otro servicio de red más, y como tal puede beneficiarse de sus capacidades de reparto de carga y de localización de servicios, por ejemplo para localizar al Agente de Directorio menos cargado.

6.2.2. Funcionamiento de las Caches de Servicios

Los Agentes de Servicio generan y mantienen la información de sus servicios, de forma que sólo el SA "hogar" de un servicio puede modificar dicha información. Sin embargo, para mejorar el rendimiento de la red, los otros dos tipos de Agentes XSDF (UAs y DAs) pueden almacenarla temporalmente en una cache.

Los Agentes de Servicio asignan una duración (`lifetime`) a la información de servicios, que especifica la duración máxima durante la que se espera que esos datos de servicio sigan siendo válidos. Por tanto, la información del servicio puede estar en la cache de los UAs y DAs hasta que se sobrepase ese periodo.

Un Agente de Directorio es simplemente una cache centralizada con la información de los servicios de un grupo. Cuando un SA registra un servicio en el DA empleando XSRP, el SA especifica el `lifetime` esperado de esa información. Cuando un UA pide información de servicios al DA mediante XSLP, la respuesta del DA también contiene la "edad" (`age`) y el tiempo de vida (`ttl`) restante de dicha información.

`age` es la cantidad de tiempo que la información de servicio lleva almacenada en el DA cuando se generó la respuesta XSLP, y permite a los UAs saber, por ejemplo, si la información de Estado del Servicio es lo suficientemente reciente.

El `ttl` es la cantidad de tiempo restante que la información puede almacenarse en la cache del UA. Por tanto, el `lifetime` del servicio debería ser igual al `ttl` restante más la `age` de la cache DA. Cada vez que un SA refresca la información de servicio almacenada en el DA, `age` vuelve a cero y el `ttl` se fija al `lifetime` especificado.

Por tanto, el valor de `lifetime` determina el comportamiento de la cache del UA. Sin embargo, la información de servicio almacenada en la cache de los DAs puede borrarse antes de que su duración máxima expire, a no ser que sea refrescada periódicamente por su SA "hogar". Cuando se confirma el registro/actualización de un servicio, el DA especifica los valores de retardo `minLife` y `maxLife` de la próxima actualización (en milisegundos). Los SAs deben actualizar dicha información antes que pase `maxLife` pero esperando como mínimo `minLife`. Por tanto, a diferencia de SLP y ASAP, este mecanismo permite que servicios con un valor `lifetime` de cero, esto es, que no deben ser almacenados en la cache de los UAs, puedan ser almacenados en el DA central durante `maxLife` (los SAs pueden negociar este valor en cada actualización).

Además de los servicios, las subscripciones XSSP también siguen este esquema robusto de *soft-state* para almacenar en una cache la información de suscripción. Si cualquier subscripción no se refresca en el intervalo `minLife-maxLife`, dicha subscripción se borra y los eventos dejan de enviarse al canal de notificaciones.

Sin embargo, la característica más interesante es que este mecanismo de `minLife-maxLife` mejora la escalabilidad de XSDF, puesto que permite evitar tormentas de actualizaciones de los SAs a los DAs, debido a que ciertas políticas de reparto de carga necesitan que la información de estado de los servicios esté lo más actualizada posible, lo cual puede ser un problema si hay muchos SAs en la red.

Para limitar el consumo de recursos en XSDF, el administrador de la red puede controlar la tasa de actualizaciones XSRP de todos los SAs de forma centralizada a través del DA. En SLP, por el contrario, cada SA define su propia tasa de actualización y, por tanto, no es consciente del tráfico generado por el resto de los SAs, de forma que el tráfico agregado puede llegar a saturar al DA.

Por tanto, conforme aumenta el número de servicios en la red, y por consiguiente la tasa de actualizaciones global, con SLP sería necesario re-configurar la tasa de actualización de todos los servicios de la red, mientras que con XSDF

bastaría con configurar en el DA central el intervalo `minLife-maxLife` de cada Tipo de Servicio. Es más, con XSDF sería posible emplear un mecanismo de adaptación automática en el DA para mantener la tasa global de actualizaciones constante. De este modo, se podría limitar el consumo de recursos de red cuando hay un gran número de SAs, o por el contrario reducir el tiempo de actualización si hay muy pocos servicios.

6.2.3. Políticas de Selección de Servicios

Cuando un UA solicita información de un servicio en *cluster* vía XSLP, es posible que encuentre varias instancias del servicio que cumplan sus necesidades y por tanto será necesario aplicar algún tipo de proceso de selección. Por ejemplo, la opción más sencilla es la selección manual, que es la única disponible en SLP, y que consistiría en mostrar la lista de servicios localizados (e.g. Impresoras) al usuario para que este elija el más adecuado (e.g. la más cercana).

Sin embargo, para servicios donde es posible realizar un proceso de selección automático, XSDF proporciona un modelo de selección similar a Rserpool: cada Tipo de Servicio puede tener una o más Políticas de Selección, y cada instancia del servicio proporciona alguna métrica sobre la "bondad" del mismo, lo que permite elegir al "mejor" de ellos sin intervención humana.

Aunque XSDF permite que cada Tipo de Servicio pueda definir sus propias Políticas de Selección de Servicio y métricas asociadas, también se han especificado varias Políticas de Selección estándar que permiten cubrir los requisitos de reparto de carga de la mayoría de los servicios conocidos:

- **Round-Robin**: Esta política indica que cada vez que un cliente desee acceder a un servicio, debería elegir cíclicamente una instancia diferente del mismo. Los servicios pueden tener un peso (`weight`) asociado, según el cual las instancias del servicio con más peso debería ser accedidas proporcionalmente más veces que aquellas con un peso menor.
- **Menos Usado**: Los servicios con esta política piden a los clientes que accedan al servidor que tenga la menor carga de trabajo (`workload`).
- **Más Recursos**: Según esta política, los clientes deberían acceder al servidor que tenga el mayor número de recursos disponibles (`resources`).
- **Más Cercano**: Con esta política un cliente debe estimar el *Round Trip*

Time (RTT) de cada uno de los servidores localizados, para poder seleccionar el servidor más cercano.

Además de esta políticas particulares, la información de Estado del Servicio también afecta al proceso de selección. Por ejemplo si el servicio no tiene más recursos libres (`resources=0`), se considera que dicho servicio está inactivo temporalmente y los clientes deberían acceder a otro de los servicios descubiertos, ordenados según la Política de Selección en uso.

Junto a los valores de `weight`, `workload` y `resources`, XSDF define una cuarta métrica de selección, que debe aplicarse antes de cualquier otra política: la prioridad (`priority`). Cuando dos servicios del mismo Tipo tienen valores de `priority` diferentes, los clientes deberían acceder al servicio con mayor prioridad. Sólo en el caso de que el servidor con mayor prioridad se desactive, los clientes pueden acceder al siguiente servicio de mayor prioridad. Cuando dos o más servicios tienen la misma prioridad, debe elegirse uno de ellos basándose en la política de selección especificada.

Al contrario que ASAP [231], donde el proceso de selección siempre se aplica en los clientes finales (PUs), en XSDF este proceso puede realizarse en el UA, en el DA, o en ambos. Por ejemplo esto permite que el DA pueda repartir las peticiones de los clientes mediante un proceso de *Round-Robin* centralizado, mientras que los UAs sólo pueden aplicar un proceso de selección aleatoria, ya que no son conscientes de las peticiones del resto de clientes.

Por tanto, los SAs pueden incluir datos de selección en la Información Principal del servicio para que los UAs elijan el mejor servicio descubierto. Adicionalmente, los SAs también pueden incluir esa información en los mensajes de registro XSRP para el DA. En ese caso, el DA ordena los servicios registrados y, cuando un UA solicita información del servicio, el DA devolverá los mejores servicios pre-seleccionados. Por tanto los SAs pueden especificar el primer, segundo u ambos tipos de datos de selección, que además pueden ser diferentes entre sí.

La combinación de políticas de selección puede resultar muy útil en determinados escenarios. Por ejemplo, la política de Menos Usado necesita una información de `workload` actualizada para que sea fiable. Ese tráfico de actualización no es problema dentro de una LAN, pero puede ser un inconveniente si el UA emplea una conexión por módem. En ese caso, el DA local podría aplicar fácilmente la política de selección del Menos Usado, mientras que el UA

podría aplicar otras políticas que requieran menos ancho de banda, como por ejemplo localizar un servidor de respaldo (con menor `priority`) para usarlo si el primario falla.

6.2.4. Organización en "Reinos"

Al igual que SLP, XSDF emplea el concepto de grupo de trabajo para dividir servicios y usuarios en grupos administrativos, de modo que las búsquedas de los usuarios estén limitadas a los servicios de su mismo grupo. Sin embargo XSDF añade el concepto de "Dominio"[8] para permitir la búsqueda de servicios Remotos. De este modo se define un "Reino"[9] como la combinación de un dominio (si lo hay) y uno o más grupos.

Aunque cada organización puede definir todos los grupos de trabajo que quiera de acuerdo a sus necesidades, XSDF ha definido tres grupos estándar:

- **DEFAULT**: Por defecto todos los servicios pertenecen al grupo `DEFAULT` y no tienen dominio. Este configuración por defecto permite que en redes pequeñas sin administración (e.g. SOHO), cualquier dispositivo nuevo que se conecte a la red pueda ser localizado inmediatamente al tener preconfigurada la información de grupo XSDF.
- **LOCAL**: Los servicios que pertenecen a este grupo tan sólo tienen sentido en el contexto del servidor local, como por ejemplo los servicios de gestión como SNMP o Telnet/SSH. Por tanto los servicios de este grupo no deben registrarse en un DA, sino que tan sólo pueden consultarse mediante peticiones enviadas directamente al SA.
- **PUBLIC**: Cuando una organización desea hacer públicos alguno de sus servicios, basta con asignarlos al Reino formado por su dominio DNS y el grupo `PUBLIC` para que sean accesibles desde el exterior (en el apartado 6.4 se detalla el descubrimiento de servicios remoto con XSDF).

Una de las limitaciones [275] de SLPv2 es que si un grupo está gestionado por varios DA, los SAs deben registrar sus servicios en todos los DAs de su grupo y mantener la información consistente en todos ellos. Por el contrario en XSDF, al igual que en Rserpool, la alta disponibilidad es un requisito fundamental, y la

[8] Domain, en inglés
[9] Realm, en inglés

utilización de varios DAs en cada Reino será habitual, por tanto, para simplificar la implementación de los SAs y de los UAs, todos los DAs del mismo grupo se comportan como un único repositorio (gracias a XSSP/XSTP), de modo que aunque los servicios se registran en un sólo DA, estarán accesible desde todos los DAs del grupo.

6.3. Protocolos de XSDF

Tal como se comentó en apartados anteriores, aunque XSDF define más protocolos que SLP, en realidad la complejidad para implementarlos no aumenta puesto que los cuatro protocolos son similares entre sí, ya que comparten una estructura de mensajes común [245], formada por una cabecera obligatoria y una o más operaciones que ya dependen del protocolo en sí.

La cabecera de todos los protocolos tiene el mismo formato, e incluye un identificador de transacción para poder asociar preguntas y respuestas, información sobre los Agentes XSDF origen y destino, que se definen como servicios, y el Reino al que pertenecen.

En cuanto a las operaciones, éstas pueden ser peticiones, o respuestas a peticiones previas, de forma que cada uno de los cuatro protocolos XSDF define las transacciones y operaciones que necesita, aunque también a este nivel comparten un gran número de elementos comunes (e.g. elemento de error) [245]. La posibilidad de incluir varias operaciones en cada mensaje es otra ventaja de XSDF frente a SLP y Rserpool, ya que, por ejemplo, permite que un SA pueda actualizar el estado de varios de sus servicios en el DA simultáneamente, empleando un único mensaje XSRP.

Además, aunque XSDF emplea UDP como el protocolo de transporte por defecto para reducir la latencia de descubrimiento de servicios y permitir operaciones *multicast*, también es posible utilizar otros protocolo de transporte como TCP o SCTP.

6.3.1. *eXtensible Service Location Protocol* (XSLP)

XSLP es el protocolo fundamental de la arquitectura XSDF, puesto que es el que permite descubrir los servicios disponibles en la red. Dependiendo de la existencia o no de un Agente de Directorio, los Agentes de Usuario pueden

emplear dos modos de funcionamiento de XSLP: consultas *unicast* a un DA, o consultas *multicast* a todos los SAs de la red. Para ello se han definido las siguientes operaciones [241]:

Service Request. Esta operación permite a un UA localizar todos los servicios disponibles de un Tipo determinado (y que ofrezcan el contenido especificado), o servicios particulares identificados por su UUID. Además es posible especificar qué parte de la información se desea recibir, o filtrar servicios ya conocidos (indicando sus UUIDs). Esta operación puede aparecer tanto en mensajes *unicast* dirigidos a un DA o un SA, o en mensajes *multicast* que recibirán todos los SAs del grupo de trabajo.

Service Reply. Este mensaje sirve de respuesta a la petición anterior, y contiene la información de los servicios seleccionados por la consulta, incluyendo su tiempo de validez máximo (`lifetime`), para que el UA pueda almacenarlo temporalmente en su cache. Además, en caso de que el Tipo de Servicio consultado tenga asociado una política de reparto de carga, el DA ordenará las entradas adecuadamente, seleccionando las mejores.

Service Advertisement. Esta operación es muy similar al `Service Reply` anterior, puesto que incluye información de servicios, así como su tiempo de validez. Sin embargo en este caso, los SAs o DAs envían este mensaje proactivamente para anunciar servicios importantes. Por ejemplo los propios DAs, que deben anunciarse periódicamente para que el resto de los Agentes XSDF conozcan su información de servicio, lo que permite reducir el número de consultas de localización de DAs, o borrar de la caches de los UAs la información de un servicio inaccesible, poniendo su `lifetime` a cero.

Realm Request. Esta operación es opcional y permite conocer a qué Reino pertenece un Agente XSDF. Los UAs pueden utilizar este mecanismo al inicializarse para descubrir a que Reino pertenecen, consultando al DA que tengan pre-configurado, por ejemplo mediante DHCP.

Realm Reply. Esta es la respuesta a la petición anterior, y contiene el dominio y los grupos de trabajo a los que pertenece el Agente XSDF que la ha enviado.

Service Type Request. Esta operación también es opcional y, a diferencia del `Service Request`, no solicita información sobre las instancias de un

servicio, sino que permite conocer el Tipo de todos los servicios registrados en el SA o DA consultado.

Service Type Reply. Esta respuesta contiene la lista de todos los Tipos de Servicio solicitados mediante la operación anterior. Gracias a esta transacción un administrador de red puede conocer todos los servicios que ofrece su red, primero preguntando por los Tipos conocidos y luego descubriendo todos los servicios particulares de cada uno.

Redirect Reply. Esta operación puede emplearse como respuesta a cualquiera de las consultas XSLP, e indica que dicha petición debería redirigirse al Agente o Agentes XSDF especificados. Tal como se explicará en el apartado 6.4.1, esta operación puede emplearse para implementar políticas de Reparto de Carga Global entre múltiples sedes.

6.3.2. *eXtensible Service Registration Protocol* (XSRP)

Este protocolo sólo es necesario en redes donde se hayan desplegado Agentes de Directorio, puesto que se emplea para que los Agentes de Servicio registren a sus servicios en el DA, y luego para que actualicen periódicamente su información. Cuando un servicio deja de estar disponible, su SA "hogar" también puede emplear XSRP para de-registrarlo del DA, aunque, para ofrecer un funcionamiento robusto, los servicios que no sean actualizados por su SA (por ejemplo si este está inaccesible) también son borrados del repositorio central. Por tanto, XSRP [242] simplemente define tres operaciones de registro y sus respectivas confirmaciones:

Service Registration. Esta operación se emplea para registrar por primera vez toda la información de un servicio en un DA del Reino al que pertenece. Además, es posible indicar que Políticas de Selección debe aplicar el DA a los servicios de ese Tipo, para ordenarlos en las respuestas a los clientes según su idoneidad.

