Desarrollo de una implementación de referencia del protocolo RELOAD

Trabajo Fin de Grado

Grado en Ingeniería Telemática

MARCOS LÓPEZ SAMANIEGO

TRABAJO FIN DE GRADO

DESARROLLO DE UNA IMPLEMENTACIÓN DE REFERENCIA DEL PROTOCOLO RELOAD

TRABAJO FIN DE GRADO

DESARROLLO DE UNA IMPLEMENTACIÓN DE REFERENCIA DEL PROTOCOLO RELOAD

Autor: MARCOS LÓPEZ SAMANIEGO

Tutor: ROBERTO GONZÁLEZ SÁNCHEZ

Director: ISAÍAS MARTÍNEZ YELMO

Titulación: GRADO EN INGENIERÍA TELEMÁTICA (2011)

Trabajo Fin de Grado
Desarrollo de una implementación de referencia del protocolo RELOAD

La obra completa (incluidas las ilustraciones)
© 2013 Malosa.

Malosa® es una marca registrada en la Oficina Española de Patentes y Marcas por Marcos López Samaniego.

www.malosa.es
marcos@lopezsamaniego.es

Diseño de portada: Angelines Amaro Gómez, Marcos López Samaniego

Ilustraciones: Beatriz Amaro Gómez

Diagramas: Angelines Amaro Gómez, Marcos López Samaniego

Maquetación: Marcos López Samaniego

Edita: Fundación Yoga
NIF G85308682
C/ Agustín Querol, 9
28014 Madrid

www.fundacionyoga.es
info@fundacionyoga.es

Depósito legal: M-19938-2013

Versión electrónica. La edición en formato libro se imprimió en 2013 en Madrid, España.

Printed in Spain

El propietario del copyright autoriza el depósito y puesta a disposición de la obra en el Repositorio Institucional de la Universidad Carlos III de Madrid, e-Archivo, de acceso libre y gratuito a través de internet, y otorgando las condiciones de uso de la licencia Creative Commons *reconocimiento-uso no comercial-sin obra derivada*.

«Quien lo conoce todo, pero no se conoce a sí mismo, es ignorante en todo.»
(Evangelio según Tomás, 67. Manuscritos de Nag Hammadi.)

TRABAJO FIN DE GRADO

INTRODUCCIÓN

Este trabajo consiste en realizar una implementación básica de RELOAD.
Este protocolo está en proceso de normalización y se convertirá en el nuevo estándar de internet RFC 6940.

Cap. 1 — 13

ESTADO DEL ARTE

Hay tres clases de redes superpuestas P2P: estructuradas, desestructuradas y jerárquicas.
RELOAD puede formar una red superpuesta con cualquiera de los tipos, permitiendo múltiples usos.

Cap. 2 — 17

OBJETIVOS

Se pretende introducir al alumno en el mundo empresarial mediante la realización de un software comercial que funcione en escenarios reales.

Cap. 3 — 43

DISEÑO E IMPLEMENTACIÓN

Un diseño fiel al protocolo y una codificación sencilla, buscando la eficiencia computacional, son las bases de una aplicación que funciona con el algoritmo Chord y el protocolo de telefonía SIP.

Cap. 4 — 51

RESULTADOS

Se han realizado pruebas de ámbito local en una pequeña red de doce nodos, y también en internet.
Se muestra la secuencia de mensajes, así como un análisis detallado de algunos tipos de tráfico.

Cap. 5 — 85

CONCLUSIONES

Se han cumplido los objetivos previstos al haber conseguido una implementación sencilla y funcional, extrayendo los principales aspectos del estándar.

Cap. 6 — 97

APÉNDICES

Plan, presupuesto 109

CONTENIDOS

Índice detallado 115

RESUMEN

El grupo de trabajo P2PSIP se creó para desarrollar protocolos y mecanismos que permitieran el uso de SIP en entornos donde el servicio de creación y gestión de sesiones se encuentra distribuido [P2PSIP], minimizando así la necesidad de servidores centralizados. Bajo esta premisa nació el protocolo RELOAD, cuyo diseño se ha generalizado para permitir otras aplicaciones con similares exigencias, y que actualmente está en proceso de normalización por el IETF.

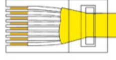
El objetivo de este Trabajo Fin de Grado es realizar una implementación de referencia de dicho protocolo, debido al gran interés mostrado en los últimos años por la comunidad científica y empresarial en los asuntos relacionados con las redes *peer-to-peer*.

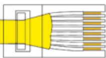
En este proyecto se ha implementado una parte relevante de RELOAD. Para ello, en primer lugar, se realizó un diseño orientado a objetos basado en un subconjunto funcional del borrador del IETF y, posteriormente, se llevó a cabo la codificación, poniendo énfasis en la sencillez del código, de manera que la aplicación pueda modificarse y reutilizarse con facilidad.

Finalmente, se han realizado y documentado pruebas, cuyo propósito ha sido demostrar el correcto funcionamiento en escenarios reales utilizando una implementación previamente existente de SIP.

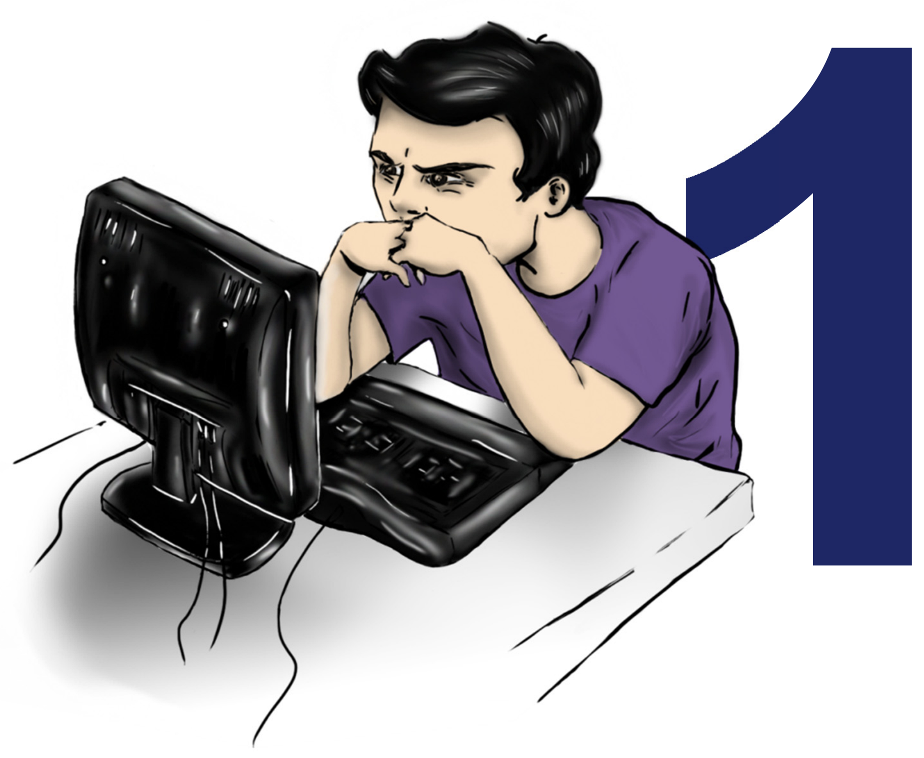
El borrador se encuentra actualmente en la cola de edición y se convertirá en el nuevo estándar de internet RFC 6940.

Palabras clave: RELOAD, *peer-to-peer*, Chord.





Capítulo



1

Introducción

1.1 INTRODUCCIÓN AL TRABAJO FIN DE GRADO

RELOAD es un protocolo de señalización en internet para redes *peer-to-peer*, que está siendo desarrollado por el Internet Engineering Task Force (IETF) y cuyo objetivo es soportar aplicaciones que puedan funcionar en entornos distribuidos. A pesar de estar ligado a SIP desde su nacimiento, RELOAD no se diseñó únicamente como protocolo de voz sobre IP, su campo de aplicación se ha extendido para que pueda ser utilizado por cualquier otro protocolo con requisitos similares. Para ello se pueden crear diferentes usos (*usages*) para funcionar sobre RELOAD. En el apartado 2.2 se enumeran algunos de ellos.

Este protocolo actúa formando una red superpuesta (*overlay network*). Su diseño modular permite utilizarlo con cualquier tipo de red P2P que previamente se defina en el estándar. El algoritmo Chord es parte del diseño base de RELOAD, por lo que este tipo de red debe ser considerada.

El presente Trabajo Fin de Grado consiste en realizar una implementación de referencia de RELOAD. Debido a lo extensa que es la especificación, se ha decido dividir el proyecto en varias partes. Aquí se presenta la primera de ellas.

El idioma de programación elegido es Java. Hasta ahora no se tiene constancia de ninguna implementación del protocolo en Java, tan solo en C y C++. Aparte de este motivo, se ha escogido por su orientación a objetos, por su carácter multiplataforma y por ser una de las lenguas instruidas en la Ingeniería en Telemática. En el trabajo se ha tenido que profundizar y ampliar las técnicas de codificación y los conocimientos previamente adquiridos en esta área, y aplicarlos a un proyecto de mayor extensión y dificultad a todo lo realizado anteriormente.

De forma adicional a esta memoria, se ha redactado un artículo de investigación para mostrar el trabajo a la comunidad. Dicho documento ha sido presentado a las XI Jornadas de Ingeniería Telemática (JITEL 2013).

1.2 CONTENIDOS DE LA MEMORIA

- En el capítulo 1 se realiza una breve introducción al trabajo, además de presentar los contenidos en los que se estructura la memoria.

- El capítulo 2 ofrece información acerca de las redes *peer-to-peer* superpuestas e identifica qué relación tienen estas con RELOAD.

- El capítulo 3 da una visión acerca de qué se pretende conseguir con el proyecto, analiza los aspectos implementados del borrador y hace un análisis del trabajo futuro, ofreciendo una salida para que el trabajo pueda ser continuado de forma inmediata.

- En el capítulo 4 se da información detallada acerca de cómo se diseñaron los diferentes módulos y realiza una ligera descripción de las clases de Java más importantes.

- El capítulo 5 brinda detalles sobre el funcionamiento de la aplicación creada, poniendo énfasis en los mensajes que se intercambian entre los nodos, y se presenta el resultado de una red que funciona con SIP de manera distribuida.

- El capítulo 6 da un repaso a los hitos conseguidos, diferenciando entre conclusiones personales y generales. Se analiza el coste y la extensión de la aplicación y se detallan los recursos que consume. Por último, se realizan críticas a RELOAD, a la par que se proponen algunas mejoras.

El borrador de internet que se ha tratado de seguir es: *REsource LOcation And Discovery (RELOAD) Base Protocol draft-ietf-p2psip-base-26* [RELO13], con fecha de caducidad: 24 de agosto de 2013. Las referencias a este documento [RELO13] son recurrentes en toda la memoria, por lo que no se citará de forma explícita a partir de este punto.

Capítulo

2

Estado del arte

2.1 REDES P2P SUPERPUESTAS

Una red superpuesta *peer-to-peer* (*P2P overlay network*) es una colección distribuida de computadores, llamados *peers*, que forman una serie de conexiones entre sí. Estos *peers* tienen roles simétricos: todos ellos actúan de cliente y servidor simultáneamente, encaminando mensajes y compartiendo recursos [BYK08].

Estas redes tienen los siguientes principios: autoorganización, simetría de roles, compartición de recursos, escalabilidad, autonomía en los *peers* y flexibilidad [BYK08].

Existen tres tipos principales de redes superpuestas P2P: estructuradas, desestructuradas y jerárquicas.

- En las estructuradas, los *peers* y en ocasiones los recursos, se organizan siguiendo criterios específicos y algoritmos. Las más de las veces utilizan tablas de *hash* distribuidas (DHT, por sus siglas en inglés) para indexar los contenidos. Sin embargo, no podemos definir DHT como sinónimo de red superpuesta estructurada [VVN12]. Este tipo de redes se caracterizan por su capacidad para proporcionar escalabilidad, tolerancia a fallos y autogestión, haciéndolas adecuadas para aplicaciones distribuidas a escala de internet [SGH09]. Algunos ejemplos de algoritmos DHT son: CAN (Content Addressable Network), Chord, Kademlia, Pastry y P-Grid.

- Las redes superpuestas desestructuradas no proporcionan ningún algoritmo para organizar o para optimizar las conexiones. Pueden ser descentralizadas, como Gnutella y Freenet, donde los *peers* son equipotentes y no se consideran nodos de mayor importancia; híbridas, como Kazaa, que permiten que haya nodos infraestructura con funciones especiales; o centralizadas, como Napster, que utiliza

un servidor central para funciones de indexación y para arrancar el sistema. Pese al parecido de este último con las redes estructuradas, no existe un algoritmo que determine las conexiones entre los *peers* [VVN12].

- Una red superpuesta jerárquica es aquella en la que varias redes de superposición de diferentes tipos están conectadas entre sí a través de otra red de superposición. Generalmente, esta interconexión se produce gracias a unos nodos llamados *super-peers*, que forman parte simultáneamente de dos redes superpuestas. Un ejemplo es H-P2PSIP [MBCGG09].

En los dos siguientes apartados se exponen un par de redes estructuradas y algoritmos DHT: Chord y Kademlia.

2.1.1. Chord

Chord es un protocolo y algoritmo de tabla de *hash* distribuida *peer-to-peer* creado en 2001 por Ion Stoica, Robert Morris, David Karger, Frans Kaashoek y Hari Balakrishnan, del Instituto de Tecnología de Massachusetts, EUA [Chord].

En este protocolo, los nodos se distribuyen en un anillo lógico de hasta 2^n nodos o *peers*, quedando generalmente huecos en la numeración, la cual se va incrementando en sentido de las agujas del reloj. Cada nodo debe tener una conexión al menos con el nodo anterior a él (el predecesor) y con el posterior (el sucesor). En Chord-RELOAD, se especifica que n=128.

Si bien, simplemente conociendo un sucesor y un predecesor sería posible que todo nodo fuese alcanzable por cualquier otro *peer*, para conseguir métodos de búsqueda más rápidos, cada nodo debe conectarse a otros que estén a 180º de su posición, a 90º, a 45º…, y así hasta que se esté vinculado a n *peers*. Estos son los llamados *fingers*.

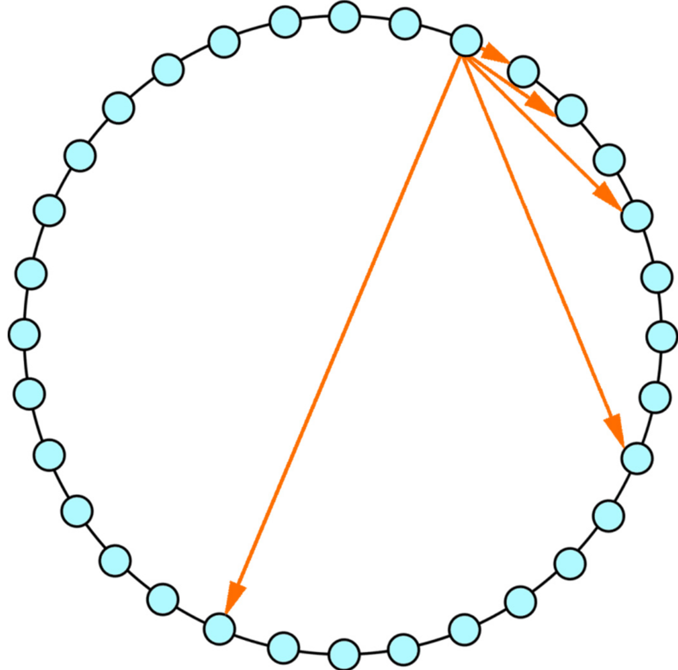
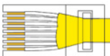


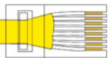
FIGURA 1: Red Chord con n=5, con uno de los nodos conectado a sus 5 *fingers*

Chord mapea claves en nodos; dada una clave cualquiera, un nodo es capaz de identificar qué nodo es responsable de dicha clave. Esta asignación se realiza mediante una función de *hash* consistente. Las claves y los nodos reciben un identificador de n bits, estando ambos en el mismo espacio de numeración [wikiChord].

La versión original de este protocolo asigna el identificador de nodo a partir de la dirección IP del nodo. Esto garantiza que los nodos se distribuyen uniformemente en el anillo. El identificador de recurso se genera a partir de una cadena de caracteres o de un *array* de bytes [wikiChord].

Para evitar la pérdida de información ante la caída de un nodo, cada *peer* replica los datos de los que es responsable en un número determinado de nodos posteriores a él.





2.1.2. Kademlia

Kademlia es un algoritmo DHT diseñado por Petar Maymounkov y David Mazières, de la Universidad de Nueva York, en el año 2002 [Kadem].

En este algoritmo, los nodos se identifican por un *node ID*, y dicho valor sirve tanto de identificador como para localizar los recursos de la red. Kademlia utiliza una búsqueda basada en distancias, la cual se calcula utilizando disyunción exclusiva (XOR) entre dos *node ID*. La distancia obtenida en la operación es un número entero que cumple las siguientes propiedades: la distancia entre un nodo y sí mismo es cero, la distancia es simétrica (de A a B es igual que de B a A) y cumple además la desigualdad triangular [wikiKadem].

Los identificadores de nodo y de recurso son de 160 bits. Para guardar información en la red, se generan unos pares <clave,valor> que se almacenan en los nodos más cercanos a las claves [Kadem]. En una búsqueda, cada iteración permite estar un bit más cerca del destino. En el peor de los casos, una búsqueda para encontrar un nodo o recurso llevará 160 pasos.

Este tipo de red tiene cuatro mensajes diferentes: PING, STORE, FIND_NODE y FIND_VALUE. Sin embargo, para que pueda utilizarse con RELOAD, estos mensajes deberán adaptarse a los exigidos por el protocolo.

En comparación con otros sistemas como Chord, Kademlia tiene, entre otras, la ventaja de ser capaz de obtener información de encaminamiento útil a partir de las consultas que se reciben. Además, debido a la asimetría de las métricas de Chord, sus tablas de encaminamiento son rígidas, porque cada entrada en la tabla de *fingers* debe almacenar el nodo exacto que precede a un intervalo en el espacio de identificación. Algún nodo del intervalo podría estar demasiado lejos de los nodos que lo preceden en ese mismo intervalo. Kademlia, en contraste, puede enviar una consulta a cualquier nodo dentro de un intervalo, lo que le permite seleccionar las rutas basándose en la latencia [Kadem].

2.2 INTRODUCCIÓN A RELOAD

Desde hace más de diez años, venía existiendo la necesidad de convertir Session Initiation Protocol (SIP) –el protocolo del IETF para la señalización de telefonía en internet– en un protocolo distribuido *peer-to-peer*. Uno de los mayores problemas en las comunicaciones multimedia por internet es conocer el *host* a través del cual podemos llegar a un determinado usuario. Para ello, se reemplazan los procedimientos del SIP cliente-servidor, basados en *proxys* y agentes de usuario, por una red de superposición *peer-to-peer*. Con estos objetivos se puso en marcha el grupo de trabajo Peer-to-Peer Session Initiation Protocol (SIPP2P) [P2PSIP].