Service Registration Acknowledgement. Esta respuesta del DA es la confirmación de que el servicio indicado ha sido registrado correctamente en el DA. Además, indica al SA cuándo debe actualizarse dicha información (`minLife-maxLife`) para que el servicio se mantenga activo.

Update Service. Esta petición debe ser enviada periódicamente por los SA para actualizar la información de sus servicios (típicamente el Estado del

Servicio), y así evitar que sean borrados del repositorio central.

Update Service Acknowledgement. Esta operación sirve de confirmación a la actualización de información de servicio anterior y, al igual que el `Service Registration Acknowledgement`, indica al SA cual es el periodo válido de actualización.

Service Deregistration. Esta petición indica al DA que debe borrar el servicio especificado de todos los grupos de trabajo donde fue registrado, debido a que el servicio va a dejar de estar disponible permanentemente. Si la desconexión es temporal (e.g. si el servicio está saturado y no puede aceptar más peticiones) basta con actualizar su Estado con `resources=0`.

Service Deregistration Acknowledgement. Esta es la respuesta a la operación de borrado de servicios anterior.

6.3.3. *eXtensible Service Subscription Protocol* (XSSP)

Este protocolo es opcional y permite a los Agentes XSDF suscribirse a cambios en la información de servicios en los que estén interesados, de forma que no tengan que estar consultando continuamente la información de esos servicios, sino que, una vez suscritos a ella, recibirán un evento cada vez que dicha información cambie. Sin embargo XSSP tan sólo es un protocolo de suscripción, la notificación de estos eventos puede realizarse mediante cualquier otro mecanismo. Por tanto XSSP se limita a suscribir "canales" de notificación de eventos a la información de grupo de servicios.

La gestión de las suscripciones XSSP [243] es muy parecida al registro de servicios y por tanto, tiene un conjunto de operaciones similares a las definidas en XSRP:

Subscribe Service. Esta petición registra en el Agente XSDF destino un canal de notificaciones al que deben enviarse los eventos que afecten a: el Reino, el Tipo de Servicio o el servicio individual que se especifique en la petición. Además es posible indicar en qué tipo de eventos se está interesado: Registro de nuevos servicios, Actualización del servicio (distinguiendo si se actualiza información permanente o sólo su Estado), o cuando se De-registra algún servicio de los indicados.

Subscribe Service Acknowledgement. Esta respuesta es la confirmación de que el servicio de notificación anterior ha sido asociado a la información a

la que se desea estar subscrito. Además, indica cuando debe actualizarse dicha subscripción mediante el mecanismo de `minLife-maxLife`.

Update Subscription. Esta petición debe ser enviada periódicamente para mantener activa la subscripción registrada previamente. De lo contrario, al igual que ocurría con XSRP, el registro se borrará pasado un tiempo.

Update Subscription Acknowledgement. Esta operación sirve de confirmación a la actualización anterior y, al igual que el `Subscribe Service Acknowledgement`, indica cual es el intervalo válido para enviar la próxima actualización de la subscripción.

Unsubscribe Service. Esta petición indica al DA que debe borrar la suscripción indicada y desactivar el envío de eventos al servicio de notificación asociado a la misma.

Unsubscribe Service Acknowledgement. Esta es la respuesta a la operación de de-subscripción anterior.

En cuanto a los servicios de notificación de eventos, uno de los protocolos soportados es el propio XSLP, de modo que los cambios en la información del servicio generarán mensajes de `Service Advertisement` para los Agentes XSDF suscritos. Aunque cualquier Agente XSDF puede suscribirse a la información de servicios de un SA o DA, el uso más habitual de este protocolo será la sincronización de los DAs que gestionan el repositorio de servicios de un Reino común, empleando XSTP como protocolo de notificación.

6.3.4. *eXtensible Service Transfer Protocol* (XSTP)

El principal objetivo del protocolo XSTP es mantener sincronizada la información de servicios de todos los DAs del mismo Reino. Por ejemplo permite que un DA que está inicializándose pueda obtener de otro DA activo toda la información de servicios conocida, para que pueda empezar a formar parte del repositorio de servicios.

Tal como se comentó en el apartado anterior, además de XSLP, XSTP también puede emplearse como un servicio de notificación de eventos XSSP para que todos los DAs estén subscritos entre sí al Reino común. De este modo todas las operaciones XSRP de registro o actualización que reciba cualquiera de ellos puedan aplicarse a todos simultáneamente, por ejemplo empleando *multicast*.

Este protocolo [244] tan sólo tiene tres operaciones:

Service Transfer Request. Esta operación solicita al DA destino toda la información de servicios que tenga disponible de un Reino determinado. Además, también es posible especificar un *timestamp*, de forma que sólo se envíen los servicios registrados después de ese instante. Esta opción permite re-sincronizar rápidamente el repositorio si el DA se ha desconectado temporalmente, o simplemente puede realizarse esta operación periódicamente para garantizar la consistencia de la información de servicios.

Service Transfer Reply. Esta es la respuesta a la operación de transferencia anterior, y contiene la información sobre todos los servicios solicitados, además de otros datos asociados al registro, como por ejemplo las Políticas de Selección en DA. Todos estos datos deben enviarse a través la conexión TCP/SCTP por la que se recibió la petición `Service Transfer Request` ya que, a diferencia del resto de operaciones y protocolos de XSDF, esta transacción no emplea UDP como protocolo de transporte, ya que puede requerir que se transfiera una gran cantidad de información.

Service Notification. Esta operación encapsula un mensaje XSRP junto a cualquier otra información relevante acerca del servicio actualizado con XSRP. Por tanto esta operación se emplea para notificar al resto de los DAs que se ha procesado una petición que afecta al registro de servicios común, al que estarán subscritos mediante XSSP.

6.4. Escenarios de Aplicación

Como XSDF extiende la arquitectura de SLP, también hereda sus propiedades de escalabilidad, y por lo tanto puede emplearse en un gran número de escenarios, desde LANs sin administración, a una Intranet corporativa. Cada tipo de escenario requerirá un subconjunto de Agentes XSDF y protocolos.

El escenario más sencillo es una única LAN que tiene muy poca o ninguna infraestructura fija (i.e. sin servidor DNS), como por ejemplo una pequeña oficina que sólo tiene ordenadores cliente que comparten ficheros e impresoras. En ese caso, para descubrir servicios mediante XSDF tan sólo se necesitan Agentes de Usuario y de Servicio, que pertenecerían al grupo de trabajo por defecto `DEFAULT`. Por lo tanto, se podría emplear el protocolo XSLP con búsquedas *multicast/broadcast*, y si se encuentran varios servicios, el proceso de selección se

realizaría en el UA, basado en la información de servicio proporcionada por los SAs.

El segundo escenario podría ser una red formada por varias LANs, donde las búsquedas *multicast* son demasiado ineficientes. En ese caso debería desplegarse un Agente de Directorio, para que los Agentes de Servicio registren sus servicios en el DA empleando XSRP, y así los UAs podrían preguntar al DA sobre qué servicios se encuentran disponibles empleando *unicast*. Por tanto el proceso de selección del mejor servicio podría centralizarse en el DA.

Un escenario más complejo sería una red corporativa donde hay varios subgrupos de usuarios y servicios, y donde además el servicio de descubrimiento XSDF comienza a ser una aplicación crítica. En ese caso podrían definirse varios grupos de trabajo, de forma que los grupos con servicios críticos estarían gestionados por dos o más DAs. Por lo tanto sería necesario emplear todos los protocolos XSDF, ya que XSTP y XSSP permiten a todos los DAs con grupos comunes formar un único repositorio de servicios de cara a los UAs y SAs. El proceso de Selección podría realizarse, no sólo en los DAs, sino también en los UAs para reaccionar rápidamente a los fallos de los servidores (e.g. servidores de respaldo con menor prioridad).

Todos los escenarios anteriores, exceptuando las características de alta disponibilidad de los DAs, también podrían gestionarse empleando SLP ya que se reducen al descubrimiento de servicios dentro de una única organización. Sin embargo, gracias al concepto de Reino, que extiende los grupos de SLP para incluir el dominio DNS de la organización, XSDF también puede emplearse para anunciar servicios públicos en Internet. Por ejemplo, una granja de servidores web podría utilizar XSDF para realizar el reparto de carga. Para ello bastaría con desplegar DAs públicos, que gestionarían los servicios del grupo `PUBLIC`, y luego anunciar esos DAs en el servidor DNS de la organización mediante Registros `SRV` [96] (en SLP se definió un mecanismo experimental similar [277] aunque nunca fue adoptado como parte de la especificación de SLPv2).

En este escenario, un UA que desease acceder a un servicio del dominio `acme.org`, primero pediría el Registro `SRV` del DNS `_xslp._udp.acme.org`. Este registro contendría la lista de los DAs públicos de la organización, de forma que el UA podría preguntarles, vía XSLP, por el servicio deseado (i.e. `fax`) del grupo `PUBLIC` remoto. Por último, el DA devolvería la lista de servidores disponibles y, si por ejemplo se emplease una política de reparto de carga *Round-Robin*, el DA la iría rotando para cada petición recibida.

6.4.1. Reparto de Carga Global con XSDF

Otro escenario de aplicación de XSDF especialmente interesante es el Reparto de Carga Global, ya que el mecanismo para el descubrimiento remoto de servicios puede extenderse fácilmente para soportar reparto de carga global entre varias sedes, sin las limitaciones que tienen las técnicas actuales.

Para explicar cómo puede extenderse el mecanismo de localización de servicios remotos de XSDF para implementar una política de Reparto de Carga Global nos basaremos en el ejemplo representado en la figura 6.6.

Tal como se comentó en el apartado anterior, para localizar a los DAs públicos del sitio remoto es necesario consultar un Registro `SRV` del dominio DNS remoto (`_xslp._udp.acme.org`).

Una vez localizado la lista de DAs públicos, el UA elegiría [96] uno de ellos y le enviaría una consulta XSLP solicitando el Tipo de Servicio al que se quiere acceder (`ftp`). En el caso normal, el DA le respondería con la lista de los servidores locales que ofrecen ese servicio, de modo que el UA podría acceder a cualquiera de ellos.

Sin embargo, como el protocolo XSLP permite que un DA indique a un UA que la consulta debe redirigirse a otro DA, el primer DA, en lugar de responder con la lista de sus servidores, podría enviar una operación `Redirect Reply` al UA, con la lista de los DAs de las sedes de réplica y la política de selección Más Cercano.

Nótese que, al contrario que otras técnicas de Reparto de Carga Global basadas en consultas DNS, las peticiones XSLP pueden incluir información sobre el servicio solicitado, como el Tipo de Servicio o el *path* de la URL, por lo que los DAs pueden realizar un proceso de selección de sede basado en el contenido de la petición.

Por tanto, cuando reciba la re-dirección el UA debería emplear alguna técnica para calcular el RTT a todos los DAs y así seleccionar la sede más cercana. Un mecanismo muy sencillo, aunque sorprendentemente eficaz [78], consistiría en enviar la consulta XSLP simultáneamente a los DAs de todas las sedes, y seleccionar la primera respuesta. De este modo la latencia del proceso de descubrimiento de servicios se reduciría al mínimo (un RTT a cada DA consultado) y, como las consultas XSLP son mensajes muy pequeños, tampoco se consumirían demasiados recursos de red.

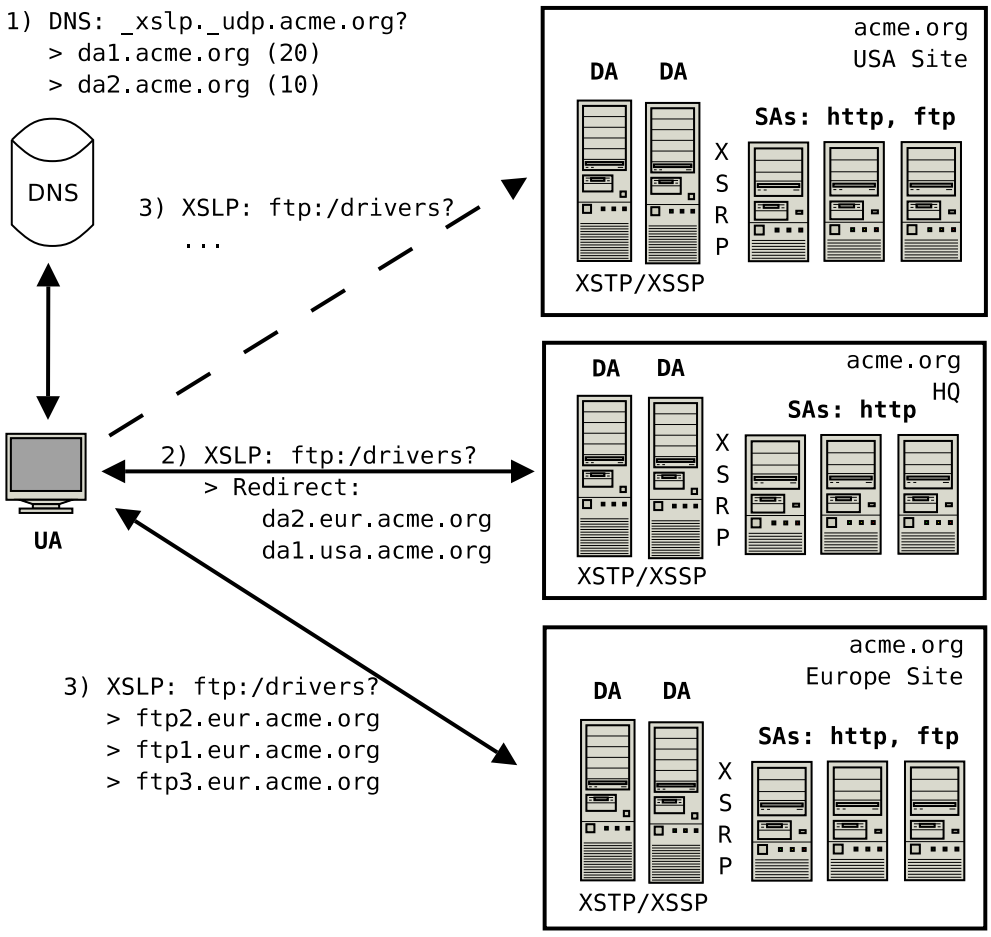


Figura 6.6: Reparto de Carga Global con XSDF

Además, a pesar de su simplicidad, esta técnica permite elegir la sede más cercana de una forma mucho más exacta que los mecanismos actuales de Reparto de Carga Global basados en el DNS, que calculan la distancia al cliente mediante el envío de *pings* o la consulta de la tablas de rutas BGP [44]. Por el contrario, en este caso se está estimando directamente el RTT entre todas las sedes y el cliente final, en lugar de la distancia entre el servidor DNS del Proveedor de Servicios y el servidor DNS del cliente, que puede estar bastante alejado [220] del mismo.

Volviendo al ejemplo, la respuesta del segundo DA ya contendría la lista de los servidores de su sede que ofrecen el servicio, que además podrían estar ordenados por carga, por lo que, además de realizar un Reparto de Carga Global entre sedes, también se aplicaría una política de reparto de carga local dentro de la sede seleccionada.

6.4.2. XSDF dentro de la arquitectura SARA

Aunque XSDF ha sido diseñado como una solución general para los problemas de descubrimiento de servicios y reparto de carga, obviamente XSDF también se ajusta a los requisitos de descubrimiento de servicios y de reparto de carga de la arquitectura SARA.

En particular, en SARA la verdadera utilidad de XSDF viene de su capacidad para soportar múltiples políticas de reparto de carga en el heterogéneo *cluster* de asistentes, aunque el descubrimiento de servicios también puede resultar muy útil para simplificar la administración del *cluster*. Por ejemplo, gracias a XSLP los asistentes pueden descubrir si hay más asistentes ejecutando alguna de sus aplicaciones de red, o si él es el asistente designado y debe ser el encargado de configurar el plano de datos del *router*.

Recordando la clasificación de las técnicas de reparto de carga en SARA definida en el capítulo 4, XSDF se empleará para coordinar los asistentes que ejecutan servicios de red con direccionamiento explícito (e.g. redes superpuestas), o aquellos donde haya un número reducido de paquetes de señalización y, por tanto, sea posible procesarlos en el plano de control del *router*.