Peer-to-Peer Protocol (P2PP) fue una de las primeras propuestas que se hicieron en el grupo de trabajo, pero finalmente se descartó en favor de RELOAD. Permitía formar una red de superposición con multitud de protocolos P2P estructurados y desestructurados definidos en el borrador y, al igual que RELOAD, permitía usarse con más aplicaciones, además de SIP [P2PP].

Arquitectónicamente, P2PP y RELOAD son muy similares. Algunas de las características de P2PP, tales como el soporte de redes P2P desestructuradas, la fiabilidad salto a salto, algunos indicadores que permiten mantener la red, los diagnósticos, etc., fueron incorporadas en la versión de RELOAD *draft-bryan-p2psip-reload-03*. De esta manera, se aunaron las características de ambos protocolos en RELOAD.

Esta es la sinopsis de RELOAD, de acuerdo al borrador del IETF:

«RELOAD [acrónimo de REsource LOcation And Discovery] es un protocolo de señalización entre iguales para usarse en internet. Todo protocolo de señalización *peer-to-peer* ofrece a sus clientes un almacenamiento abstracto y un servicio de mensajería entre un grupo de *peers* que cooperan y forman la red superpuesta. RELOAD está diseñado para admitir una red Peer-to-Peer

Session Initiation Protocol (P2PSIP), pero también puede ser utilizado por otras aplicaciones que tengan requisitos similares, definiendo nuevos usos que especifiquen los tipos de datos que necesitan almacenarse para una determinada aplicación. RELOAD define un modelo de seguridad basado en un servicio de inscripción de certificados que proporciona identidades únicas. NAT transversal es un servicio fundamental del protocolo. RELOAD también permite el acceso de los nodos “clientes” que no necesitan encaminar tráfico o almacenar datos para otros.»

La creación de un protocolo normalizado como RELOAD puede considerarse un hito en la historia de internet. Su planteamiento supone un cambio del modelo cliente-servidor actual a un nuevo modelo distribuido entre iguales. Es previsible la tendencia empresarial a asumir progresivamente este cambio de modelo, debido al alto coste que suponen los servidores centralizados.

Las redes superpuestas P2P se han asociado hasta el momento presente con redes de intercambio de archivos. Resulta interesante cómo RELOAD no solo ha tomado esta misma filosofía para un fin mucho más generalista, sino que incluye en su diseño a las redes *peer-to-peer* ya existentes, lo que permite centrar el esfuerzo en nuevos aspectos como la seguridad y los diferentes tipos de mensaje, en lugar de tener que redefinir nuevos algoritmos.

Este protocolo es especialmente significativo por su extensibilidad, ya que, al no estar pensando para un protocolo concreto, permite múltiples usos. Es enorme la cantidad de usos que se están definiendo actualmente para funcionar sobre RELOAD, aparte de SIP: Distributed Conference Control (DisCo) [DisCo13], Shared Resources (ShaRe) [ShaRe13], Constrained Application Protocol (CoAP) [CoAP13], Simple Network Management Protocol (SNMP) [SNMP13], Service Discovery [MACA13], Public Switched Telephone Network (PSTN) Verification [PRJ12]…, teniendo en cuenta que RELOAD aún está en proceso de normalización.

2.3 ARQUITECTURA DE RELOAD

RELOAD se define como un protocolo de la capa de aplicación, el nivel superior de la torre de protocolos TCP/IP. Como nivel de transporte permite utilizar los protocolos “seguros” TLS (orientado a conexión) o DTLS (no orientado a conexión).

Arquitectura TCP/IP

Aplicación	RELOAD			
Transporte	TLS		DTLS	
	TCP		UDP	
Interred	IP			
Enlace	Ethernet	Frame Relay	PPP	…
Físico	…			

FIGURA 2: Situación de RELOAD en la torre de protocolos de internet

RELOAD es fundamentalmente una red superpuesta, como puede observarse en la figura 3, en la que se redefinen los niveles de red, transporte y aplicación, y también un nuevo nivel de encaminamiento.

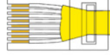
El nivel de enlace de esta red será TLS o DTLS, por lo que dos nodos directamente conectados, podrían estar separados miles de kilómetros y tener decenas de saltos IP entre ellos.

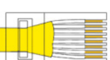
Forwarding & Link Management es el nivel de interred, por lo que esta capa proveerá conectividad extremo a extremo.

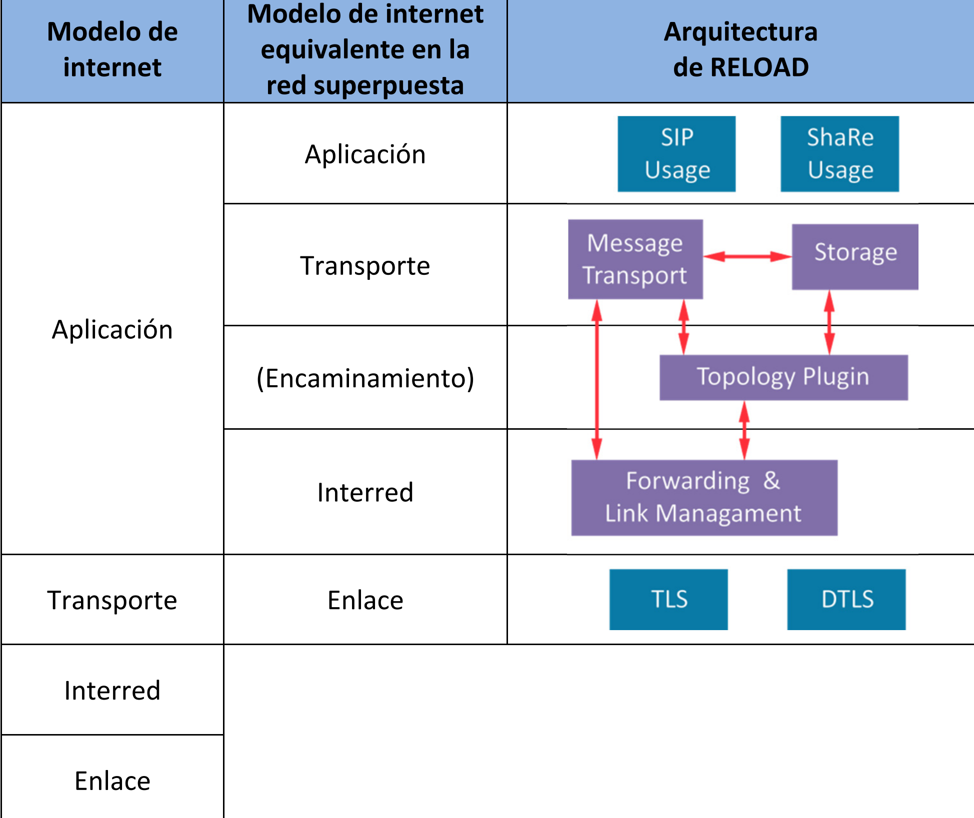
Otra parte del nivel de interred está configurado por la red *peer-to-peer* implementada, en este caso Chord. Más en concreto, las decisiones de encaminamiento son tomadas por este módulo.

En el nivel de transporte tenemos los módulos Message Transport y Storage.

Por último, el nivel de aplicación de esta red se corresponde con el uso.







Modelo de internet	Modelo de internet equivalente en la red superpuesta	Arquitectura de RELOAD
Aplicación	Aplicación	SIP Usage, ShaRe Usage
	Transporte	Message Transport, Storage
	(Encaminamiento)	Topology Plugin
	Interred	Forwarding & Link Managament
Transporte	Enlace	TLS, DTLS
Interred		
Enlace		

FIGURA 3: Arquitectura de RELOAD comparada con otros modelos

2.3.1. Direccionamiento

El identificador de los Nodos de RELOAD es el Node-ID, formado por un número variable de bits: entre 128 y 160 –128 bits en Chord–. En el mismo espacio de numeración se encuentra el Resource-ID, que identifica a los recursos almacenados en la red.

Cada nodo será responsable de aquellos Resource-ID que sean menores o iguales que su propio Node-ID y que a su vez sean mayores que el Node-ID inmediatamente anterior.

Por ejemplo, en una red que tenga tres nodos: 85, 107 y 240, el Node-ID 107 será responsable de los Resource-ID del 86 al 107. La idea aquí es que, si bien generalmente existirán huecos en la numeración y, por tanto, raramente estarán ocupados la totalidad de los Node-ID, sí podemos garantizar que, independientemente del tamaño de la red, siempre habrá un nodo que responda ante un recurso determinado.

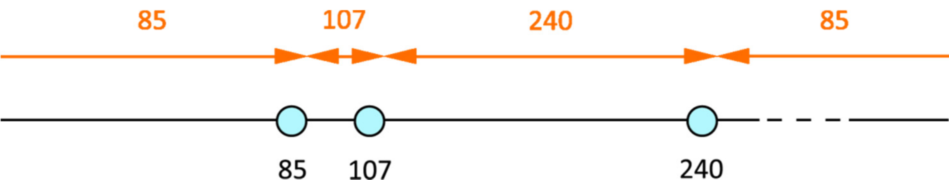


FIGURA 4: Situación de 3 nodos y sección de la que cada uno es responsable

En el ejemplo anterior, si enviamos un mensaje Ping al Node-ID 200, no recibiremos respuesta, pero si lo hacemos al Resource-ID 200, el nodo 240 estará encantado de respondernos. Naturalmente, el nodo 85 será responsable desde el Resource-ID 241 hasta el 85 pasando por el cero, por lo que la numeración tiene carácter circular. Entre los tres nodos abarcan la totalidad de la red, por lo que cualquier recurso tendrá como responsable a un Node-ID.