De esto modo, como es el *router* el que implementa las políticas de reparto de carga, lo razonable sería instanciar un Agente de Directorio directamente en el plano de control del *router* para que éste siempre tenga disponible la

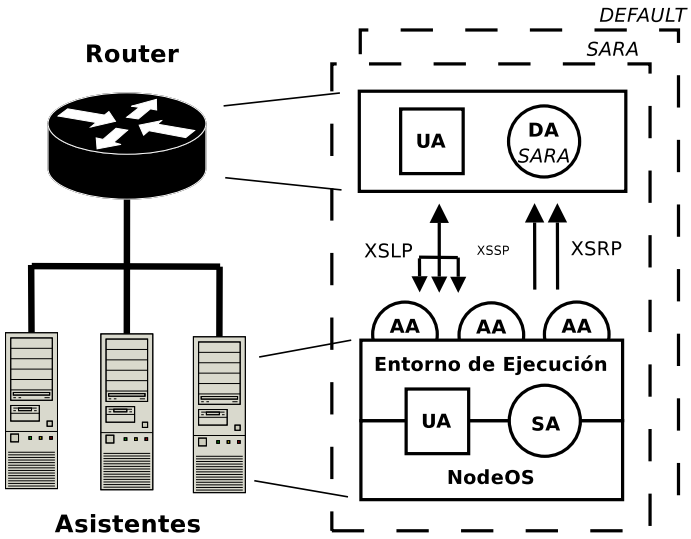


Figura 6.7: XSDF dentro de la arquitectura SARA.

información de todos los servicios de red que ejecutan en el *cluster*, así como el estado actualizado de todos los asistentes que los ejecutan. De esta forma sería posible elegir inmediatamente al mejor asistente al que debe desviarse el paquete de señalización de un servicio de red determinado.

Sin embargo, si la CPU central del *router* no tiene la capacidad de procesamiento suficiente para ejecutar un Agente de Directorio, siempre sería posible delegar esta función a uno de los asistentes, y usar XSLP para que el *router* pueda obtener la información de servicio que necesita.

En cualquier caso, para aquellos servicios de red donde la carga de los asistentes puede variar rápidamente y no es posible fiarse de las actualizaciones periódicas de estado, el *router* podría enviar consultas XSLP *multicast*, para obtener exactamente el estado actual de los asistentes.

La figura 6.7 muestra esquemáticamente la integración de SARA y XSDF. Los asistentes utilizarán Agentes de Servicio y de Usuario, mientras que el *router* implementará un Agente de Directorio y de Usuario. En el caso general existirán dos Grupos de trabajo, uno sin gestionar (`DEFAULT`), y otro gestionado por el DA del *router* (`SARA`). Las aplicaciones de red que pertenezcan al Grupo gestionado utilizarán sus SAs para registrar con XSRP su información de estado en el DA del *router*, indicando la política de selección más adecuada para su servicio de red.

Por otro lado, el grupo no-gestionado sólo se empleará para los servicios de

red donde es esencial conocer el estado real de los asistentes para asignar cada sesión de datos. En ese caso, cuando llegue un paquete de señalización al plano de datos del *router* este realizará una consulta XSLP *multicast* para escoger al mejor asistente en base a las respuestas que reciba.

Sin embargo, en la mayor parte de los casos, el Agente de Directorio del *router* dispondrá de información lo suficientemente actualizada para saber cual es el mejor asistente que puede procesar los paquetes de señalización que recibe, aplicando la política de reparto de carga registrada en el DA, que será la más adecuada para el servicio de red solicitado.

Opcionalmente, los asistentes podrían utilizar XSLP/XSSP para coordinarse entre sí, y así descubrir a los demás asistentes que ejecutan el mismo servicio de red (e.g. con el Tipo de Servicio "`sara-aa-555`") y suscribirse al Asistente designado, para poder sustituirlo en caso de que éste falle.

6.5. Identificadores de servicios: URIs Anidadas

Un aspecto muy importante a tener en cuenta en el despliegue de XSDF, o en general de cualquier mecanismo de descubrimiento de servicios alternativo al DNS, es su integración dentro de las aplicaciones de usuario.

Obviamente, la solución más directa consistiría en que la aplicación utilizase explícitamente el API de XSLP para descubrir servicios. Sin embargo este mecanismo requeriría modificar todas las aplicaciones, y además para coexistir con otros protocolos de descubrimiento como SLP o el propio DNS, necesitaría que la aplicación estuviese preparada para utilizar todos los métodos y permitir que el usuario final escogiese el mecanismo a utilizar.

Por tanto, parece conveniente definir una técnica común para todas las aplicaciones que permitiese seleccionar fácilmente el mecanismo de descubrimiento de servicios que se desea utilizar.

Una alternativa interesante consistiría en emplear los *Uniform Resource Identifiers* (URIs) [25] que constituyen los identificadores globales de servicio más habituales en Internet, ya que su sintaxis (figura 6.8) permite ocultar las complejidades subyacentes, como las direcciones IP, los puertos, o las consultas al DNS. Por tanto, las URIs serían un mecanismo ideal para seleccionar el mecanismo de descubrimiento de servicios que se desea emplear.

```
URI = scheme ":" authority "/" path ["?" query]["#" fragment]

authority = [userinfo "@"] host [":" port]

host = IP-literal/IPv4Address/reg-name

IP-literal = "[" (IPv6Address/IPvFuture) "]"

IPvFuture = "v" 1*HEXDIGIT "." 1*(unreserved/sub-delims/":")

reg-name = *(unreserved/pct-encoded/sub-delims)

port = *DIGIT
```

Figura 6.8: Sintaxis Genérica de las URIs

Sin embargo, aunque la especificación actual de las URIs permite definir futuros formatos de direcciones IP mediante el prefijo `v?.`, hoy en día la localización de un servicio tan sólo puede especificarse mediante la dirección IP o el nombre DNS del servidor que lo ofrece. Por tanto, como las URIs dependen del DNS como el único mecanismo de resolución de nombres permitido, los usuarios no pueden emplear las URIs para seleccionar qué mecanismo de descubrimiento de servicios desea utilizar, ni XSDF ni cualquier otro que pueda definirse en el futuro.

Otra limitación conocida [101] de la URIs es la identificación de la capa de transporte que se desea utilizar, ya que permite indicar un número de puerto pero, curiosamente, no el protocolo de transporte en sí. Por tanto, al igual que en el caso anterior, cada aplicación está ligada a un único protocolo de transporte, lo que complicará enormemente la migración de las aplicaciones actuales a los nuevos protocolos de transporte, como el *Datagram Congestion Control Protocol* (DCCP) o el *Stream Control Transmission Protocol* (SCTP).

Una solución a este problema podría consistir en definir un nuevo *scheme* de URI cada vez que se desee soportar otro protocolo de transporte o mecanismo de resolución de nombres. Sin embargo esta alternativa obligaría a modificar todas las aplicaciones existentes para reconocer los nuevos *schemes*, y además la combinación de los protocolos de transporte y de los mecanismos de descubrimiento de servicios generaría un gran número de *schemes*: `http-xslp-tcp`, `http-upnp-tcp`, `http-srv-tcp`, `http-xslp-sctp`, ...

Como el mecanismo de resolución de nombres y el protocolo de transporte que se desea emplear son elecciones ortogonales, es razonable permitir que

cada una pueda realizarse independientemente, y de este modo se resolvería el segundo problema. Por tanto otra solución alternativa podría consistir en definir opciones estándares para la sección *query*, lo que permitiría URIs como: `lpr://printer:515/queue?resolv=XSLP&proto=SCTP`

Sin embargo este mecanismo también necesita que las aplicaciones actuales se modifiquen para soportar las nuevas opciones, y lo que es peor, también sería necesario re-definir los *schemes* que no incluyen una sección de *query*, puesto que es un elemento opcional en la sintaxis de las URIs.

La mayor ventaja de las soluciones anteriores es que no requieren modificar la sintaxis genérica de las URIs, ya que se basan en elementos extensibles de la especificación. Sin embargo la manera más elegante para solventar las limitaciones de las direcciones de transporte parece utilizar la propia sección *authority*, en particular los componentes *host* y *port*.

Lamentablemente, la sección de *port* no es nada extensible, mientras que la parte de *host* sólo puede extenderse para incluir representaciones alternativas de direcciones IP entre corchetes, como las IPv6, o incluso formatos futuros.

Por tanto es necesario modificar la definición actual de las URIs para añadir otros mecanismos de resolución de nombres, o identificadores del protocolo de transporte en la sección *authority*.

Sin embargo, sería muy deseable que la nueva sintaxis fuese compatible con el resto de las secciones de la URI, o incluso con la sintaxis previa. Además, siguiendo con la filosofía de las URIs, debería ser fácil de leer y escribir por los usuarios. Y por último, el nuevo formato debería ser extensible para que pueda soportar nuevos mecanismos de resolución de nombres y protocolos de transporte definitivamente.

Entonces, ¿por qué no emplear las propias URIs? Exceptuando estas limitaciones son totalmente extensibles, su sintaxis es popular entre los usuarios, y con mecanismos de encapsulación adecuados deberían ser totalmente compatibles con la sintaxis anterior, ya que, por definición, no contienen caracteres reservados.

Por ejemplo, podrían emplearse las siguientes URIs para identificar direcciones IPv4 (`ipv4:10.117.139.166`), IPv6 (`ipv6:2001:DB8::2c0:9fff:fe18:31d4`) o incluso IPX (`ipx:00000001:00081A0D01C2`), nombres de DNS (`dns:www.example.com`) [129], servicios XSDF (`xslp:printer:DEFAULT`), o puertos de transporte TCP

```
URI = scheme ":" authority "/" path ["?" query]["#" fragment]

authority = [userinfo "@"] host [":" port]

host = IPv4Address/reg-name/nested-uri

reg-name = *(unreserved/pct-encoded/sub-delims)

nested-uri = "[" (IPv4uri/IPv6uri/dns-uri/slp-uri/xsdf-uri) "]"

port = *DIGIT / "[" ("tcp"/"udp"/"sctp"/"dccp") ":" *DIGIT "]"
```

Figura 6.9: Sintaxis Genérica de URIs Anidadas

```
http://www.example.com:80/index.html#top
http://10.117.139.166:80/index.html#top

http://[dns:www.example.com]:[tcp:80]/index.html#top
http://[ipv4:10.117.139.166]:[tcp:80]/index.html#top
http://[ipv6:2001:DB8::2c0:9fff:fe18:31d4]:[tcp:80]/index.html#top
```

Figura 6.10: Ejemplos sencillos de URIs anidadas

(`tcp:80`) o SCTP (`sctp:80`).

De este modo, cuando fuese necesario, estas URIs de localización podrían emplearse para definir las secciones *host* y/o *port* de la URI, aunque delimitadas entre corchetes ("[","]") para identificarlas como URIs anidadas.

Esta sintaxis (figura 6.9) es compatible hacia atrás con la previa, exceptuando quizá las direcciones IPv6, que también se delimitan entre corchetes. Sin embargo esto no es un problema demasiado grave puesto que, aunque muchas implementaciones soportan direcciones IPv6, el número de URIs IPv6 debería ser muy bajo, y la transición a la nueva sintaxis es extremadamente sencilla.

La figura 6.10 muestra varios ejemplos de la nueva sintaxis, incluyendo URIs convencionales que siguen siendo válidas, e incluso preferibles a las anidadas, puesto que son más cortas.

Por tanto, la utilidad real de las URIs anidadas viene de los usos avanzados que permite, que no son posibles con las URIs convencionales. Por ejemplo la figura 6.11 muestra tres URIs que apuntan a una impresora. La primera es casi una URI normal, basada en el nombre DNS de la impresora, pero especifica que debe emplearse el protocolo de transporte SCTP, en lugar de TCP.

```
lpr://printer1.example.com:[sctp:515]/queue

lpr://[dns:_lpr._tcp.example.com?type=SRV]/queue

lpr://[xslp:printer@example.com:PUBLIC]/queue
```

Figura 6.11: URIs Anidadas para el Descubrimiento de Servicios

```
telnet://[ipv6:fe80::2c0:9fff:fe18:31d4#eth0]

http://[dns:www.example.com?type=AAAA]/index.html

ftp://[dns://192.168.1.31:53/_drivers._sctp.acme.org?type=SRV]:
      [sctp:21]/lp2000
```

Figura 6.12: Ejemplos avanzados de URIs anidadas

El segundo se parece al tercer ejemplo de la figura 6.10, pero indica que el DNS debe emplearse para localizar las impresoras en el dominio `example.com`. Para ello basta con consultar el Registro `SRV` [96], que contendrá una lista con los nombres de las impresoras públicas, asociados con un peso para repartir la carga entre ellas.

La tercera URI emplea XSDF para localizar las impresoras públicas en el mismo dominio. Pero, a diferencia del método anterior, como XSLP es un protocolo dinámico, podemos estar seguros que las impresoras localizadas se encuentran realmente activas.

La figura 6.12 muestra más ejemplos avanzados del uso de URIs anidadas. Por ejemplo, ahora las URIs IPv6 pueden especificar el ámbito de las direcciones, que en el caso de las direcciones de enlace, es el identificador de la interfaz correspondiente (`eth0`).

La resolución de nombres mediante DNS también puede beneficiarse de la flexibilidad de las URIs anidadas, ya que permite añadir opciones a la consulta DNS. Por ejemplo seleccionar el tipo de Registro que se desea obtener, lo que en el segundo caso de la figura fuerza que se usen direcciones IPv6. El último ejemplo se deja como ejercicio para el lector.

Por tanto aunque la flexibilidad de las URIs es por sí misma una gran ventaja sobre las otras alternativas, quizá el aspecto más interesante de este mecanismo es que no requiere que las aplicaciones tengan que modificarse para soportarlo.

Para ello, antes de nada es necesario entender cómo utilizan las aplicaciones las URIs para acceder a los servicios de red. Aunque inicialmente el código de las aplicaciones de red dependía en gran medida de las direcciones IPv4 y de la semántica del DNS como la función `gethostbyname()`, la introducción de IPv6 condujo a la definición de nuevas funciones de red que simplificasen el desarrollo de aplicaciones independientes del protocolo-IP.

Por tanto, a día de hoy, la función recomendada [222] para reemplazar la obsoleta `gethostbyname()` es `getaddrinfo()`, que permite ocultar los detalles de la resolución de nombres en una estructura opaca, e independiente de los protocolos de transporte y red utilizados (figura 6.13).

Además, como este mecanismo también integra la resolución del puerto de transporte, la librería de `getaddrinfo()` es capaz de procesar todos los campos de localización de la URI, y devolver toda la información necesaria para que las aplicaciones puedan establecer una sesión de transporte con el servidor. De este modo las aplicaciones no tienen que distinguir si la URI es un nombre DNS o una dirección IP, sino que simplemente pasan la cadena de texto a la librería de resolución de nombres.

Este comportamiento es esencial para que las aplicaciones puedan soportar URIs anidadas sin que tengan que ser modificadas. En su lugar tan sólo es necesario actualizar la librería `getaddrinfo()` del sistema para que sea capaz de comprender la sintaxis de las URIs anidadas y luego emplear el mecanismo de resolución de nombres adecuado para tratarlas.

Para ilustrar este mecanismo, la figura 6.14 muestra un sistema actualizado, que ejecuta el código de aplicación de la figura 6.13, y a la que se le pasa la siguiente URI anidada: `lpr://[xslp:printer]`

La aplicación empleará la librería de URIs para extraer las secciones de *host* (`"[xslp:printer]"`) y *port* (`NULL`) de la URI de entrada y se las pasará directamente a la función `getaddrinfo()`, como si fuera una URI convencional.

Como las URIs anidadas está delimitadas por corchetes, las librerías de URIs podrían identificar las sección *host* como una simple dirección IPv6. Sin embargo, una URI anidada en el elemento *port* podría ser considerado como un error, de modo que algunas librerías de análisis de URIs también deberían actualizarse a la nueva sintaxis.

La librería de resolución de nombres reconocería la URI anidada (o un simple

```
struct addrinfo {
  int     ai_flags;
  int     ai_family;
  int     ai_socktype;
  int     ai_protocol;
  size_t  ai_addrlen;
  struct sockaddr *ai_addr;
  char   *ai_canonname;
  struct addrinfo *ai_next;
};

int getaddrinfo (const char *node,
                 const char *service,
                 const struct addrinfo *hints,
                       struct addrinfo **res);

/* ... */


struct addrinfo hints, * res;
int error, sockfd;

memset(&hints, 0, sizeof(hints));
hints.ai->family   = AF_UNSPEC;
hints.ai->socktype = SOCK_STREAM;

error = getaddrinfo(SERVER, SERVICE, &hints, &res);
if (error != 0) {
    /* handle getaddrinfo error */
}

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sockfd < 0) {
    /* handle socket error */
}

error = connect(sockfd, res->ai_addr, res->ai_addrlen);
if (error < 0) {
    /* handle connect error */
}

/* ... */
```

Figura 6.13: Código recomendado para aplicaciones multi-protocolo

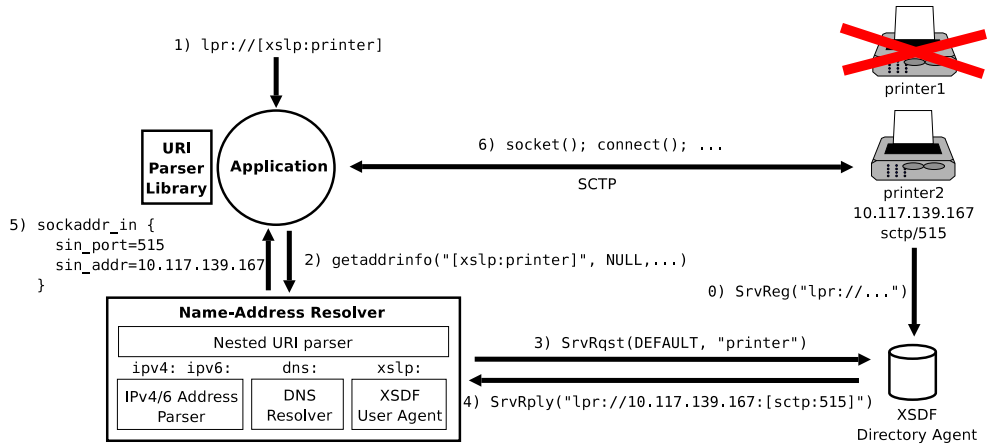


Figura 6.14: Comportamiento de las aplicaciones con URIs anidadas

nombre DNS, así que las URIs convencionales también servirían), y dependiendo de su *scheme* (`"xslp:"`), la URI de localización se resolvería con el mecanismo adecuado. En este caso se emplearía un Agente de Usuario XSDF, aunque si no se dispusiese del mecanismo adecuado, la función `getaddrinfo()` podría devolver el código de error `EAI_NONAME`.

El Agente de Usuario XSDF enviaría una operación XSLP de `ServiceRequest` al Agente de Directorio local (o en *multicast* a todos los Agentes de Servicio si este no estuviese presente), para descubrir todas las impresoras de la red. Como los Agentes de Servicio de las impresoras habrán registrado previamente la información de servicio en el Agente de Directorio, este será capaz de responder con la lista de las mejores impresoras disponibles, que tengan menos trabajos en cola.

Así, cuando la URI XSLP anidada se resuelva por completo, las direcciones de transporte resultantes se devolverían a la aplicación para que ésta pudiese acceder a la impresora seleccionada. Por tanto el mecanismo de descubrimiento de servicios es totalmente transparente para la aplicación, que obtendría exactamente el mismo resultado que con una URI convencional.

Además este mecanismo también simplifica el despliegue de los nuevos protocolos de transporte, ya que, por ejemplo en este caso podría emplearse SCTP en lugar de TCP transparentemente para la aplicación, ya que SCTP puede imitar [232] perfectamente el API de TCP (i.e. el mismo razonamiento podría aplicarse a DCCP/UDP), pero añadiendo capacidades avanzadas, como el soporte de *multihoming*.

6.6. Comparación cuantitativa XSDF vs. SLPv2

Este apartado compara la utilización de recursos de los protocolos XSDF comparados con SLPv2 (y alguna de sus variantes) evaluando varios escenarios mediante simulación.

Para ello se han desarrollado dos módulos para el simulador de eventos discretos OMNet++ [183]. El primero de ellos es una implementación del protocolo SLPv2, la extensión de *Attribute List* [100], la propuesta de `serviceid` [101] para servicios complejos, y el API de SLP [138]. El segundo implementa los tres tipos de agentes XSDF (UA, SA y DA), los protocolos XSLP y XSRP, y un API muy sencillo similar al de SLP. Los dos modelos están integrados en un entorno de red simplificado con estaciones cliente y servidores. Cada servidor tiene un Agente de Servicio asociado donde registra el servicio que proporciona, mientras que cada cliente solicita la localización de los servicios a su Agente de Usuario local.

La figura 6.15 muestra la suma de todas las tramas de nivel de enlace[10] (en Mega-Bytes) generadas por los diferentes protocolos de descubrimiento de servicios durante 8 horas de simulación sobre varios escenarios. El eje X muestra el número de Agentes de Usuario y Agentes de Servicio en la red (hay una relación 10:1 entre clientes y usuarios, por tanto una red con 22 estaciones tiene 2 SAs y 20 UAs). Los clientes localizan servicios cada 15 minutos, mientras que los servidores actualizan su información de servicio cada 60 segundos.

Como los escenarios no gestionados (`Unmanaged`) no incluyen ningún Agente de Directorio, no hay tráfico de actualización y las peticiones de servicio (`Service Request`) deben enviarse en paquetes *multicast*. Tal como puede observarse, la búsqueda SLPv2 básica no escala demasiado bien en redes grandes dado que son necesarias dos peticiones *multicast*, la primera (`SrvRqst`) para obtener la URI del servicio, y la segunda (`AttrRqst`) para conseguir los atributos del servicio. Como en XSLP, la extensión *Attribute List* de SLPv2 (`SLPv2+Ext`) permite que los UAs puedan conseguir tanto la URI del servicio como sus atributos con una única búsqueda *multicast*, por tanto ambos protocolos tienen una mejora significativa comparados con la búsqueda *multicast* básica de SLPv2.

Los escenarios gestionados (`Managed`) tiene una estación adicional con un Agente de Directorio que centraliza toda la información de servicios. En estos

[10] Un paquete *multicast* genera $N - 1$ tramas de nivel 2, siendo N el número total de estaciones de la red.

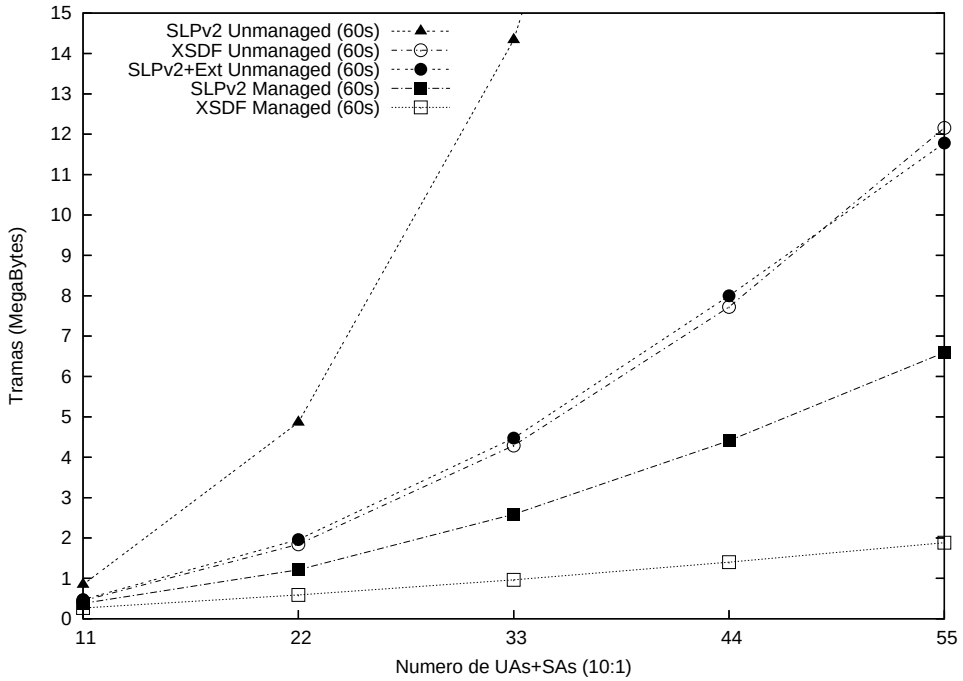


Figura 6.15: Utilización de recursos de red de XSDF y SLPv2

casos XSDF muestra una clara ventaja sobre SLPv2 debido a dos factores: la codificación binaria XBE32 es mucho más compacta que los atributos textuales de SLP (figuras 6.3, 6.3), pero especialmente debido al modelo de servicio de XSDF.

Mientras que los Agentes de Usuario SLP siempre descargan todos los datos del servicio, incluidos algunos que no son necesarios para su localización (e.g. `service-model`), los Agentes de XSDF pueden registrar la información en el DA (SAs) o guardarla en su cache (UAs) y luego limitarse a actualizar periódicamente el Estado del servicio, ya que este permite a los UAs saber si ha cambiado el resto de la información de su cache. De este modo, con SLPv2 el crecimiento del tráfico es exponencial con el número de servicios, mientras que con XSLP el crecimiento es lineal.

Aunque la extensión de *Attribute List* tiene muchas ventajas para la búsqueda *multicast*, en entornos gestionados puede convertirse en un problema si se emplea en las consultas *unicast* de servicios complejos, debido a las restricciones de MTU. Tanto los mensajes de SLPv2 como de XSDF deben codificarse dentro de un único datagrama UDP y, por tanto, el número de servicios en cada respuesta está limitado.

Esto no es un problema para SLPv2 básico puesto que el primer `SrvRply` tan sólo devuelve las URIs de los servicios, mientras que los atributos se envían en mensajes `AttrRply` diferentes. Sin embargo con la extensión de *Attribute List*, las URIs de los servicios se envían junto con sus atributos en un único mensaje, y por tanto el tamaño de la información del servicio puede ser un problema. Por ejemplo en el último escenario, SLPv2 básico y XSLP son capaces de obtener la información de los 5 servicios, mientras que SLPv2 con la extensión de *Attribute List* tan sólo es capaz de obtener la información de un servicio por petición.

Por tanto si se quisiese utilizar SLP o Rserpool para repartir la carga en escenarios con muchos servidores replicados, sería necesario emplear TCP/SCTP para descargarse todos los servicios, y luego elegir el mejor, con el consiguiente aumento de latencia y ancho de banda. Por el contrario, como XSDF permite implementar la política de reparto de carga en el DA, aunque la red tenga más de 5 servicios los UAs no necesitarán descargarse todos sino que el DA ya habrá seleccionado los 5 servicios menos cargados, y el UA podrá elegir directamente el mejor servidor disponible.

Capítulo 7

Conclusiones y Trabajos futuros

7.1. Conclusiones

Hoy en día existe una clara tendencia a incrementar las capacidades de procesamiento de los nodos de red para soportar cada vez más funcionalidades de seguridad, calidad de servicio, reparto de carga, etc. Por tanto no cabe duda de que las futuras arquitecturas de *routers* de altas prestaciones serán mucho más flexibles que las actuales, ya que estarán diseñadas para soportar numerosos servicios de red avanzados. Además, a la luz de las investigaciones en redes programables, es muy probable que los *routers* dejen de emplear diseños monolíticos y se tienda a arquitecturas modulares, basadas en capas y con interfaces bien definidas entre ellas. En particular la separación entre el plano de datos y el de control será cada vez más estricta e independiente de la plataforma subyacente, llegando a diferenciarse en entidades físicas separadas, por lo que tendrán que emplearse protocolos en la línea de GSMP -en el caso de los conmutadores- o ForCES -en los *routers*- para que puedan coordinarse y funcionar como un único sistema.

Sin embargo, hasta que los primeros *routers* totalmente programables lleguen a comercializarse, la gran mayoría de los nodos de red de alta velocidad desplegados serán plataformas optimizadas para el reenvío de paquetes, integradas verticalmente, y por tanto muy poco flexibles. Por esta razón es necesario idear mecanismos de transición que simplifiquen el desarrollo y el despliegue de nuevos protocolos y servicios de red sobre los nodos de red actuales.

Esta es la motivación principal de *Simple Assistant-Router Architecture*

(SARA), la arquitectura distribuida de nodo de red programable propuesta en esta tesis. Un nodo de red SARA está compuesto por un *router* de altas prestaciones "legado" y uno o más asistentes en los que se delega el procesamiento avanzado de paquetes. De este modo, el *router* se limita al encaminamiento de paquetes, mientras que los asistentes ofrecen una plataforma de computación independiente y basada en *hardware* de propósito general, lo que simplifica tanto el desarrollo como el despliegue de los nuevos servicios de red.

Por tanto, la arquitectura SARA puede englobarse dentro de la investigación en redes programables, abarcando características realistas del campo de las redes activas y la estandarización de interfaces abiertas. La utilización de un Entorno de Ejecución en lugar de una interfaz estática simplifica el despliegue dinámico de las aplicaciones de red. Sin embargo, la distribución de código en SARA es mucho más pragmática que la de los trabajos de redes activas, puesto que está basada en servidores de código fiable, autorizado por el administrador de red.

Otra característica de SARA que simplifica el despliegue de nuevos protocolos y servicios de red, es el desvío transparente de paquetes de señalización a los asistentes. De este modo los sistemas finales no tienen que estar pre-configurados con la localización de los nodos de la red que ejecutan el servicio deseado, sino que pueden localizar dinámicamente a los nodos programables en su camino al destino o, llegado el caso, que el servicio de red sea totalmente "transparente" para los sistemas finales. Para poder implementar eficientemente esta funcionalidad en los *routers* actuales, los paquetes de señalización pueden marcarse con una opción estándar de la cabecera IP, denominada *Router Alert*, mientras que para su desvío al asistente que debe procesarlos, tampoco es necesario modificar la arquitectura del router "legado" sino que los paquetes de señalización pueden reenviarse directamente en tramas de nivel 2, como si el asistente fuese el siguiente salto en su ruta al destino.

La viabilidad de esta arquitectura ha sido demostrada con el desarrollo de varios prototipos, incluido uno basado en un *router* comercial de altas prestaciones: el Ericsson-Telebit AXI462. Para ello tan sólo fue necesario realizar modificaciones mínimas al *software* de control estándar del *router* para poder adaptarlo a la arquitectura SARA.

Este primer prototipo se completaba con un asistente Linux que permitía: procesar los paquetes "activos" desviados y reenviarlos al destino, consultar el estado del *router* empleando SNMP, y cargar dinámicamente las aplicaciones de red, cuyo código podía descargarse de repositorios fiables bajo demanda.

Posteriormente se ha desarrollado un nuevo prototipo que es capaz de capturar flujos de datos transparentemente, lo que permite a un nodo de red SARA aplicar procesamiento avanzado (i.e. a cualquier nivel y con estado por flujo) al tráfico de aplicaciones convencionales, sin que sea necesario modificarlas para que puedan hacer uso del mismo.

A partir de este prototipo, se ha comprobado que la utilización de un gran número de reglas de filtrado puede afectar negativamente al desvío y/o la captura de flujos en *routers* sin soporte *hardware* (e.g. *router* Linux). Por tanto, en este tipo de *routers* es preferible definir reglas de desvío basadas en clases de servicio (e.g. trafico web), y luego distribuir los flujos individuales entre varios asistentes, de modo que cada uno de ellos utilice un número reducido de reglas para diferenciarlos.

Aunque los mecanismos básicos de SARA para el desvío de paquetes de señalización y la consulta del estado del *router* vía SNMP permiten desarrollar servicios de red sencillos, no cabe duda que para poder explotar todas las posibilidades de esta plataforma es necesario emplear mecanismos de cooperación *router*-asistente más avanzados, como por ejemplo el *Router-Assistant Protocol* (RAP), propuesto en este trabajo.