2.3.2. Capa de uso

RELOAD no puede funcionar por sí mismo, sino que debe servir de base para algún tipo de aplicación. Los usos que se le pueden dar a RELOAD se denominan, precisamente, usos (*usages*).

El uso debe definir cómo una aplicación puede almacenar su información en la red, para lo cual se pueden definir nuevos tipos de datos y las reglas que delimiten cómo deben usarse esos datos. Una sola aplicación puede necesitar múltiples usos.

2.3.3. Transporte de mensajes / Message Transport

Esta capa es la encargada de transportar los mensajes de extremo a extremo. Es llamada por la capa de uso y por el componente de almacenamiento, y debe ser capaz de enviar los mensajes al nodo de destino, tanto si este es un Node-ID como si es un Resource-ID.

Igualmente en sentido contrario, cuando Message Transport recibe un nuevo mensaje, lo entrega al módulo pertinente, en función del tipo de mensaje (ver apartado 2.4: Mensajes en RELOAD).

2.3.4. Almacenamiento / Storage

Una de las características de RELOAD es que, además de ser una red de mensajes, es una red de almacenamiento. Los nodos de la red guardan en memoria aquellos datos que la aplicación considera necesarios. El componente de almacenamiento es el encargado de guardar los datos y de devolver estos datos cuando se le soliciten.

Un Node-ID almacenará necesariamente aquellos recursos de los que sea responsable, pero también le serán enviadas peticiones para almacenar Resources-ID cuyo encargado sea otro nodo. De esta manera, se podrán almacenar réplicas por toda la red y existirá una determinada redundancia ante una caída de nodos.

El tipo de datos que un nodo puede almacenar se denomina Kind, y se identifica por su Kind-ID, que es un número entero de 32 bits asignado por la IANA –o bien, perteneciente a un rango privado–. Un Resource-ID puede contener varios Kind-ID.

El modelo de los datos que puede almacenar un Kind-ID se denomina Data Model, en un principio se consideran tres, aunque usos futuros podrían definir nuevos modelos. Los posibles tipos a almacenar son: valor individual; *array*, múltiples valores indexados por un número; o diccionario, varios valores indexados por una clave o palabra.

2.3.5. Complemento de topología / Topology Plugin

El complemento de topología define una estructura genérica sobre la cual pueden funcionar diversos algoritmos de superposición *peer-to-peer* estructurados y desestructurados –los estructurados funcionan, generalmente con tablas de *hash* distribuidas–. Las funciones de estos complementos (*plugins*) son básicamente definir la topología de la red y cómo los mensajes son encaminados a través de los nodos.

Sin embargo, no es posible utilizar cualquier tipo de algoritmo preexistente. Progresivamente, se deberán ir adaptando las diferentes redes *peer-to-peer* para que puedan funcionar con RELOAD. En el momento de escribir esta memoria, el único complemento definido es Chord, un algoritmo DHT, que además es obligatorio implementar. Chord se modifica ligeramente respecto a su diseño original para que funcione con RELOAD (ver detalles en el apartado 2.5.3). Es importante especificar que debe ser posible sustituir Chord por otro algoritmo y que el resto de niveles sigan funcionando sin realizar ninguna modificación en ellos.

El complemento de topología es el encargado de mantener las tablas de rutas y de conexiones. Cómo son estas tablas depende del algoritmo en cuestión. Se debe consultar a Topology Plugin para tomar las decisiones acerca de encaminar los paquetes.

2.3.6. Capa de reenvío y manejo de enlaces / Forwarding & Link Management

Esta capa se comunica con el complemento de topología para obtener las tablas de conexiones y rutas, y así poder entregar el mensaje al siguiente nodo. Es la encargada de realizar las conexiones con nuevos nodos y de atravesar NAT (Network Address Translation) y cortafuegos mediante ICE (Interactive Connectivity Establishment).

Forwarding & Link Management tiene acceso al nivel de enlace de la red superpuesta y se ocupa de mantener las conexiones, por lo que será el módulo responsable de recibir los mensajes de la red y de enviar nuevos paquetes a petición de las capas superiores.

2.3.7. Capa de enlace / Link

Link aporta una cabecera extra llamada Framing Header (**FH**), que solo tiene sentido en el contexto del enlace y se elimina a cada salto.

Cuando se usa un protocolo de transporte fiable como TCP, se utiliza para enmarcar los mensajes y proporcionar temporización.

Debido a la naturaleza no fiable de UDP, cuando se utiliza DTLS en el nivel de enlace de la red superpuesta, es requerido el uso del protocolo Simple Reliability. Este protocolo hace uso de la cabecera Framing Header para aportar control de congestión y semifiabilidad.

2.4 MENSAJES EN RELOAD

A diferencia de como ocurre habitualmente en TCP/IP, cada nivel de la nueva torre de protocolos no define necesariamente una cabecera propia. En lugar de esto, existe un mensaje común a todos los niveles de la red superpuesta, que consta de tres partes: Forwarding Header, Message Contents y Security Block. Esta estructura se sitúa a continuación de la cabecera Framing Header, como puede verse en los siguientes paquetes IP de RELOAD:

IP	TCP	TLS	FH	Forwarding Header	Message Contents	Security Block

IP	UDP	DTLS	FH	Forwarding Header	Message Contents	Security Block

FIGURA 5: Mensaje de Reload en un paquete IP

Se permiten tres protocolos en el nivel de enlace de la red superpuesta:

- **DTLS sobre UDP con Simple Reliability.** Se utiliza el protocolo Simple Reliability, las comprobaciones de STUN y se envían *keepalives*.
- **TLS sobre TCP con Framing Header, sin ICE.** No se utilizan las comprobaciones de conectividad de STUN ni se envían *keepalives*.
- **DTLS sobre UDP con Simple Reliability, sin ICE.** Se utiliza Simple Reliability, pero no las comprobaciones de STUN ni los *keepalives*.

El tamaño del mensaje es de longitud variable. Existen algunos campos de longitud fija y otros que no lo son. En aquellos no fijos, acompaña un campo de longitud que se refiere exclusivamente al tamaño en bytes del siguiente segmento del mensaje. No existe, por tanto, un campo de longitud total del paquete.

A continuación se expone un mensaje típico de RELOAD. Solo se detallan los campos principales:

Forwarding Header

<table>
<tr><td>0</td><td>8</td><td>16</td><td>24</td><td>31</td></tr>
<tr><td colspan="5">RELO TOKEN</td></tr>
<tr><td colspan="5">OVERLAY</td></tr>
<tr><td colspan="2">CONFIGURATION SEQUENCE</td><td>VERSION</td><td colspan="2">TTL</td></tr>
<tr><td colspan="5">FRAGMENT</td></tr>
<tr><td colspan="5">LENGTH</td></tr>
<tr><td colspan="5">TRANSACTION ID</td></tr>
<tr><td colspan="5">MAX RESPONSE LENGTH</td></tr>
<tr><td colspan="2">VIA LIST LENGTH</td><td colspan="3">DESTINATION LIST LENGTH</td></tr>
<tr><td colspan="2">OPTIONS LENGTH</td><td colspan="3">VIA LIST</td></tr>
<tr><td colspan="5">VIA LIST ARRAY[]
[…]</td></tr>
<tr><td colspan="5">DESTINATION LIST ARRAY[]
[…]</td></tr>
<tr><td colspan="5">OPTIONS ARRAY[]
[…]</td></tr>
</table>

FIGURA 6: Estructura de la cabecera de reenvío

- **RELO Token:** La cadena “RELO” en ASCII con el primer bit de “R” a 1.
- **Overlay:** *Hash* de 32 bits del nombre de la red superpuesta según aparece en el fichero de configuración (por ej.: “CHORD-RELOAD”).
- **Configuration Sequence:** El número de secuencia del fichero de configuración.
- **Version:** Versión del protocolo RELOAD entre 0.1 y 25.4 (se codifica como un entero entre 1 y 254, no estando permitido 0 ni 255).
- **TTL:** Número de nodos por los que puede pasar un mensaje antes de ser descartado, cuyo valor inicial es 100 y se decrementa a cada salto.
- **Fragment:** Este campo se utiliza para manejar la fragmentación.
- **Transaction ID:** Núm. aleatorio único para identificar retransmisiones.
- **Maximum Response Length:** Tamaño máximo de la respuesta en bytes.
- **Via List:** Explicado en 2.5.1.
- **Destination List:** Explicado en 2.5.1.
- **Options:** Campos de extensión de la cabecera mediante opciones.

Message Contents

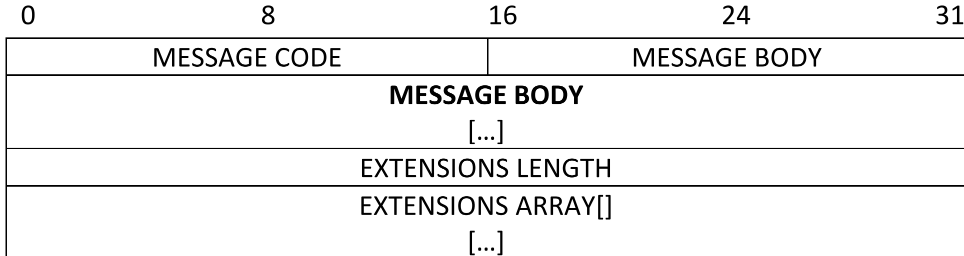


FIGURA 7: Estructura de los contenidos del mensaje

- **Message Code:** Se detalla a continuación del Security Block.
- **Message Body:** Se detalla a continuación del Security Block.
- **Extensions:** Extensiones futuras al mensaje (aún no definidas).

Security Block

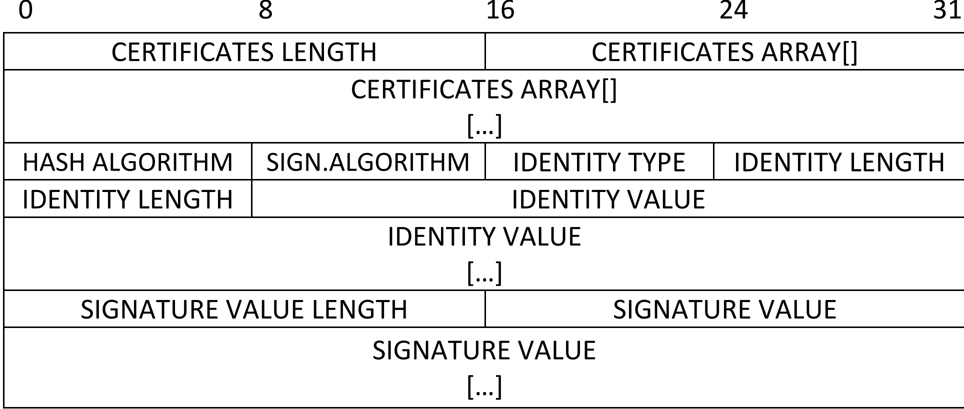


FIGURA 8: Estructura del bloque de seguridad

- **Certificates:** Una serie de certificados necesarios para verificar las firmas. Pueden ser de varios tipos (por ejemplo: X.509).
- **Hash Algorithm & Signature Algorithm:** Algoritmos en uso en la firma.
- **Signer Identity Type:** Indican el formato de Signer Identity Value.
- **Signer Identity Value:** *Hash* del certificado.
- **Signature Value:** Valor de la firma.

MESSAGE BODY es el campo central, pues es donde lleva el contenido del mensaje, por lo que una estructura de RELOAD también puede verse de la siguiente manera:

MESSAGE HEADER	MESSAGE BODY	MESSAGE FOOTER

FIGURA 9: Visión alternativa de los mensajes de RELOAD

Lógicamente, MESSAGE HEADER corresponde con todos los campos previos a MESSAGE BODY y MESSAGE FOOTER lo identificamos con los posteriores.

MESSAGE BODY tiene un formato diferente en función del tipo de mensaje RELOAD que se esté enviando, pero es un campo de contenido opaco para la capa que gestiona la estructura del mensaje.

Message Transport eliminará la cabecera y el pie del mensaje y entregará MESSAGE BODY al módulo que corresponda. La codificación del tipo de mensaje se realiza con un número entero en el campo MESSAGE CODE, el cual identifica de forma inequívoca al nivel responsable.

Entre los diferentes mensajes en Reload, destacan:

- **Attach:** crea una conexión (a nivel TLS/DTLS) con otro nodo.
- **Join:** el nodo que lo envía está listo para formar parte de la red.
- **Update:** transfiere la información de encaminamiento e indica a otros nodos que el *peer* que lo manda está listo para encaminar para ellos.
- **Store y Fetch:** se usan para almacenar y obtener información en la red, respectivamente.
- **Leave:** el nodo que lo envía abandona la red.

La siguiente tabla enumera todos los tipos de mensajes y sus códigos:

Tipo de Mensaje	Código de mensaje	Módulo encargado
Probe Request	1	Topology Plugin
Probe Answer	2	
Attach Request	3	Forwarding & Link Management
Attach Answer	4	
Store Request	7	Storage
Store Answer	8	
Fetch Request	9	Storage
Fetch Answer	10	
Find Request	13	Storage
Find Answer	14	
Join Request	15	Topology Plugin
Join Answer	16	
Leave Request	17	Topology Plugin
Leave Answer	18	
Update Request	19	Topology Plugin
Update Answer	20	
Route Query Request	21	Topology Plugin
Route Query Answer	22	
Ping Request	23	Forwarding & Link Management
Ping Answer	24	
Stat Request	25	Storage
Stat Answer	26	
App Attach Request	29	Forwarding & Link Management
App Attach Answer	30	
Configuration Update Request	33	Forwarding & Link Management
Configuration Update Answer	34	

FIGURA 10: Tipos de mensajes de RELOAD y módulos responsables

Así, a modo de ejemplo, para un mensaje Join Request, se generaría el siguiente paquete IP:

<table>
<tr><td colspan="5">TOPOLOGY PLUGIN</td><td>JOIN REQ</td><td></td></tr>
<tr><td colspan="4">MESSAGE TRANSPORT</td><td>HEADER</td><td>BODY</td><td>FOOTER</td></tr>
<tr><td colspan="4">FORWARDING & LINK MANAGMENT</td><td colspan="3">RELOAD MESSAGE</td></tr>
<tr><td colspan="3">OVERLAY LINK LAYER</td><td>FH</td><td colspan="3">RELOAD MESSAGE</td></tr>
<tr><td colspan="2"></td><td>TLS HEAD.</td><td colspan="4">ENCRYPTED APPLICATION DATA</td></tr>
<tr><td></td><td>TCP HEAD.</td><td colspan="5">TCP DATA</td></tr>
<tr><td>IP HEADER</td><td colspan="6">IP DATA</td></tr>
</table>

FIGURA 11: Desglose del paquete (el nivel de Forwarding no aporta cabecera)

Cuando se recibe un nuevo paquete, cada nivel procesa su cabecera, para posteriormente eliminarla y entregar al nivel superior los datos que a este le interesan. En sentido contrario, cuando se desea enviar un mensaje, las capas deben poner su cabecera antes de pasar la información a su nivel inferior.

A continuación se presenta una imagen de Wireshark en la que se ha capturado un paquete de RELOAD. Es un paquete real de la implementación que se ha realizado para el presente trabajo.

Pueden observarse detallados los diferentes campos del mensaje Attach Request. Dado que no se ha implementado la seguridad, en el campo Security Block se envían dos certificados obtenidos del fichero de configuración y una firma vacía, salvo por el identificador del nodo que crea el mensaje.

El destino es un recurso con ID 0xd7ae7ea9bf56ebdc60d222a046176d06, tal y como puede observarse en la imagen. En el primer Attach que envía un nodo el destinatario será siempre un Resource-ID, pues no se conoce el identificador de ningún nodo de la red.

No se visualizan el bloque de Message Contents por cuestiones de espacio en la figura. Sin embargo, en el capítulo 5: Resultados se dará más información acerca de este tema y se detallará el contenido de tres tipos de mensaje.

```
Filter: reload                                    Expression...  Clear  Apply  Save

No.   Time          Source          Destination     Protocol  Length  Info
   6 0.022979000   192.168.1.15    192.168.1.70    RELOAD     1269 Attach Request
   8 0.158876000   192.168.1.70    192.168.1.15    RELOAD     1304 Attach Response
  15 0.199647000   192.168.1.13    192.168.1.15    RELOAD     1304 Update Request
  17 0.255607000   192.168.1.15    192.168.1.13    RELOAD     1246 Update Response

⊞ Ethernet II, Src: Microsof_1d:3f:17 (00:03:ff:1d:3f:17), Dst: 74:f0:6d:a2:25:c1
⊞ Internet Protocol Version 4, Src: 192.168.1.15, Dst: 192.168.1.70
⊞ Transmission Control Protocol, Src Port: bsquare-voip (1071), Dst Port: distinct (9999)
⊟ REsource LOcation And Discovery
  ⊟ ForwardingHeader: Attach Request
      relo_token (uint32): 0xd2454c4f
      overlay (uint32): 0x9aa32b8d
      configuration_sequence (uint16): 22
      version (uint8): Unknown (0x0a)
      ttl (uint8): 30
    ⊞ fragment (uint32): 0xc0000000 (Fragment) (Last)
      length (uint32): 1215
      transaction_id (uint32): 0xb0181d8e25a0c0bb
      max_response_length (uint32): 0
        [Response length not restricted]
      via_list_length (uint16): 0
      destination_list_length (uint16): 19
      options_length (uint16): 0
    ⊟ destination_list (Destination<19>): 1 elements
      ⊟ Destination: resource
          type (DestinationType): resource (0x02)
          length (uint8): 17
        ⊟ resource_id (ResourceId<16>)
            length (uint8): 16
            data (bytes): d7ae7ea9bf56ebdc60d222a046176d06
  ⊞ MessageContents
  ⊟ SecurityBlock
    ⊟ certificates (GenericCertificate<1098>): 2 elements
        length (uint16): 1098
      ⊟ GenericCertificate
          type (CertificateType): X.509 (0)
          length (uint16): 1080
        ⊞ certificate
      ⊟ GenericCertificate
          type (CertificateType): X.509 (0)
          length (uint16): 12
        ⊞ certificate
    ⊟ signature (Signature)
      ⊟ algorithm (SignatureAndHashAlgorithm)
          hash (HashAlgorithm): SHA1 (2)
          signature (SignatureAlgorithm): RSA (1)
      ⊟ identity (SignerIdentity)
          identity_type (SignerIdentityType): cert_hash (1)
          length (uint16): 18
        ⊞ identity (SignerIdentityValue[18])
      ⊞ signature_value (opaque<0>)

0000  74 f0 6d a2 25 c1 00 03  ff 1d 3f 17 08 00 45 00   t.m.%... ..?...E.
0010  04 e7 17 be 40 00 80 06  5a ad c0 a8 01 0f c0 a8   ....@... Z.......
0020  01 46 04 2f 27 0f 94 9a  95 81 27 12 3e 56 50 18   .F./'... ..'.>VP.
0030  ff ff 2d 4d 00 00 d2 45  4c 4f 9a a3 2b 8d 00 16   ..-M...E LO..+...
0040  0a 1e c0 00 00 00 00 00  04 bf b0 18 1d 8e 25 a0   ........ ......%.
0050  c0 bb 00 00 00 00 00 00  00 13 00 00 02 11 10 d7
```

FIGURA 12: Detalle de un mensaje Attach Request

2.5 OTROS ASPECTOS DE RELOAD

2.5.1. Encaminamiento

RELOAD utiliza un encaminamiento simétrico recursivo. La simetría implica que el camino a la vuelta es el mismo que a la ida, invertido. Para ello, se permite realizar encaminamiento en origen.