El protocolo RAP permite que una aplicación de red en un asistente pueda acceder a todas las funcionalidades del plano de datos del *router*, como si esta se ejecutase directamente sobre el plano de control del mismo. Obviamente, este mecanismo es mucho más intrusivo y requiere realizar varias modificaciones a la arquitectura del *router*. Por esta razón se ha optado, en la medida de lo posible, por evitar desarrollos *ad-hoc* y emplear estándares para facilitar su aceptación por parte de los fabricantes de *routers*.

Por tanto, RAP ha sido diseñado para permitir la integración de SARA dentro de la arquitectura definida por el grupo de trabajo del IETF ForCES (*Forwarding and Control Element Separation*). Según la nomenclatura ForCES, el plano de datos del *router* está formado por uno o más *Forwarding Elements* (FE), gestionados por un *Control Element* (CE) mediante un protocolo de control ForCES. Pues bien, RAP permite que los asistentes externos envíen comandos ForCES al CE del *router* de modo que éste, a modo de *proxy*, los reenvíe a los FE, para que el plano de datos del *router* pueda ser configurado remotamente.

De este modo, el protocolo de control RAP permite la encapsulación de mensajes estándar de ForCES, de forma que cualquier *router* ForCES podría

implementar RAP fácilmente para extender sus capacidades de procesamiento mediante asistentes externos.

Además de las capacidades de configuración remotas, que incluyen la posibilidad de descubrir las funcionalidades del *router*, acceder a su estado en detalle, o suscribirse a los eventos internos, el protocolo RAP también puede emplearse para el desvío de paquetes entre el *router* y los asistentes. Mediante RAP es posible encapsular el contenido de los paquetes junto a la meta-información necesaria para procesarlos, como por ejemplo la interfaz del *router* por donde se recibió el paquete, o indicar su siguiente salto al re-inyectarlo en el *router*.

Para ello se ha definido el *eXtensible Binary Encoding (XBE32)*, un sistema de codificación de datos binarios, extensible, muy compacto, y alineado a 32 bits, lo que, a diferencia de otras técnicas de codificación como ASN.1 o XML, permite que pueda ser generado e interpretado a altas velocidades por el plano de datos de un *router*.

La última característica fundamental de la arquitectura SARA es la capacidad de emplear un *cluster* de asistentes para poder soportar servicios de red con cientos o miles de usuarios. Por tanto, para aprovechar toda la capacidad de procesamiento del *cluster* es imprescindible emplear unas políticas de reparto de carga adecuadas para los diferentes servicios de red que pueden estar ejecutando simultáneamente en los asistentes.

A pesar de la enorme heterogeneidad de los servicios de red, si tenemos en cuenta sus necesidades de reparto de carga, éstos pueden clasificarse en dos grandes grupos: reparto de carga por paquete y reparto de carga por sesión de datos. En el primer caso la velocidad de los algoritmos de selección es el factor clave, puesto que es necesario elegir un asistente por cada paquete recibido, mientras que en el segundo caso es preferible aplicar políticas de reparto de carga más sofisticadas para seleccionar realmente el mejor asistente disponible. Por tanto, dadas las necesidades de rendimiento, el primer tipo de reparto de carga debe implementarse en el plano de datos del *router*, mientras que el segundo puede realizarse en el plano de control, que es mucho más flexible.

Analizando el problema del reparto de carga a velocidad de línea, es posible distinguir a su vez dos tipos de servicios de red: aquellos que necesitan almacenar estado por cada flujo procesado (e.g. NAT), y los que pueden procesar los paquetes independientemente (e.g. *Firewall* estático) y por tanto no necesitan estado por flujo. Obviamente el segundo caso es el más sencillo, ya que

es posible emplear algoritmos como *Weighted Round-Robin* para distribuir los paquetes entre todos los asistentes disponibles rápidamente.

Sin embargo, en el caso de los servicios de red con estado por flujo la solución no es tan simple, ya que el algoritmo de selección debe ser capaz de repartir la carga entre los asistentes, pero asignando todos los paquetes de un flujo siempre al mismo asistente, y además realizar esta operación a altas velocidades. Como la utilización de una tabla de flujos requeriría soporte *hardware* y no sería escalable a decenas de miles de flujos, es preferible emplear técnicas de *hashing* que permiten asignar flujos a asistentes, sin necesidad de almacenar estado por flujo en el *router*.

Por otro lado, las técnicas de *hashing* básicas (asignar un paquete a un asistente en función del identificador de su flujo) no toleran cambios en el número de asistentes del *cluster*. Por ejemplo, en caso de que alguno falle la mayoría de los flujos se re-asignarían a un asistente diferente, con la consecuente perdida de estado. Una solución a este problema es el algoritmo de *Robust Hash*, que permitiría la asignación persistente de flujos a asistentes, incluso cuando cambia el número de asistentes, aunque a costa de evaluar una función *hash* por cada asistente activo del *cluster* (complejidad $O(n)$). Sin embargo, como las funciones de *hash* emplean operaciones relativamente complejas, la necesidad de realizar n *hashes* por cada paquete puede ser excesiva para la capacidad de procesamiento de un *router* actual.

Por esta razón, en esta tesis se han ideado y evaluado dos nuevas técnicas de *Fast Robust Hashing*, que reducen el número de operaciones *hash* a realizar por paquete, manteniendo la capacidad de asignación persistente del *Robust Hashing*. La primera técnica (*Small Robust Hash*) tiene complejidad constante ($O(1)$) ya que para cualquier paquete basta con realizar una única operación de *hash*. Esta técnica tiene la limitación de que sólo puede aplicarse a *clusters* de tamaño reducido, ya que la complejidad en memoria es $O(n!)$.

Por el contrario, la segunda técnica (*Big Robust Hash*) puede aplicarse a *clusters* muy grandes (sólo necesita $\frac{n(n+1)}{2}$ posiciones de memoria), aunque el número de operaciones *hash* a realizar por paquete depende del porcentaje de asistentes inactivos (k). En el caso más habitual, cuando todos los asistentes están disponibles, basta con realizar una operación de *hash*, mientras que si comienzan a fallar los asistentes el número medio de operaciones crece de forma logarítmica ($O(\ln \frac{n}{n-k})$).

Las simulaciones realizadas sobre capturas de tráfico real muestran que las técnicas de *Fast Robust Hashing* son capaces de repartir equitativamente la carga entre los asistentes del *cluster*, incluso ante eventos como la pérdida de un asistente. Además, requieren un número de operaciones *hash* significativamente menor que los algoritmos de *Robust Hash* "tradicionales".

Uno de los argumentos a favor de la técnica de *Robust Hash* tradicional es la capacidad de paralelización de los cálculos. Sin embargo, esto requeriría n unidades *hardware* de *hashing*, lo que encarecería el coste y limitaría la escalabilidad del *cluster*. Para poder comparar objetivamente los algoritmos se ha utilizado una arquitectura de procesador de red (*Network Processor*) real, el IXP1200, que dispone de una unidad de *hash* con *pipeline*. Basándose en los parámetros de esta arquitectura, como por ejemplo el retardo de acceso a memoria SRAM, esta tesis ha demostrado que los algoritmos de *Fast Robust Hash* reducen, en la mayoría de los casos, la latencia por paquete que requiere la técnica de *Robust Hashing* clásica, incluso cuando se dispone de unidad *hardware* segmentada capaz de realizar varias operaciones *hash* simultáneamente.

Por estas razones, se concluye que las técnicas de *Fast Robust Hashing* presentadas en esta tesis son ideales para ser utilizadas como algoritmos de reparto de carga persistente para el plano de datos de un *router* de altas prestaciones actual.

Dada la heterogeneidad de los servicios de red, es necesario que cada servicio pueda elegir la política de reparto de carga más adecuada a sus necesidades. En el caso del reparto de carga en el plano de datos del *router*, en principio no sería necesario definir ningún mecanismo de configuración nuevo, puesto que los asistentes pueden emplear RAP para configurar el módulo de reparto de carga del *router* con la política de selección elegida: *Round Robin* o *Fast Robust Hash*.

Por el contrario, los servicios de red con reparto de carga en el plano de control, al no estar tan limitados por el rendimiento del algoritmo, tendrán a su disposición un gran número de políticas de selección, mucho más complejas, pero que les permitirá elegir con mayor precisión el asistente adecuado para cada sesión de datos, por ejemplo el menos cargado. Por tanto, como RAP está centrado en el plano de datos, es necesario definir un mecanismo adicional para indicar al plano de control del *router* qué política de selección debe aplicarse a cada servicio de red, descubrir qué aplicaciones están ejecutando en cada asistente, conocer el estado del mismo, etc.

El mecanismo propuesto para realizar esta tarea es el *eXtensible Service Discovery Framework* (XSDF), un entorno de descubrimiento de servicios extensible que integra en un único proceso la localización dinámica de servicios y el reparto de carga entre los servidores de un *cluster*. Sin embargo XSDF no está limitado solamente a este escenario de aplicación, sino que puede aplicarse a redes LAN sin infraestructura, a grandes redes corporativas, o incluso permite implementar políticas de reparto de carga global en Internet.

XSDF integra las arquitecturas definidas por *Service Location Protocol* (SLP) y *Reliable Server Pooling* (Rserpool) en un único entorno que combina todas las funcionalidades de ambos protocolos, y además añade varias mejoras de diseño:

- Configuración Cero. En una red que soporte XSDF, un cliente no necesita tener pre-configurada la localización de ningún servicio, ni siquiera la del propio servicio XSDF. Una vez que obtenga conectividad IP (e.g. vía DHCP o con auto-configuración IPv4/IPv6) será capaz de descubrir cualquier servicio de red, como por ejemplo el servidor DNS local, el Agente de Directorio XSDF menos cargado, o todas las impresoras disponibles.

- Reparto de Carga. Aunque la selección del mejor servicio podría considerarse como un paso más del proceso de descubrimiento de servicios, ni SLP ni el resto de los protocolos de descubrimiento de servicios incluyen un mecanismo estándar para escoger al mejor servidor que proporciona un servicio. XSDF incluye un mecanismo de selección, parecido a Rserpool, que permite la utilización de políticas de reparto de carga habituales como *Round-Robin*, o el uso de servidores primarios y de respaldo. Este proceso de selección puede realizarse tanto en los Agentes de Usuario como, a diferencia de Rserpool, en los Agentes de Directorio XSDF.

- Modelo de Servicio Mejorado. XSDF identifica cada servicio con un UUID, y define un modelo de datos jerárquico y extensible para los servicios. Este modelo es mucho más flexible que los de SLP y Rserpool, lo que incluso le permite representar a los propios Agentes XSDF como un servicio más, y por tanto que también puedan beneficiarse del marco de trabajo XSDF para descubrir otros Agentes XSDF. Además, este modelo de servicios mejorado no implica mensajes más grandes ya que se emplea la codificación binaria XBE32, propuesta en esta tesis, y además es posible seleccionar qué tipo de información de servicio se desea.

- Alta Disponibilidad y Escalabilidad. Una vez desplegado, el descubrimiento de servicios puede convertirse en un componente crítico de la red. Por

tanto XSDF define un mecanismo, similar al protocolo ENRP de Rserpool, para construir un repositorio de servicios con alta disponibilidad, replicado en múltiples DAs. XSDF ha sido diseñado para ser, al menos, tan escalable como SLP, aunque además, XSDF permite el control centralizado para evitar avalanchas de tráfico debido a las actualizaciones periódicas del estado de los servicios.

- Descubrimiento de Servicios en Internet. A diferencia de SLP, que fue diseñado para localizar servicios en redes locales, XSDF también permite descubrir servicios de redes remotas, consultando sus Agentes de Directorio públicos. Para ello define el concepto de "Reino", que extiende los grupos de SLP con el Dominio DNS de la organización que lo proporciona. De esta forma es posible localizar a los DAs remotos consultando los Registros `SRV` del DNS. Además, XSDF permite la utilización de políticas de reparto de carga global, basadas en el contenido de las peticiones del cliente y en su distancia real con cada sede de la organización.
- Solución Extremo-a-Extremo. Como los clientes finales son conscientes de los diferentes servidores que forman el *cluster*, pueden seleccionar el mejor y conectarse directamente a él. Por tanto no son necesarios elementos intermedios adicionales, como conmutadores L4/L7, que pueden introducir un punto único de fallo y limitan la escalabilidad del *cluster*.

Las simulaciones realizadas sobre varios escenarios para comparar la utilización de recursos de red de XSDF frente a SLPv2, permiten concluir que la escalabilidad de un entorno XSDF gestionado es superior a las SLPv2, ya que la información de servicios pueden actualizarse con más frecuencia, y a la vez consumir menos recursos de red.

También se ha analizado la extensión de *Attribute List*, lo que ha permitido demostrar que esta actualización de SLPv2 mejora significativamente la escalabilidad de SLP en entornos no gestionados. Sin embargo, esta extensión debería ser desplegada con precaución en entornos gestionados, ya que, por limitaciones de la MTU de la LAN, las consultas UDP al DA pueden devolver un número muy reducido de servicios complejos, que tengan muchos atributos.

Por último, para permitir que las aplicaciones de red actuales puedan beneficiarse de XSDF, esta tesis propone una nueva sintaxis de URIs anidadas, que resuelve dos importantes limitaciones de las URIs convencionales:

1. Los servicios sólo pueden identificarse por la dirección IP o el nombre DNS

del servidor que los ofrece.

2. Las URIs permiten incluir un número de puerto, pero no pueden identificar al protocolo de transporte que se desea emplear.

Estas limitaciones a nivel de aplicación podrían complicar enormemente el despliegue de XSDF, o en general de cualquier mecanismo de descubrimiento de servicios o reparto de carga, así como la migración a nuevos protocolos de transporte. Sin embargo, gracias a la nueva sintaxis, que es compatible hacia atrás y mucho más flexible que la actual, es posible seleccionar qué mecanismo de resolución de nombres y protocolo de transporte se desean emplear.

Además, a diferencia de otras soluciones, es posible soportar la nueva sintaxis de URI sin necesidad de modificar las aplicaciones actuales, sino que basta con actualizar las librerías de análisis de URIs y de resolución de nombres (`getaddrinfo()`) para que todas las aplicaciones del sistema puedan beneficiarse inmediatamente de los nuevos protocolos de transporte, descubrimiento de servicios y reparto de carga.

7.2. Principales aportaciones de la Tesis

- **Simple Assistant-Router Architecture (SARA)**. Esta arquitectura distribuida para nodos de red programable permite extender las capacidades de un *router* de altas prestaciones actual, delegando el procesamiento avanzado de paquetes a unos sistemas externos, denominados asistentes, que se conectan al *router* mediante una LAN de alta velocidad. De este modo los servicios de red ejecutan en los asistentes, mientras que el *router* se reduce a encaminar los paquetes. Además, SARA define varios mecanismos de cooperación *router*-asistente que, en su mayoría, no son intrusivos con el *router*, tal como se ha demostrado con el desarrollo de un prototipo basado en un *router* comercial.

- **Descarga dinámica de código fiable**. La arquitectura de nodo de red programable SARA se encuentra entre las investigación de redes activas y la definición de interfaces abiertas, puesto que emplea un Entorno de Ejecución para el despliegue de los servicios de red, pero para su distribución no se basa en código móvil (por sus implicaciones de seguridad), sino que las aplicaciones de red pueden descargarse bajo demanda de servidores de

código, gestionados por el administrador de la red, y por tanto con código autorizado y fiable.

- **Desvío transparente de paquetes de señalización y flujos de datos**. Esta característica de la arquitectura SARA simplifica el despliegue de los servicios de red, puesto que los sistemas finales no tienen que estar preconfigurados con la situación de los nodos intermedios que implementan el servicio de red solicitado, sino que es posible localizarlos dinámicamente. Para ello, este trabajo propone enviar paquetes de señalización (marcados con la opción IP *Router Alert*) directamente al sistema final destino, y será el *router* de los nodos de red SARA el que desvíe dicho paquete a uno de los asistentes para su procesamiento. De igual modo, si el *router* tiene el soporte adecuado, será posible desviar flujos de datos convencionales a los asistentes, para poder realizar procesamiento avanzado sobre los mismos sin necesidad de modificar las aplicaciones de red finales.