La cabecera Forwarding Header lleva el campo DESTINATION LIST, que es un *array* de direcciones destino. Un mensaje será encaminado en estricto orden a través de los nodos que aparecen en la lista. Otro campo relevante es el VIA LIST, que es un segundo *array* al que se van añadiendo los nodos que atraviesa el mensaje a cada salto.

Cuando un nodo reenvía un mensaje coloca al final de la VIA LIST el *peer* que le entregó el paquete. Nótese que Forwarding Header no lleva un campo de dirección origen, en esta cabecera no se transmite el nodo que nos entrega el mensaje (último salto) ni tampoco el nodo que generó el mensaje. Sin embargo, cuando un nodo crea un mensaje lo ha de firmar, por lo que hallaremos su Node-ID en el Security Block. En lo que respecta al nodo que nos entrega el mensaje, no es necesario transmitirlo, sino que podemos consultarlo en nuestra tabla de conexiones.

El encaminamiento recursivo simétrico consiste en que cuando un nodo destino recibe una petición y debe responderla, este genera una DESTIONATION LIST en el mensaje respuesta, dando la vuelta a la VIA LIST. Es decir, pongamos un ejemplo en el que A envía una petición a D sin usar encaminamiento en origen. Su DESTINATION LIST deberá ser [D] y su VIA LIST estará inicialmente vacía.

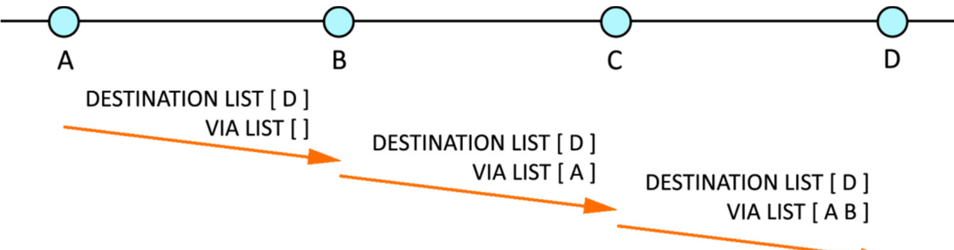


FIGURA 13: Camino de una petición

La red encamina el paquete y el camino de ida ha resultado ser: A-B-C-D. Cuando D recibe de C un paquete con VIA LIST [A B], lo primero que hace es añadir a esta lista su *peer*: C, quedando finalmente la VIA LIST [A B C]. Como el mensaje es para D, él procesa la respuesta, cuya DESTINATION LIST será necesariamente [C B A], es decir, la VIA LIST invertida.

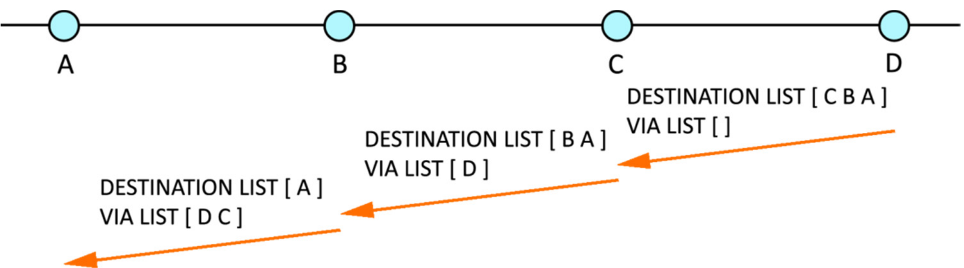


FIGURA 14: Camino de la respuesta

2.5.2. Clientes

RELOAD es un protocolo que se define idénticamente para clientes y para *peers*. Los clientes son aquellos nodos que no tienen responsabilidad de encaminar ni de almacenar. Un cliente solo es capaz de enviar mensajes a aquellos a los que está directamente conectado, puesto que no encamina para los demás. Cuando un nodo quiere entrar a formar parte de la red envía un Attach, lo que le convierte automáticamente en un cliente. Podemos considerar

que los clientes son aquellos nodos que aún no han enviado mensajes Join ni Update, y puede que nunca lo hagan si no quieren convertirse en *peers*.

2.5.3. Chord en RELOAD

A diferencia del diseño original de Chord, que contaba con un único sucesor y predecesor, RELOAD considera más eficiente que existan, al menos, tres sucesores y tres predecesores. Estos nodos próximos configuran la tabla de vecinos. La tabla de rutas de RELOAD, cuando se utiliza Chord como algoritmo, es conceptualmente la unión de la tabla de rutas y de la tabla de vecinos.

El identificador de nodo en RELOAD se denomina Node-ID y el de clave se llama Resource-ID. Ambos están formados por números enteros de 128 bits.

El identificador de recurso se genera a partir de una cadena de caracteres; por ejemplo, cuando se usa SIP, el Resource-ID se genera aplicando la función de *hash* al String que conforma el AOR (Address Of Record), que es la URI que identifica al usuario.

2.5.4. Seguridad

RELOAD implementa seguridad en tres niveles: mensaje, objeto y conexión. Para ello se firma cada mensaje y cada nodo almacenado. Gracias al uso de TLS y DTLS se proporciona una determinada seguridad en las conexiones.

2.5.5. Transversal NAT

Las operaciones para atravesar NAT son una parte fundamental del diseño de RELOAD. El protocolo está pensado para funcionar en situaciones en las que la mayoría de nodos están detrás de NAT o de cortafuegos. Hace uso de Interactive Connectivity Establishment (ICE) para establecer nuevas conexiones de RELOAD y de las aplicaciones que funcionan como usos en el nivel superior del protocolo. ICE trabaja conjuntamente con TURN y STUN.

TURN (Traversal Using Relays around NAT) permite que un nodo intermedio con una dirección IP pública ayude a otros nodos a comunicarse entre sí retransmitiendo sus mensajes. TURN difiere de otros protocolos de control de retransmisión en que permite a un cliente comunicarse con múltiples *peers* utilizando una sola dirección. TURN fue diseñado para usarse como parte de ICE, aunque puede funcionar sin él [MMR10]. Un nodo de RELOAD puede hacer de servidor TURN gracias al uso que se define a tal efecto en el borrador.

STUN (Session Traversal Utilities for NAT) sirve como herramienta a otros protocolos como ICE para tratar con Transversal NAT [MMR10].

2.5.6. Herramientas de trabajo

Para realizar la implementación y las pruebas del código se ha usado el programa gratuito jGrasp, que es un entorno de desarrollo liviano, que permite una visualización estructurada del código, así como la generación los esquemas UML de forma bastante automática –no así la colocación espacial de los bloques, que es completamente manual–.

Junto con lo anterior, se ha utilizado el kit de desarrollo de software Java Development Kit (JDK) 1.7, de Oracle, distribuido bajo licencia Sun, siendo la mayor parte de ella GNU General Public License (GPL).

El analizador de protocolos Wireshark, software libre mantenido bajo licencia GPL, se ha utilizado como herramienta para comprobar la correcta formación de los mensajes.

Por último, la edición de las figuras ha sido posible gracias a los programas Adobe Illustrator, Adobe Photoshop y Microsoft Visio, mientras que la memoria ha sido redactada y maquetada en Microsoft Word.

Estas aplicaciones funcionan gracias al sistema operativo Microsoft Windows 7. Windows Virtual PC ha permitido formar máquinas virtuales con Windows XP.

Capítulo

Objetivos

3.1 OBJETIVOS GENERALES

Con la realización de este proyecto, a lo largo de diferentes fases, se persigue una implementación fiel de RELOAD que pueda funcionar como una aplicación comercial. En futuros trabajos se irán ampliando características hasta que se consiga una funcionalidad completa.

El estudiante ha de comprender el funcionamiento de las redes superpuestas, cómo estas han evolucionado hasta llegar al borrador actual de RELOAD, observar qué ventajas tienen frente a los modelos tradicionales y qué aplicaciones pueden sacarles un mayor partido.

Pese a que la especificación de RELOAD es larga y compleja, el principal objetivo es probar la capacidad del alumno para realizar una aplicación real basada en estándares. Para ello, deberá familiarizarse con los documentos producidos por las organizaciones de normalización de internet, como el IETF. El proyectando deberá utilizar todos los conocimientos en redes y en programación adquiridos en estos años de carrera, ponerlos en práctica y conseguir una funcionalidad suficiente que permita realizar pruebas, con las cuales se puedan evaluar los hitos conseguidos.

Con el Trabajo Fin de Grado se persigue demostrar la capacidad de resolución de problemas en entornos nuevos, gestionar la información de manera eficiente, presentarla de forma clara y concisa, y ser capaz de aprender de forma autónoma, lo que permitirá una adaptación al mercado laboral en un sector en constante cambio.

Esta es la primera entrega, en la cual se ha determinado un subconjunto mínimo para obtener una implementación básica que se pueda probar. Las siguientes partes consistirán en finalizar el trabajo, de forma que se complete el desarrollo y sea capaz de operar con otras implementaciones.

3.2 TRABAJO ACTUAL

Originalmente, cuando el director del trabajo propuso este Proyecto Fin de Carrera en 2009, la idea era que un alumno pudiera desarrollar el protocolo de forma completa. Sin embargo, debido a cómo había crecido esta especificación en los años anteriores, se tomó la decisión de que el proyectando debía hacer una selección de determinadas partes que supusieran una carga de trabajo en torno al 50% del total. Quedaría en manos de otro futuro estudiante continuar este trabajo hasta completarlo.

Sin embargo, debido a las fuertes relaciones que los diferentes módulos tienen entre sí, enseguida se constató que no era sencillo hacer algo como eso. El requisito lógico de que la aplicación resultante funcionase de forma autónoma y se pudiera probar y, en consecuencia, evaluar solo se podía dar si todos los niveles estaban suficientemente implementados y funcionaban de forma conjunta. Incluso, y aunque no estaba previsto en un primer momento, se decidió codificar una pequeña parte del borrador de SIP draft-ietf-p2psip-sip-09 [SIP13], una fracción que permitiese monitorizar el funcionamiento de RELOAD con un uso real para el que fue diseñado.

Algunos de los componentes de RELOAD, como la seguridad, era razonable posponerlos, no solo porque dicha inclusión no afecta al funcionamiento de ninguno de los módulos, sino también porque una implantación demasiado temprana podría afectar negativamente al desarrollo de las pruebas. Por ejemplo, a la hora de realizar capturas de los paquetes por la red, utilizar una capa de transporte segura impedía ver el contenido de los mensajes. De hecho, las versiones más recientes del Wireshark son capaces de identificar el mensaje de RELOAD directamente sobre TCP y UDP, lo cual es significativo, porque el borrador en ningún caso se plantea siquiera esa posibilidad. Por esta razón, la implementación realizada funciona, por el momento, únicamente sobre TCP, al considerarlo más adecuado para señalización que UDP.

Otros aspectos, como los relacionados con Transversal NAT, tampoco han sido codificados en esta entrega. Se permite que algunos nodos estén detrás de NAT y otros tengan directamente una IP pública, pero el mapeo de puertos en el primer caso se realizará en el enrutador (*router*) de la forma tradicional.

No han sido tomadas en consideración la totalidad de mensajes de error que define RELOAD, y los que se transmiten no se receptan correctamente. Se ha preferido aunar esfuerzos en corregir errores de codificación o de comprensión de la especificación antes que en tener en cuenta comportamientos no esperados por parte de otros *peers* (que corran bajo otras implementaciones).

Una de las mayores limitaciones de esta primera entrega es que solo se garantiza el funcionamiento entre nodos que estén ejecutando todos ellos esta misma implementación. No es de extrañar si tenemos en cuenta que la seguridad es una parte fundamental del protocolo, y hasta que no se complete no será posible empezar a realizar las primeras pruebas de interoperabilidad.

En cualquier caso, hay que tener en cuenta que no han sido implementadas la totalidad de las funcionalidades de cada módulo. Se ha preferido incidir en aquellas que son utilizadas por SIP porque son las que se pueden probar, ya que este ha sido el uso que se ha codificado.

Aspectos implementados de RELOAD	Aspectos aún no implementados
– Uso SIP con funcionalidad mínima – Estructuras, Message Transport – Storage, con implementación íntegra de las estructuras de almacenamiento – Chord al completo – Forwarding & Link Management – Nivel de enlace con TCP	– Seguridad – UDP/DTLS, Simple Reliability – Transversal NAT – Tratamiento de errores – Clientes – Temporizadores, retransmisiones y determinadas funcionalidades – Framing Header

3.3 TRABAJOS FUTUROS

En las siguientes partes, el primero de los objetivos debe ser completar aquellos aspectos que faltan ya mencionados, como NAT transversal y la seguridad.

Otra función que no se ha considerado hasta el momento es la posibilidad de que haya nodos que no tengan la misma funcionalidad de los *peers*: los clientes. Estos nodos que no encaminan ni almacenan para otros no suponen un gran cambio en la implementación, pero es una funcionalidad requerida por RELOAD.

Como es natural, conforme avanza el desarrollo, se irán corrigiendo errores de la aplicación y completando funciones del protocolo. Y como corresponde a una implementación que en la actualidad tiene poca madurez, existen detalles que todavía no han sido abordados, como la temporización y las retransmisiones, que no afectan en condiciones normales, pero son necesarias para un correcto funcionamiento del programa.

Resulta muy importante centrarse en la interoperabilidad con otras implementaciones. Para ello, es imprescindible programar la gestión de errores que define RELOAD. Un uso adecuado de esta herramienta permite observar de qué forma reacciona nuestro programa cuando los mensajes que recibimos, o su orden, no es el esperado.

A este fin, se deben destacar los eventos donde diferentes implementadores de todo el mundo se juntan para comprobar el funcionamiento de su código. Poder asistir a un lugar así es una oportunidad única para realizar este hito de la interoperabilidad y conocer a otros desarrollares con los que intercambiar experiencias y resolver dudas. Por estas razones, este alumno ha confirmado su asistencia al próximo evento, que se celebrará en Berlín los días 27 y 28 de julio de 2013 [PETI12].

Sin embargo, si tenemos en cuenta que la meta del diseño de RELOAD es que pueda servir con una variedad de redes estructuradas, no es menos relevante proponer la creación de un segundo algoritmo que funcione conjuntamente con Chord. Esto permitirá comprobar que la estructura del protocolo es independiente de la topología en uso, garantizando que no es necesario modificar ninguno de los otros módulos para insertar un nuevo algoritmo.

En la actualidad, no se ha redefinido ninguna red *peer-to-peer* aparte de Chord. Si esto no cambia en el futuro próximo, un posible trabajo podría ser adaptar un algoritmo existente para su uso con RELOAD. Igualmente interesante podría ser diseñar un uso desde cero, ya sea un nuevo protocolo, o bien modificar uno ya existente, para que funcione en entornos distribuidos.

Otro asunto relevante es el del almacenamiento. En la implementación actual los datos almacenados se guardan en memoria RAM. Esto puede ser suficiente en un buen número de escenarios, pero en otras situaciones puede ser necesario que los datos se guarden en disco. Para ello, se propone la creación de un nuevo módulo que utilice bases de datos.

Por último, resulta conveniente probar la escalabilidad de la implementación con un número mayor de nodos usando un emulador de redes tipo ModelNet o PlanetLab. Debido al enorme número de computadores que pueden formar parte de este tipo de redes, determinadas aplicaciones que funcionan perfectamente en entornos controlados, pueden no escalar bien cuando aumenta exponencialmente el número de nodos. Una vez completado el código, es convenientemente realizar este tipo de pruebas, depurando, cuando sea posible, los problemas que se encuentren.

El alumno que escribe ha solicitado el poder continuar con el proyecto como Trabajo Fin de Máster de Ingeniería de Telecomunicación, que comenzará en septiembre. De esta manera, podrá dar una salida inmediata al trabajo.

Capítulo

Diseño e implementación

4.1 FASES DEL DESARROLLO

Estas son las fases en las que se ha dividido el diseño y la implementación de esta primera parte del proyecto.

4.1.1. Documentación

La primera etapa ha consistido en la documentación y la comprensión del protocolo. No se tenían conocimientos previos de redes P2P, por lo que ha sido necesario documentarse sobre estas y tratar de entender y asimilar el documento del IETF.

4.1.2. Codificación de estructuras

Como aún no se tenía una visión suficiente para poder hacer un diseño completo del protocolo, se optó por empezar a programar aquellas partes que sí estaban claras. La primera de ellas fue crear la estructura de los mensajes, que venían completamente definidos en un idioma similar a C. Así pues, la primera tarea fue traducir estas clases de C a Java. Debido a la orientación a objetos de Java, algunas decisiones fueron inmediatas: cada estructura correspondería a una clase diferente de Java.

4.1.3. Diseño de Forwarding

Posteriormente, se deliberó acerca de qué módulo de RELOAD empezar a implementar. El problema consistía en que todos los módulos funcionan conjuntamente, por lo que no era posible implementar un módulo y probarlo individualmente. La decisión adoptada fue que Forwarding sería el núcleo de la implementación. En Forwarding estaría el bucle infinito que envía y recibe constantemente en todo protocolo de comunicaciones, y también en esta capa se crearían los diferentes hilos que componen la implementación. Esta decisión se tomó porque Forwarding está en un nivel adecuado para realizar esta

función. Es un módulo que tiene acceso al nivel inferior de enlace y, a su vez, tiene acceso directo a dos módulos superiores: Message y Topology, por lo que tiene control sobre la mayoría de los bloques.

4.1.4. Diseño y programación del nivel de enlace

Link fue sencillo de esbozar: básicamente es una clase con un *socket* en su interior. La idea del diseño es que los métodos de acceso (enviar, recibir…) fuesen idénticos, con independencia del nivel de transporte, que podría ser TCP, TLS o DTLS.

En este punto se programó el módulo de enlace, así como una primera versión muy básica de Forwarding, de modo que ambas capas funcionaran conjuntamente y se pudieran realizar unas pruebas sencillas.