- **Router-Assistant Protocol (RAP)**. Además del desvío de paquetes de señalización y la consulta del estado del *router* vía SNMP, SARA define un tercer mecanismo de cooperación *router*-asistente, denominado RAP, que está basado en el protocolo de configuración del IETF ForCES. Gracias a RAP es posible configurar el plano de datos del *router* y suscribirse a eventos importantes del mismo, desde un asistente externo. Además permite encapsular, junto a los paquetes de datos desviados, su metainformación asociada, que en muchos casos puede ser necesaria para que los asistentes puedan procesar dichos paquetes.

- **Cluster de Asistentes**. La tercera característica fundamental de SARA es la posibilidad de utilizar un *cluster* de asistentes para implementar servicios de red de manera escalable y con alta disponibilidad. Para ello, las aplicaciones de red pueden replicarse en varios asistentes y luego distribuir la carga entre todos ellos. Sin embargo, dada la heterogeneidad de los servicios de red, es necesario permitir la utilización simultanea de varias políticas de reparto de carga en el *cluster*, que además pueden tener requisitos de rendimiento y fiabilidad muy diferentes entre sí. Por ello, esta tesis propone dos nuevos mecanismos de reparto de carga, uno de alto rendimiento para el plano de datos, y otro mucho más flexible para el plano de control del *router*.

- **Fast Robust Hashing**. Esta nueva familia de algoritmos de reparto de carga, permite realizar la asignación de flujos a asistentes a velocidad de línea, esto es, por cada paquete desviado. Esta asignación flujo-asistente es

persistente, ya que no se ve afectada por cambios en el número de asistentes (e.g. si un asistente falla), y además requiere un número de operaciones *hash* significativamente menor que las técnicas de *Robust Hashing* actuales. Adicionalmente se ha ideado un método, independiente de la técnica de *Robust Hashing* empleada, que permite realizar una re-asignación de flujos sin pérdida de paquetes en el caso de añadir un nuevo asistente al *cluster*.

- **eXtensible Service Discovery Framework (XSDF)**. En esta tesis se ha propuesto un marco de trabajo extensible, denominado XSDF, que integra el descubrimiento de servicios y el reparto de carga en un único proceso. Aunque el objetivo inicial de XSDF era la localización de asistentes y aplicaciones de red dentro de la arquitectura SARA, su diseño final es lo suficientemente general como para que pueda aplicarse en cualquier entorno. XSDF está basado en SLP y Rserpool, por lo que ofrece todas sus funcionalidades, aunque además añade varias mejoras: un modelo de servicios extensible y modular, la posibilidad de implementar políticas de reparto de carga centralizadas, y la capacidad para localizar servicios en Internet (incluso realizando reparto de carga global).

- **URIs Anidadas**. Analizando los posibles mecanismos de despliegue de XSDF, se observó que la sintaxis actual de las URIs tiene dos limitaciones importantes: no permite especificar ni el protocolo de transporte que se desea emplear, ni utilizar un mecanismo de resolución de nombre distinto al DNS. La nueva sintaxis propuesta en este trabajo, basada en el anidamiento de URIs, solventa estas limitaciones y permite que las aplicaciones puedan beneficiarse de mecanismos de descubrimiento de servicios o de nuevos protocolos de transporte, sin necesidad de ser modificadas.

- **eXtensible Binary Encoding (XBE32)**. Tanto RAP como los protocolos de XSDF se basan en esta nueva codificación de datos binarios alineada a 32 bits, que es mucho más flexible, simple, compacta y fácil de procesar que otras técnicas de codificación como XDR, ASN.1 o XML.

7.3. Trabajos futuros

No cabe duda que la arquitectura SARA permite explorar un gran número de posibilidades en el campo del las redes programables, ya que ofrece la posibilidad de realizar procesamiento escalable dentro de la red, mediante la utilización de nodos de red distribuidos. En particular, sería muy interesante emplear SARA

como plataforma para desarrollar servicios de red sofisticados, como los definidos en el capítulo 4, y así evaluar con un caso real la escalabilidad de la arquitectura SARA.

La segunda línea de investigación abierta consiste en aplicar los principios de SARA para diseñar una arquitectura de nodo de red programable desde cero, sin las limitaciones que impone la utilización de un *router* legado. Este trabajo podría evolucionar en dos líneas de investigación inter-relacionadas. Por un lado sería muy interesante seguir el desarrollo del protocolo ForCES y en general de las arquitecturas distribuidas, para aplicar el concepto de asistente dentro del propio *router*, esto es, ampliar la arquitectura ForCES con FEs de servicios que realizan el procesamiento avanzado y transparente sobre los paquetes.

Por tanto la mayoría de los resultados de esta tesis podría reutilizarse para estas arquitecturas avanzadas de *router*, especialmente la utilización de la técnicas de *Fast Robust Hash*, para el reparto de carga dentro del plano de datos del *router*, o de otros dispositivos como *firewalls*, NATs o balanceadores de carga.

En la misma línea, la tecnología de *Network Processors* (NPs) parece muy prometedora, especialmente para implementar servicios de red de alto rendimiento. Por tanto, los procesadores de red pueden ser una plataforma ideal para los asistentes, o para los FEs de servicios de la arquitectura integrada. Un posible trabajo consiste, por tanto, en adaptar varias de las técnicas propuestas y evaluadas en esta tesis a una plataforma de nodo de red programable, desacoplada y basada en NPs, que emplee las técnicas de *Fast Robust Hashing* para repartir la carga entre los diferentes procesadores de paquetes de la misma.

Por último, una línea de trabajo ortogonal a las redes programables surge a partir de XSDF, o en general de seguir investigando los beneficios que conlleva la integración del descubrimiento de servicios y el reparto de carga en un único proceso. En este sentido sería muy interesante estudiar la aplicación de XSDF a otros escenarios, como por ejemplo realizar el reparto de carga sobre *clusters* de servidores web directamente en los navegadores de los clientes, y así poder comparar XSDF con otros mecanismos, como los balanceadores de carga L4/L7.

Bibliografía

[1] Active Network Backbone (ABone): <http://www.isi.edu/abone/>.

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), August 2002.

[3] Akamai. Turbo-Charging Dynamic Web Sites with Akamai EdgeSuite. White-paper, 2001.

[4] B. Alarcos. *Arquitectura de Seguridad en una red Multiservicio y Multidominio basada en tecnología de redes programables.* PhD thesis, Universidad de Alcalá, September 2004.

[5] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles, and J. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):23–96, May/Jun 1998.

[6] D. Alexander, W. Arbaugh, A. Keromytis, and J. Smith. Security in Active Networks. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems.* Springer-Verlag, Berlin Germany, 1999.

[7] D. Alexander, B. Braden, C. Gunter, A. Jackson, A. Keromytis, G. Minden, and D. Wetherall. Active network encapsulation protocol (ANEP), August 1997.

[8] J. Allen, B. Bass, C. Basso, R. Boivie, and J. Calvignac. IBM PowerNP Network Processor Hardware, Software and Applications. *IBM Journal of Research and Development*, 47(2/3):177–194, March/May 2003.

[9] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 131–145. ACM Press, 2001.

[10] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an Application of Fast Packet Interposing, May 1996.

[11] T. Anderson and J. Buerkle. RFC 3532: Requirements for the Dynamic Partitioning of Switching Elements, May 2003.

[12] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):53–74, February 1995.

[13] D. Andresen, T. Yang, and O. Ibarra. Toward a Scalable Distributed WWW Server on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 42, 1997.

[14] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-Based Web Servers. In *Usenix Annual Technical Conference*, pages 185–198, Monterey, California, June 1999.

[15] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Usenix Annual Technical Conference*, pages 323–336, San Diego, California, June 2000.

[16] A. Audu, R. Gopal, H. Khosravi, and C. Wu. ForwArding and Control ElemenT protocol (FACT) <draft-gopal-forces-fact-06>, November 2003.

[17] A. Audu, J. Salim, and A. Doria. Forwarding and Control Element Separation IP Transport Mapping Layer <draft-audu-forces-iptml-00>, October 2004.

[18] A. Babir, E. Burger, R. Chen, S. McHenry, H. Orman, and R. Penno. RFC 3752: Open Plugable Edge Services (OPES) UseCases and Deployment Scenarios, April 2004.

[19] A. Babir, R. Penno, R. Chen, M. Hofmann, and H. Orman. RFC 3835: An Architecture for Open Pluggable Edge Services (OPES), August 2004.

[20] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the web infrastructure - from caching to replication. *IEEE Internet Computing*, 1(2):18–27, March-April 1997.

[21] Y. Bai and M. Ito. QoS Control for Video and Audio Communication in Conventional and Active Networks: Approaches and Comparison. *IEEE Communication Surveys*, 6(1):42–49, First Quarter 2004.

[22] F. Baker. RFC1812: Requirements for IP Version 4 Routers, June 1995.

[23] S. Baker and B. Moon. Distributed cooperative Web servers. *Computer Networks*, 31(11–16):1215–1229, 1999.

[24] M. Bakke, J. Hufferd, M. Krueger, and T. Sperry. Finding iSCSI Targets and Name Servers Using SLP <draft-ietf-ips-iscsi-slp-09>, August 2004.

[25] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, January 2005.

[26] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. Technical Report 1998-003, Boston University, 1, 1998.

[27] C. Bettstetter and C. Renner. A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In *6th EUNICE Open European Summer School*, Twente, Netherlands, September 2000.

[28] S. Bhattacharjee, K. Calvert, and E. Zegura. An architecture for active networking. In *High Performance Networking (HPN'97)*, pages 265–279, 1997.

[29] J. Biswas, A. Lazar, J. Huard, K. Lim, S. Mahjoub, L. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein. The IEEE P1520 standards initiative for programmable network interfaces. *IEEE Communications Magazine*, 36(10):64–70, October 1998.

[30] J. Biswas, J. Vicente, M. Kounavis, D. Villela, M. Lerner, S. Yoshizawa, and S. Denazis. P1520/TS/IP-013: Proposal for IP L-Interface Architecture, January 2000.

[31] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[32] V. Bollapragada, C. Murphy, and R. White. *Inside Cisco IOS Software Architecture*. Cisco Press, 2000.

[33] T. Brisco. RFC 1794: DNS Support for Load Balancing, April 1995.

[34] M. Calderon, M. Sedano, A. Azcorra, and C. Alonso. Active network support for multicast applications. *IEEE Network*, 12(3):46–52, May/June 1998.

[35] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. RFC 3588: Diameter Base Protocol, September 2003.

[36] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active networks. *IEEE Communications Magazine*, 36(10):72–78, October 1998.

[37] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[38] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The State of the Art in Locally Distributed Web-server Systems. Technical Report RC22209, IBM Research Division, October 2001.

[39] R. Carter and M. Crovella. Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In *IEEE INFOCOM*, pages 1014–1021, 1997.

[40] T. Casavant and J. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[41] H. Chao, C. Lam, and E. Oki. *Broadband Packet Switching Technologies: A Practical Guide to ATM Switches and IP Routers.* Wiley-Interscience, September 2001.

[42] J. Choi, M. Kang, J. Choi, G. Lee, and J. Cha. General Switch Management Protocol (GSMP) v3 for Optical Support <draft-ietf-gsmp-optical-spec-02>, June 2003.

[43] Cisco Systems. Scaling the Internet Web Servers. White Paper <http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/scale_wp.htm>, 1997.

[44] Cisco Systems. Cisco DistributedDirector. White Paper <http://www.cisco.com/en/US/products/hw/contnetw/ps813/products_white_paper09186a0080091e1c.shtml>, 1999.

[45] Cisco Systems. The Cisco Dynamic Feedback Protocol. White Paper <http://www.cisco.com/en/US/tech/tk331/tk896/technologies_white_paper09186a0080091ea9.shtml>, 1999.

[46] Cisco Systems. Failover Configuration for LocalDirector. White Paper <http://www.cisco.com/en/US/products/hw/contnetw/ps1894/products_white_paper09186a0080091ed3.shtml>, 2000.

[47] Cisco Systems. The Evolution of High-End Router Architectures. White Paper <http://www.cisco.com/warp/public/cc/pd/rt/12000/tech/ruar_wp.pdf>, 2001.

[48] Cisco Systems. Using HSRP for Fault-Tolerant IP Routing. White Paper <http://www.cisco.com/univercd/cc/td/doc/cisintwk/ics/cs009.htm>, 2001.

[49] Cisco Systems. Enhanced IP resiliency Using Cisco Stateful Network Address Translation. White Paper <http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/stnat_wp.htm>, 2002.

[50] Cisco Systems. Scalable Content Switching. White Paper <http://www.cisco.com/en/US/products/hw/contnetw/ps792/products_white_paper09186a0080136856.shtml>, 2002.

[51] Cisco Systems. *IOS Security Configuration Guide*, chapter Configuring Unicast Reverse Path Forwarding. Cisco Press, 2003.

[52] Cisco Systems. The Global Server Load Balancing Primer. White Paper <http://www.cisco.com/en/US/netsol/ns340/ns394/ns50/ns254/networking_solutions_white_paper09186a00801b7725.shtml>, 2003.

[53] L. Coene. RFC 3257: Stream Control Transmission Protocol Applicability Statement, April 2002.

[54] L. Coene, P. Conrad, and P. Lei. Reliable Server pool applicability Statement <draft-ietf-rserpool-applic-02>, July 2004.

[55] A. Cohen, S. Rangarajan, and J. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[56] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. In *IEEE INFOCOM*, pages 854–863, 2000.

[57] M. Colajanni, P. Yu, and V. Cardellini. Dynamic Load Balancing in Geographically Distributed Heterogeneous Web Servers. In *International Conference on Distributed Computing Systems*, pages 295–302, 1998.

[58] D. Comer. Network Processors: Programmable Technology for Building Network Systems. *Internet Protocol Journal*, 7(4):2–12, December 2004.

[59] P. Conrad and P. Lei. TCP Mapping for Reliable Server Pooling Enhanced Mode <draft-ietf-rserpool-tcpmapping-02>, June 2004.

[60] Intel Corporation. IXP12XX Product Line of Network Processors: http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[61] A. Crouch and M. Handley. ForCES Applicability Statement <draft-ietf-forces-applicability-02>, June 2003.

[62] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An Architecture for a Secure Service Discovery Service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[63] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *6th International World Wide Web Conference*, April 1997.

[64] DARPA AN Working Group. Architectural framework for active networks, August 1998.

[65] M. Day. Exclusion Extension for Service Location Protocol v2 <draft-day-svrloc-exclusion-03>, January 2003.

[66] D. Decasper, G. Parulkar, and B. Plattner. A Scalable, High Performance Active Network Node. *IEEE Network*, January 1999.

[67] D. Decasper and B. Plattner. DAN - Distributed Code Caching for Active Networks. In *IEEE INFOCOM*, April 1988.

[68] A. DeKok. A Discussion of Metadata in ForCES <draft-dekok-forces-metadata-00>, October 2003.

[69] S. Denazis, K. Miki, J. Vicente, and A. Campbell. Interfaces for Open Programmable Routers. In *International Working Conference on Active Networks (IWAN)*, 1999.

[70] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *IEEE International Computer Conference (COMPCON)*, pages 85–92, 1996.

[71] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999.

[72] A. Doria. GSMPv3 Base Specification <draft-ietf-gsmp-v3-base-spec-06>, November 2004.

[73] A. Doria. ForCES Protocol Specification <draft-ietf-forces-protocol-02>, February 2005.

[74] A. Doria, F. Hellstrand, K. Sundell, and T. Worster. RFC 3292: General Switch Management Protocol (GSMP) V3, June 2002.

[75] A. Doria and K. Sundell. RFC 3294: General Switch Management Protocol (GSMP) Applicability, June 2002.

[76] T. Dreibholz. An Efficient Approach for State Sharing in Server Pools. In *The 27th Annual IEEE Conference on Local Computer Networks (LCN)*, Tampa/Florida, U.S.A., November 2000.

[77] T. Dreibholz and M. Tuxen. High Availability using Reliable Server Pooling. In *Linux Conference Australia (LCA)*, Perth, Australia, January 2003.

[78] S. Dykes, K. Robbins, and C. Jeffery. An Empirical Evaluation of Client-Side Server Selection Algorithms. In *IEEE INFOCOM*, pages 1361–1370, 2000.

[79] S. Eibe. *NAC: Arquitecturade un Nodo Activo en Cluster*. PhD thesis, Universidad Politécnica de Madrid, 2003.

[80] Ericsson-Telebit <http://tbit.dk>.

[81] FAIN D7 - Final Active Node Architecture and Design, May 2003.

[82] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server. <http://www.enseirb.fr/formations/departement/telecommunication/conferences/onlinegames.pdf>, November 2002.

[83] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, June 1999.

[84] P. Fleming and I. McDonald. RFC3712: Lightweight Directory Access Protocol (LDAP): Schema for Printer Services, February 2004.

[85] IETF ForCES Working Group <http://www.ietf.org/html.charters/forces-charter.html>.

[86] Foundry Netwoks. Global Server Load Balancing with ServerIron. White Paper <http://www.foundrynet.com/solutions/appNotes/GSLB.html>.

[87] Foundry Netwoks. Monitoring Server Farm Health Using ServerIron. White Paper <http://www.foundrynet.com/solutions/appNotes/HealthChecks.html>.

[88] Foundry Networks. Maintaining Session Persistence. White Paper <http://www.foundrynet.com/solutions/appNotes/sessionPersistence .html>.

[89] Foundry Networks. URL Switching with ServerIron. White Paper <http://www.foundrynet.com/solutions/appNotes/URLSwitching.html>.

[90] M. Fry and A. Ghosh. Application level active networking. *Computer Networks*, 31(7):655–667, 1999.

[91] A. Galis, B. Plattner, J. Smith, S. Denazis, E. Moeller, H. Guo, C. Klein, J. Serrat, J. Laarhuis, G. Karetsos, and C. Todd. A Flexible IP Active Networks Architecture. In *International Working Conference on Active Networks (IWAN)*, pages 1–15, 2000.

[92] Y. Goland, T. Cai, Y. Gu, and S. Albright. Simple Service discovery Protocol/1.0 Operating without an Arbiter <draft-cai-ssdp-v1-03>, April 2000.

[93] J. Govea and M. Barbeau. Results of Comparing Bandwidth Usage and Latency: Service Location Protocol and Jini. In *Workshop on Ad hoc Communications, held in conjunction with the 7th European Conference on Computer Supported Cooperative Work (ECSCW)*, Bonn, Germany, September 2001.

[94] P. Graham. A DSM Cluster Architecture Supporting Aggressive Computation in Active Networks. In *International Workshop on Distributed Shared Memory*, 2001.

[95] P. Graham and R. Singh. A Mechanism for the Dynamic Construction of Clusters Using Active Networks. In *International Symposium on Cluster Computing and the Grid*, 2001.

[96] A. Gulbrandsen, P. Vixie, and L. Esibov. RFC 2782: A DNS RR for specifying the location of services (DNS SRV), February 2000.

[97] J. Guo, F. Chen, L. Bhuyan, and R. Kumar. A Cluster-Based Active Router Architecture Supporting Video/Audio Stream Transcoding Service. In *International Parallel and Distributed Processing Symposium*, pages 22–26, April 2003.

[98] E. Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, 1999.

[99] E. Guttman. Applicability Statement for Service Location Protocol, Version 2 <draft-guttman-svrloc-as-00>, December 2001.

[100] E. Guttman. RFC 3059: Attribute List Extension for the Service Location Protocol, February 2001.

[101] E. Guttman. The serviceid: URI scheme for Service Location <draft-guttman-svrloc-serviceid-02>, August 2002.

[102] E. Guttman and J. Kempf. Service Location Protocol, Version 2 <draft-guttman-svrloc-rfc2608bis-03>, August 2002.

[103] E. Guttman, C. Perkins, and J. Kempf. RFC 2609: Service Templates and Service: Schemes, June 1999.

[104] E. Guttman, C. Perkins, J. Veizades, and M. Day. RFC 2608: Service Location Protocol, Version 2, June 1999.

[105] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *ACM SIGCOMM*, pages 288–298, 1995.

[106] T. Hain. RFC 2993: Architectural Implications of NAT, November 2000.

[107] S. Hairiri, W. Wang, S. Saha, V. Radhakrishnan, S. Oh, K. Lee, and G. Choi. Survey and Classification of Programmable Network Technologies, 2002.

[108] E. Hensbergen and A. Papathanasiou. KNITS: Switch-based Connection Hand-off. In *IEEE INFOCOM*, 2002.

[109] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[110] R. Hinden. RFC3768: Virtual Router Redundancy Protocol (VRRP), April 2004.

[111] G. Hjálmtýsson. The Pronto Platform - A Flexible Toolkit for Programming Networks using a Commodity Operating System. In *Open Architectures and Network Programming (OPENARCH)*, 2000.

[112] M. Holdrege and P. Srisuresh. RFC 3027: Protocol Complications with the IP Network Address Translator, January 2001.

[113] S. Hollenbeck, M. Rose, and L. Masinter. RFC 3470: Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols, January 2003.

[114] E. Hughes, D. McCormack, M. Barbeau, and F. Bordeleau. An application for discovery, configuration, and installation of SLP services. In *5th Mitel Workshop on Innovation in Technology and Application (MICON)*, Ottawa, Canada, August 2000.

[115] E. Hughes, D. McCormack, M. Barbeau, and F. Bordeleau. Service Recommendation using SLP. In *IEEE International Conference on Telecommunications (ICT)*, Bucharest, June 2001.

[116] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *7th International World Wide Web Conference*, Brisbane, Australia, April 1998.

[117] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997 1997.

[118] Intel IXP2XXX Product Line of Network Processors: <http://www.intel.com/design/network/products/npfamily/ixp2xxx .htm>.

[119] ITU-T. X.680: Abstract Notation One (ASN.1): Specification of basic notation, December 1997.

[120] ITU-T. X.681: Abstract Notation One (ASN.1): Information object specification, December 1997.

[121] ITU-T. X.682: Abstract Notation One (ASN.1): Constraint specification, December 1997.

[122] ITU-T. X.683: Abstract Notation One (ASN.1): Parameterization of ASN.1, December 1997.

[123] ITU-T. X.690: ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), December 1997.

[124] ITU-T. X.691: ASN.1 encoding rules: Specification of Packed Encoding Rules (PER), December 1997.

[125] International Working Conference on Active Networking (IWAN) 2004 <http://www.iwan2004.org/>, 2004.

[126] R. Jaeger, S. Bhattacharjee, J. Hollingsworth, R. Duncan, T. Lavian, and F. Travostino. Integrating Active Networking and Commercial-Grade Routing Platforms. In *USENIX 2000 - Special Workshop on Intelligence at the Network Edge*, March 2000.

[127] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. DEC Research Report TR-301, September 1984.

[128] Jini Architectural Overview. Technical Whitepaper, January 1999.

[129] S. Josefsson. Domain Name System Uniform Resource Identifiers. <draft-josefsson-dns-url-11>, February 2005.

[130] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. In *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, California, November 2001.

[131] Juniper Networks. Implementing a Flexible Hardware-based Router for the New IP Infrastructure. White Paper <http://www.juniper.net /solutions/literature/white_papers/200015.pdf>, 2000.

[132] Juniper Networks. T-series Routing Platforms: System and Packet Forwarding Architecture. White Paper <http://www.telecomweb.com /reports/optimizingip/Tseries_RoutingPlatform_WP.pdf>, 2002.

[133] Juniper Networks. IP Services PICs: DATASHEET <http://www.juniper.net/products/modules/100048.html>, 2004.

[134] E. Jåsund, C. Bettstetter, and C. Schwingenschlögl. A Service Browser for the Service Location Protocol Version 2 (SLPv2). In *7th EUNICE Open European Summer School and the IFIP Workshop on IP and ATM Traffic Management (WATM)*, Paris, France, September 2001.

[135] D. Katz. RFC 2113: IP Router Alert Option, February 1997.

[136] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, 1994.

[137] J. Kempf and J. Goldschmidt. RFC 3082: Notification and Subscription for SLP, March 2001.

[138] J. Kempf and E. Guttman. RFC 2614: An API for Service Location, June 1999.

[139] J. Kempf and G. Montenegro. RFC 3105: Finding an RSIP Server with SLP, October 2001.

[140] L. Kencl and J. LeBoudec. Adaptive Load Sharing for Network Processors. In *IEEE INFOCOM 2002*, volume 2, pages 545–554, June 2002.

[141] H. Kennedy. RFC 3252: Binary Lexical Octet Ad-hoc Transport, April 2002.

[142] S. Kent and R. Atkinson. RFC 2401: Security Architecture for the Internet Protocol, November 1998.

[143] H. Khosravi and T. Anderson. RFC3654: Requirements for Separation of IP Control and Forwarding, November 2003.

[144] H. Khosravi, S. Chawla, F. Ansari, and J. Maloy. TCP/IP based TML (Transport Mapping Layer) for ForCES protocol <draft-khosravi-forces-tcptml-01>, February 2005.

[145] H. Khosravi, G. Kullgren, S. Shew, J. Sadler, and A. Watanabe. RFC 3604: Requirements for Adding Optical Support to the General Switch Management Protocol version 3 (GSMPv3), October 2003.

[146] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, 2nd edition*, volume 2. Addison-Wesley, 1997.

[147] J. Kornblum, D. Raz, and Y. Shavitt. Active Process Interaction with Its Environment. In *Active Networks: Second International Working Conference, IWAN 2000*, October 2000.

[148] D. Larrabeiti, M. Calderón, A. Azcorra, and M. Urueña. A practical approach to Network-based processing. In *IEEE 4th Internacional Workshop on Active Middleware Services*, July 2002.

[149] P. Leach, M. Mealling, and R. Salz. A UUID URN Namespace <draft-mealling-uuid-urn-05>, December 2004.

[150] Q. Li and B. Moon. Distributed Cooperative Apache Web Server. In *10th International World Wide Web Conference*, Hong Kong, 2001.

[151] P. Lin, S. Denazis, J. Vicente, M. Suzuki, J. Redlich, F. Cuervo, J. Biswas, W. Wang, K. Miki, and J. Gutierrez. P1520/TS/IP-001: Programming Interfaces for IP Networks, A White Paper, June 1999.

[152] P. Lin, S. Denazis, J. Vicente, M. Suzuki, J. Redlich, F. Cuervo, J. Biswas, W. Wang, K. Miki, and J. Gutierrez. P1520/TS/IP-003: Programming Interfaces for IP Routers and Switches, an Architectural Framework Document, June 1999.

[153] J. Loughney, M. Stillman, Q. Xie, R. Stewart, and A. Silverton. Comparison of Protocols for Reliable Server Pooling <draft-ietf-rserpool-comp-09>, February 2005.

[154] D. Maltz and P. Bhagwat. TCP Splicing for Application Layer Proxy Performance. Technical Report RC21139, IBM Research Division, March 1998.

[155] M. Mealling and R. Denenberg. RFC 3305: Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations, August 2002.

[156] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an Active Network, September 1999.

[157] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A Node OS for Active Networks. In *INFOCOM (3)*, pages 1127–1136, 2000.

[158] K. Miki, M. Kounavis, and A. Campbell. P1520/TS/IP-010: L-Interface for Routing Control, September 1999.

[159] P. Mockapetris. RFC 1034: Domain names - concepts and facilities, November 1987.

[160] P. Mockapetris. RFC 1035: Domain names - implementation and specification, November 1987.

[161] J. Mogul. Network Behavior of a Busy Web Server and its Clients. Research Report 95/5, Western Research Laboratory, October 1995.

[162] J. Mogul and K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[163] P. Molinero. *Circuit Switching In The Internet*. PhD thesis, Stanford University, June 2003.

[164] D. Mosedale, W. Foss, and R. McCool. Lessons learned administering Netscape's Internet site. *IEEE Internet Computing*, 1(2):28–35, March-April 1997.

[165] A. Mourad and H. Liu. Scalable Web server architectures. In *Second IEEE Symposium on Computers and Communications*, pages 12–16, July 1997.

[166] R. Nait. Pre Association Phase Protocol (PAP) <draft-nait-forces-pap-00>, June 2003.

[167] T. Narten, E. Nordmark, and W. Simpson. RFC2461: Neighbor Discovery for IP Version 6 (IPv6), December 1998.

[168] J. Naugle, K. Kasthurirangan, and G. Ledford. RFC 3049: TN3270E Service Location and Session Balancing, January 2001.

[169] Network Processing Forum (NPF). <http://www.npforum.org>.

[170] Network Processing Forum (NPF). API Software Framework Implementation Agreement. Revision 1.0, August 2002.

[171] Network Processing Forum (NPF). Software API Framework Lexicon Implementation Agreement. Revision 1.0, August 2002.

[172] Network Processing Forum (NPF). MPLS Forwarding Service APIs with Diffserv and TE Extensions Implementation Agreement. Revision 1.0, September 2003.

[173] Network Processing Forum (NPF). Packet Handler API Implementation Agreement. Revision 1.0, September 2003.

[174] Network Processing Forum (NPF). Software API Conventions Implementation Agreement. Revision 2.0, September 2003.

[175] Network Processing Forum (NPF). Interface Management API Implementation Agreement. Revision 3.0, November 2004.

[176] Network Processing Forum (NPF). IPv4 Unicast Forwarding Service API Implementation Agreement. Revision 2.0, June 2004.

[177] Network Processing Forum (NPF). IPv6 Unicast Forwarding Service API Implementation Agreement. Revision 2.0, June 2004.

[178] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. RFC 1987: Ipsilon's General Switch Management Protocol Specification Version 1.1, August 1996.

[179] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching Liaw, T. Lyon, and G. Minshall. RFC 2297: Ipsilon's General Switch Management Protocol Specification Version 2.0, March 1998.

[180] NLARN Trace from an OC-12c PoS link of the University of Florida: <ftp://pma.nlanr.net/traces/daily/20041108/UFL-1099937094-1.tsh.gz>, November 2004.

[181] Nortel Networks. Alteon Application Switch Family. White Paper <http://www.nortelnetworks.com/products/01/alteon/2224/collateral/nn104642-091703.pdf>.

[182] Nortel Networks. Application Switching: Scaling Next Generation networks. White Paper <http://www.nortelnetworks.com/products/01/alteon/2224/collateral/nn105400-090503.pdf>.

[183] OMNeT++ simulator site: <http://www.omnetpp.org>.

[184] L. Ong and J. Yoakum. RFC 3286: An Introduction to the Stream Control Transmission Protocol (SCTP), May 2002.

[185] M. Ott, G. Welling, S. Mathur, D. Reininger, and R. Izmailov. The JOURNEY active network model. *IEEE Journal on Selected Areas in Communications*, 19(3):527–537, March 2001.

[186] Packeteer. Whitepaper: Strategies for managing application traffic <http://packeteer.com/resources//prod-sol/mngapptraf.pdf>, 2005.

[187] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[188] C. Partridge and A. Jackson. RFC 2711: IPv6 Router Alert Option, October 1999.

[189] V. Paxson and S. Floyd. Wide Area Traffic: The Failure of Poisson Modeling. *IEEEACM Transactions on Networking*, 3(3):226–244, June 1995.

[190] C. Perkins. RFC 2003: IP Encapsulation within IP, October 1996.

[191] C. Perkins and E. Guttman. RFC 2610: DHCP Options for Service Location Protocol, June 1999.

[192] D. Peterson. RFC 3822: Finding Fibre Channel over TCP/IP (FCIP) Entities Using Service Location Protocol version 2 (SLPv2), July 2004.

[193] L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 2001.

[194] M. Peyravian, G. Davis, and J. Calvignac. Search Engine Implications for Network Processor Efficiency. *IEEE Network*, 17(4):12–20, July-August 2003.

[195] J. Postel. RFC793: Transmission Control Protocol, September 1981.

[196] K. Psounis. Active Networks: Applications, Security, Safety, and Architectures. *IEEE Communications Surveys*, pages 1–16, First Quarter 1999.

[197] D. Putzolu. ForCES Protocol Evaluation Draft <draft-ietf-forces-evaluation-00>, December 2003.

[198] M. Raguparan, J. Biswas, W. Weiguo, and S. Yoshizawa. P1520/TS/IP-012: L+ Interface for routers that support differentiated services, July 2000.

[199] M. Raguparan and J. Vicente. P1520/TS/IP-014: L- Interface Specification for IP devices, July 2000.

[200] D. Raz and Y. Shavitt. Active Networks for Efficient Distributed Network Management. *IEEE Communications Magazine*, 38(3):138–143, 2000.

[201] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. de Groot, and E. Lear. RFC 1918: Address Allocation for Private Internets, February 1996.

[202] Resonate. Resonate Central Dispatch. In Depth and Technical. White Paper <http://www.resonate.com/solutions/pdf/cd_techwhitepaper.pdf>.

[203] Resonate. TCP Connection Hop. White Paper <http://www.resonate.com/solutions/pdf/cd_tcp_connect_hop_iwp .pdf>.

[204] A. Robertson. Linux-HA Heartbeat System Design. In *4th International Linux Showcase and Conference*, October 2000.

[205] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261: SIP: Session Initiation Protocol, June 2002.

[206] K. Ross. Hash-Routing for Collections of Shared Web Caches. *IEEE Network*, 11(6):37–44, November/December 1997.

[207] A. Rousskov. RFC 4037: Open Pluggable Edge Services (OPES) Callout Protocol (OCP) Core, March 2005.

[208] M. Ruiz, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23, March-April 2001.

[209] J. Salim, R. Haas, and S. Blake. Netlink2 as ForCES Protocol <draft-ietf-jhsra-forces-netlink2-02>, October 2003.

[210] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. RFC3549: Netlink as an IP Services Protocol, July 2003.

[211] J. Saltzer, D. Reed, and D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[212] Salutation Architecture Specification (Part-1) Version 2.1, 1999.

[213] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Workshop on Internet Server Performance*, June 1998.

[214] R. Schemers. lbnamed: A Load Balancing Name Server in Perl. In *USENIX*, 1995.

[215] J. Schoenwaelder. RFC3535: Overview of the 2002 IAB Network Management Workshop, May 2003.

[216] B. Schwartz, W. Zhou, A. Jackson, W. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks, March 1999.

[217] D. Scott, W. Arbaugh, A. Keromytis, and J. Smith. A secure active network environment architecture: Realization in SwitchWare. *IEEE Network*, 12(3):37–45, May-June 1998.

[218] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared Passive Network Performance Discovery. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[219] A. Sevilla, M. Calderón, M. Urueña, and D. Larrabeiti. Control de recursos en un nodo de red programable con entorno de ejecución basado en JAVA. In *JITEL - IV Jornadas de Ingeniería Telemática*, pages 245–252, Las Palmas de Gran Canaria, September 2003.

[220] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *IEEE INFOCOM*, Anchorage, Alaska, April 2001.

[221] H. Shin, S. Lee, and M. Park. Multicast-based Distributed LVS (MD-LVS) for improving scalability and availability. In *8th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 748–754, June 2001.

[222] M-K. Shin, Y-G. Hong, J. Hagino, P. Savola, and E. M. Castro. Appication Aspects of IPv6 Transition. RFC 4038, March 2005.

[223] N. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44, 1992.

[224] H. Sjostrand, J. Buerkle, and B. Srinivasan. RFC 3295: Definitions of Managed Objects for the General Switch Management Protocol (GSMP), June 2002.

[225] T. Spalink, S. Karlin, and L. Peterson. Evaluating Network Processors in IP Forwarding. Technical report, Technical Report TR-626-00, Princeton, January 2001.

[226] R. Srinivasan. RFC 1832: XDR: External Data Representation Standard, August 1995.

[227] P. Srisuresh and K. Egevang. RFC 3022: Traditional IP Network Address Translator (Traditional NAT), January 2001.

[228] P. Srisuresh and M. Holdrege. RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations, August 1999.

[229] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transmission Protocol, October 2000.

[230] R. Stewart, Q. Xie, M. Stillman, and M. Tuexen. Aggregate Server Access Protocol (ASAP) and Endpoint Handlespace Redundancy Protocol (ENRP) Parameters <draft-ietf-rserpool-common-param-08>, February 2005.

[231] R. Stewart, Q. Xie, M. Stillman, and M. Tuexen. Aggregate Server Access Protocol (ASAP) <draft-ietf-rserpool-asap-11>, February 2005.

[232] R. Stewart, Q. Xie, L. Yarroll, J. Wood, K. Poon, and M. Tuexen. Sockets API Extensions for Stream Control Transmission Protocol (SCTP). <draft-ietf-tsvwg-sctpsocket-10>, February 2005.

[233] M. Stillman, R. Gopal, S. Sengodan, E. Guttman, and M. Holdrege. Threats Introduced by Reserpool and Requirements for Security in response to Threats <draft-ietf-rserpool-threats-04>, January 2005.

[234] K. Sundell. GSMPv3 Packet Capable Switch Support <draft-ietf-gsmp-packet-spec-02>, October 2003.

[235] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[236] D. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), 1996.

[237] D. Thaler and C. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, February 1998.

[238] M. Tuexen, Q. Xie, R. Stewart, M. Shore, J. Loughney, and A. Silverton. Architecture for Reliable Server Pooling <draft-ietf-rserpool-arch-09>, February 2005.

[239] M. Tuexen, Q. Xie, R. Stewart, M. Shore, L. Ong, J. Loughney, and M. Stillman. RFC3237: Requirements for Reliable Server Pooling, January 2002.

[240] M. Urueña and D. Larrabeiti. eXtensible Binary Encoding (XBE32) <draft-uruena-xbe32-00>, March 2004.

[241] M. Urueña and D. Larrabeiti. eXtensible Service Location Protocol (XSLP) <draft-uruena-xslp-00>, March 2004.

[242] M. Urueña and D. Larrabeiti. eXtensible Service Registration Protocol (XSRP) <draft-uruena-xsrp-00>, March 2004.

[243] M. Urueña and D. Larrabeiti. eXtensible Service Subscription Protocol (XSSP) <draft-uruena-xssp-00>, March 2004.

[244] M. Urueña and D. Larrabeiti. eXtensible Service Transfer Protocol (XSTP) <draft-uruena-xstp-00>, March 2004.

[245] M. Urueña and D. Larrabeiti. XSDF: Common Elements and Procedures <draft-uruena-xsdf-common-00>, March 2004.

[246] M. Urueña, D. Larrabeiti, M. Calderón, A. Azcorra, J. E. Kristensen, L. K. Kristensen, E. Exposito, D. Garduno, and M. Diaz. An Active Network Approach to Support Multimedia Relays. In *IDMS/PROMS*, pages 353–364, November 2002.

[247] M. Vettorello, C. Bettstetter, and C. Schwingenschlögl. Some Notes on Security in the Service Location Protocol Version 2 (SLPv2). In *Workshop on Ad-hoc Communications, at the 7th European Conference on Computer Supported Cooperative Work (ECSCW)*, Bonn, Germany, September 2001.

[248] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Web++: A System for Fast and Reliable Web Service. In *USENIX Annual Technical Conference*, pages 171–184, 1999.

[249] S. Vrontis, I. Sygkouna, M. Chantzara, and E. Sykas. Enabling distributed QoS management utilizing active network technology. In *IFIP-IEEE International Conference on Network Control and Engineering for Qos, security and mobility*, pages 139–151, October 2003.

[250] W. Wang. Topology Representation for ForCES FE Model <draft-wang-forces-model-topology-00>, August 2003.

[251] W. Wang, Y. Guo, G. Wang, and L. Dong. General Router Management Protocol (GRMP) Version 1 <draft-wang-forces-grmp-02>, May 2004.

[252] G. Welling, M. Ott, and S. Mathur. A Cluster-Based Active Router Architecture. *IEEE Micro*, 21(1):16–25, Jan/Feb 2001.

[253] D. Wetherall. Active network vision and reality: lessions form a capsule-based system. In *ACM Symposium on Operating Systems Principles*, pages 64–79, 1999.

[254] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Open Architectures and Network Programming (OPENARCH)*, April 1998.

[255] D. Wetherall and D. Tennenhouse. The ACTIVE IP Option. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[256] C. Wills and H. Shang. The Contribution of DNS Lookup Costs to Web Object Retrieval. Technical Report WPI-CS-TR-00-12, Worcester Polytechnic Institute, July 2000.

[257] T. Worster, A. Doria, and J. Buerkle. RFC 3293: General Switch Management Protocol (GSMP) Packet Encapsulations for Asynchronous Transfer Mode (ATM), Ethernet and Transmission Control Protocol (TCP), June 2002.

[258] Q. Xie, R. Stewart, M. Stillman, M. Tuexen, and A. Silverton. Endpoint Handlespace Redundancy Protocol (ENRP) <draft-ietf-rserpool-enrp-11>, February 2005.

[259] Q. Xie and L. Yarroll. RSERPOOL Redundancy-model Policy <draft-xie-rserpool-redundancy-model-03>, November 2004.

[260] J. Xu and M. Singhal. Cost-Effective Flow Table Designs for High-Speed Routers: Architecture and Performance Evaluation. *IEEE Transactions on Computers*, 51(9):1089–1099, September 2002.

[261] C. Yang and M. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999.

[262] C. Yang and M. Luo. A Content Placement and Management System for Distributed Web-Server Systems. In *20 Conference on Distributed Computing Systems*, pages 691–698, 2000.

[263] L. Yang, R. Dantu, T. Anderson, and R. Gopal. RFC3746: Forwarding and Control Element Separation (ForCES) Framework, April 2004.

[264] L. Yang, J. Halpern, R. Gopal, A. DeKok, Z. Haraszti, S. Blake, and E. Deleganes. ForCES Forwarding Element Model <draft-ietf-forces-model-03>, July 2004.

[265] L. Yang, J. Halpern, R. Gopal, A. DeKok, Z. Haraszti, S. Blake, and E. Deleganes. ForCES Forwarding Element Model <draft-ietf-forces-model-04>, February 2005.

[266] Y. Yemini and S. daSilva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.

[267] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *USENIX Annual Technical Conference*, 1997.

[268] Zeus Technology. Building High Performance, High-Availability Clusters. White Paper <http://support.zeus.com/doc/tech/cluster.pdf>, 2000.

[269] W. Zhang. Linux virtual server for scalable network services. In *Ottawa Linux Symposium*, 2000.

[270] X. Zhang, M. Barrientos, J. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *3rd USENIX Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.

[271] W. Zhao and H. Schulzrinne. Improving SLP Efficiency and Extendability by Using Global Attributes and Preference Filters. In *International Conference on Computer Communications and Networks (ICCCN)*, Miami, Florida, October 2002.

[272] W. Zhao and H. Schulzrinne. The SLP URL Format <draft-zhao-slp-url-01>, June 2002.

[273] W. Zhao and H. Schulzrinne. Locating IP-to-Public Switched Telephone Network (PSTN) Telephony Gateways via SLP <draft-zhao-iptel-gwloc-slp-06>, February 2004.

[274] W. Zhao, H. Schulzrinne, and E. Guttman. mSLP - Mesh-enhanced Service Location Protocol. In *International Conference on Computer Communications and Networks (ICCCN)*, October 2000.

[275] W. Zhao, H. Schulzrinne, and E. Guttman. RFC 3528: Mesh-enhanced Service Location Protocol (mSLP), April 2003.

[276] W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. RFC 3421: Select and Sort Extensions for the Service Location Protocol (SLP), November 2002.

[277] W. Zhao, H. Schulzrinne, E. Guttman, C. Bisdikian, and W. Jerome. RFC 3832: Remote Service Discovery in the Service Location Protocol (SLP), July 2004.

Acrónimos

A

AA Active Application

AAA Authentication, Authorization, and Accounting

AAL ATM Adaptation Layer

ABONE Active Network Backbone

ANEP Active Network Encapsulation Protocol

ACL Access Control List

AIN Advanced Intelligent Network

ANSI American National Standards Institute

API Application Programming Interface

AR Aplicación de Red

ARP Address Resolution Protocol

ASAP Aggregate Server Access Protocol

ASIC Applicacion-Specific Integrated Circuit

ASN.1 Abstract Syntax Notation One

ATM Asynchronous Transfer Mode

B

BGP Border Gateway Protocol

C

CBR Constant Bit Rate

CCM Connection Control and Management

CE Control Element

CLI Command Line Interface

D

DA Directory Agent

DARPA Defense Advanced Research Project Agency

DCCP Datagram Congestion Control Protocol

DHCP Dynamic Host Configuration Protocol

Diffserv Differentiated Services

DNS Domain Name System

DoS Denial of Service

DRAM Dynamic Random Access Memory

E

ECN Explicit Congestion Notification

EE Execution Environment

ENRP Endpoint haNdlespace Resolution Protocol

F

FE Forwarding Element

ForCES Forwarding and Control Element Separation

FPGA Field Programmable Gate Array

FTP File Transfer Protocol

G

GSMP General Switch Management Protocol

H

HTTP HyperText Transfer Protocol

I

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

IDL Interface Definition Language

IDS Intrusion Detection System

IEEE Instituite of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IOS Cisco Internetwork Operating System

IP Internet Protocol

ISP Internet Service Provider

IWAN Internation Working Conference On Active Networks

J

JNI Java Native Interface

JVM Java Virtual Machine

L

LAN Local Area Network

LDAP Lightweight Directory Access Protocol

LFB Logical Function Block

M

MBONE Multicast Backbone

MIB Management Information Base

MPEG Moving Picture Experts Group

MPLS Multiprotocol Label Switching

mSLP Mesh-enhanced Service Location Protocol

MTU Maximum Transmission Unit

N

NA Network Application

NAS Network Attached Storage

NAT Network Address Translation

NE Network Element

NGSL Network-Generic Services Level

NodeOS Node Operating System

NP Network Processor

NPF Network Processing Forum

NRT-VBR Non-Real-Time Variable Bit Rate

O

ORB Object Request Broker

P

P2P Peer-to-Peer

PDU Protocol Data Unit

PE Pool Element

PHB Per Hop Behavior

PL Protocol Layer

POTS Plain Old Telephone Service

PU Pool Element

Q

QoS Quality of Service

R

RAP Router-Assistant Protocol

RED Random Early Detection

RFC Request For Comments

RIP Routing Information Protocol

RMI Remote Method Invocation

RPC Remote Procedure Call

Rserpool Reliable Server Pooling

RSIP Realm-Specific Internet Protocol

RSVP Resource ReSerVation Protocol

RTP Real-time Transport Protocol

RTT Round Trip Time

S

SA Service Agent

SAN Storage Area Network

SARA Simple Assistant-Router Architecture

SCTP Stream Control Transmission Protocol

SDS Secure Service Discovery Service

SIP Session Initiation Protocol

SLM Salutation Manager

SLP Service Location Potocol

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SOHO Small Office/Home Office

SRAM Static Random Access Memory

SS7 Signalling System 7

SSDP Simple Service Discovery Protocol

T

TCAM Ternary Content Addressable Memory

TCP Transmission Control Protocol

TDM Time-Division Multiplexing

TLS Transport Layer Security

TLV Type-Length-Value

TML Transport Mapping Layer

TTL Time To Live

U

UA User Agent

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

UUID Universally Unique Identifier

V

VASL Value Added Services Level

VNDL Virtual Network Device Level

VPN Virtual Private Network

VRRP Virtual Router Redundancy Protocol

W

WAN Wide Area Network

X

XBE32 eXtensible Binary Encoding

XDR eXternal Data Representation

XML eXtensible Markup Language

XSDF eXtensible Service Discovery Framework

XSLP eXtensible Service Location Protocol

XSRP eXtensible Service Registration Protocol

XSSP eXtensible Service Subscription Protocol

XSTP eXtensible Service Transfer Protocol