4.1.5. Diseño y codificación de Message Transport

Message fue el siguiente módulo en diseñarse. Buena parte de este módulo ya estaba implementado, lo cual es lógico si tenemos en cuenta que la mayor parte de las cabeceras y estructuras generadas en el segundo punto pertenecen precisamente a esta capa.

La idea aquí era simplemente tener una clase que fuera llamada por Forwarding, que descodificara los paquetes haciendo uso de la estructura del mensaje (MessageStructure), obteniendo así la información del mensaje.

4.1.6. Diseño de Topology Plugin, codificación de Chord

La siguiente fase consistió en el diseño del módulo de Topology. Teniendo como requisito un módulo que inicialmente sería Chord, pero que debía ser intercambiable en el futuro por otras redes. Se diseñaron y codificaron, en primer lugar, las tablas de conexiones y de rutas, que se programarían como

sencillas listas o *arrays*. Y en segundo lugar, el resto de clases que permitieran el funcionamiento del algoritmo.

4.1.7. Diseño y programación de Storage

Tocaba la implementación del último gran bloque: Storage. Lo más importante era crear las estructuras de almacenamiento de datos. Para ello se crearon determinadas tablas, que contendrían básicamente elementos similares a los que tienen algunos mensajes de la clase, como Store Request.

4.1.8. Codificación de Forwarding y tareas

Con los diferentes módulos completados, se terminó de programar Forwarding según se había diseñado en 4.1.3. Sin embargo, se detectó una carencia importante: la imposibilidad de mantener el estado de cualquier tipo de mensaje enviado. Por ejemplo, al enviar un PingReq, es preciso recordar que hay que recibir un PingAns. Para ello, se creó un nuevo concepto llamado “tarea”, y se crearon tres nuevos hilos para que tres de estas tareas pudieran ejecutarse en paralelo. Cada tarea corresponde a uno de los principales módulos: Forwarding, Topology y Storage. De esta manera, se cerraba el diseño del módulo Forwarding & Link Management.

4.1.9. Codificación de Reload Interface y del uso (SIP)

Tras haber completado la implementación de todos los módulos de RELOAD, el último paso fue la creación de una API global. El desafío era identificar qué funciones debían ser accesibles por los usos y mapear dichas funciones a los módulos pertinentes. Por ejemplo, la función *store()* de la clase ReloadInterface, simplemente es una llamada a la función *store()* de la clase StorageInterface.

Por último, y para comprobar el funcionamiento de todo lo realizado hasta el momento, se implementó una versión reducida de SIP como uso, tomando como base el borrador existente [SIP13].

4.2 VISTA GENERAL

4.2.1. Organización del código

Existen 7 paquetes principales de Java, con cada uno de los módulos que conforman RELOAD: el nivel de enlace, Fowarding and Link Management, Topology Plugin, Message Transport, Storage, uno para clases comunes a los diferentes módulos y, por último, uno para los paquetes específicos del uso en cuestión, en este caso SIP. A su vez, estos podrán contener más subpaquetes.

FIGURA 15: Organización de los paquetes

El código busca aprovechar al máximo la orientación a objetos de Java. Para ello, cada estructura definida en el borrador se codifica en una clase diferente. Es una novedad de esta implementación que, en lugar de utilizar mecanismos clásicos como *parsers* para codificar y descodificar las estructuras, se ha preferido diseñar un nuevo procedimiento.

Cada estructura se modela de la forma más sencilla posible: cada uno de los campos se codifica con un atributo, que podrá ser un tipo básico de Java u otra estructura (es decir, un objeto). Cada estructura tiene, de forma general, dos constructores: el primero, cuyos argumentos coinciden con los atributos de la clase y cuya asignación es, por tanto, trivial; y un segundo constructor cuyo único argumento es un *array* de bytes y que igualmente rellena los atributos, tras la consiguiente descodificación de la estructura.

El primero de los constructores tiene como objetivo la formación de un mensaje que será enviado por la red, mientras que el segundo se utiliza cuando se recibe un paquete de la red y necesitamos saber su contenido. La

descodificación es recursiva: cada clase descodifica su estructura y, si en su interior hay atributos que no sean tipos básicos, a su vez llama a los constructores de dichas estructuras. La codificación se realiza llamando al método *getBytes()* y es igualmente recursiva.

```
public class Signature{

    private SignatureAndHashAlgorithm algorithm;
    private SignerIdentity identity;
    private Opaque signature_value; //<0..2^16-1>

    public Signature (SignatureAndHashAlgorithm algorithm, SignerIdentity identity,
                      byte[] signature_value) throws ReloadException{

        this.algorithm = algorithm;
        this.identity = identity;
        this.signature_value = new Opaque(16, signature_value);

    }

    public Signature (byte[] data) throws Exception{

        algorithm = new SignatureAndHashAlgorithm(data); // Size: 2 bytes
        identity = new SignerIdentity(Utils.cutArray(data, 2));

        int lengthIdentity = identity.getBytes().length;
        signature_value = new Opaque(16, data, lengthIdentity+2);

    }

    public byte[] getBytes() throws IOException{

        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        baos.write(algorithm.getBytes());
        baos.write(identity.getBytes());
        baos.write(signature_value.getBytes());

        return baos.toByteArray();

    }
```

FIGURA 16: Ejemplo de estructura sencilla: campo Signature de Security Block

4.2.2. Llamadas entre los módulos

Se ha realizado un diseño modular poniendo énfasis en la independencia de las distintas capas de la red superpuesta.

Cada paquete tiene una clase, que es la Interfaz de Programación de Aplicaciones (API), o simplemente *interfaz*, que es la única forma de acceso desde los otros módulos. Esto simplifica el funcionamiento y garantiza que, por ejemplo, todas las peticiones que pueden realizarse al módulo de almacenamiento se dirijan únicamente a la clase StorageInterface. Sin embargo, internamente en cada capa no hay restricciones sobre cómo las clases se comunican entre sí. Esto permite, por un lado, que el funcionamiento interno de cada módulo sea independiente y, por otro, llevar un control sobre cómo unas partes del código se comunican con otras.

Se han creado interfaces en todos los niveles, en aquellas capas propiamente de RELOAD: Forwarding And Link Managment Interface (FALMI), Topology Plugin Interface (TPI), Message Transport (MT) y Storage Interface (SI); y en el nivel de enlace la red superpuesta: Link Interface (LI).

Igualmente se define una clase que agrupa a todos los módulos del protocolo: Reload Interface. Cualquier uso que se defina para funcionar sobre RELOAD únicamente tendrá que crear un objeto de tipo ReloadInterface y realizar llamadas a los métodos de los que consta esta clase. Las interfaces enmascaran la complejidad de cada uno de los módulos y de la aplicación en su conjunto. Esto facilita la tarea a programadores que quieran utilizar el código, al quedar una API muy sencilla.

Una inspiración para esta estructuración del código en interfaces es la programación basada en *sockets*, que enmascara la torre de protocolos TCP/IP en una interfaz de baja complejidad. Se ha pretendido realizar algo análogo en Reload Interface, que proporcionara una API de alto nivel.

Otro protocolo de características similares que define una interfaz de acceso es P2PP, en cuyo borrador aparece “P2PP API” como método de acceso desde la aplicación hacia la red superpuesta.

Aunque se ha querido aprovechar al máximo la orientación a objetos para simplificar la programación, la forma en la que los distintos módulos se comunican entre sí ha requerido la creación de una clase que define como estáticas a las principales interfaces: FALMI, TPI, MTI y SI. Esas cuatro interfaces son las únicas clases estáticas de toda la aplicación, con el fin de que puedan ser accedidas indistintamente por los niveles superiores e inferiores.

4.2.3. Aclaraciones

Para evitar confusiones, hay que especificar que cuando nos referimos a un nodo como cliente o servidor, puede serlo en tres aspectos:

- En función de quién inició la **conexión**: si es entrante se comporta como servidor; si es saliente, como un cliente. Esto es relevante a nivel de enlace de la red superpuesta al utilizar Socket y ServerSocket.

- En función del **rol** en las comunicaciones: cuando enviamos peticiones y esperamos respuestas, actuamos como cliente; si es al contrario, el rol del nodo es de servidor.

- Y a la vez en función de la **responsabilidad** en la red: existen nodos clientes y otros que son *peers*. En este último caso, no existe el servidor, pero sí el cliente.

Cuando se usa Chord, un mínimo de 138 hilos realizan en paralelo diversas funciones en RELOAD. De los cuales, al menos 134 hilos asumen el rol de servidores y atienden a cada uno de los nodos que figuran en la tabla de conexiones. Otros 3 asumen el rol de clientes. Y el último hilo, que es el hilo principal de la aplicación –que se crea al ejecutar el método *main()*–, se dedica a procesar las órdenes que el uso en cuestión podría requerir (por ejemplo: SIP). Esta cifra no debería cambiar significativamente si se utiliza otro tipo de algoritmo *peer-to-peer*.

4.3 CAPA DE ENLACE (LINK)

Transporte	Enlace	TLS	DTLS

FIGURA 17: La capa de enlace coincide con el nivel de transporte de internet

Link es un módulo cuya principal función es enmascarar las funciones de Java (Socket y ServerSocket) que colocan los bytes por la red.

La interfaz, Link Interface, básicamente cuenta con los métodos para enviar y recibir. Esta capa está diseñada para usarse con tres protocolos: TCP, TLS y DTLS. La idea es que el protocolo de transporte sea transparente para la capa superior, de tal forma que usar uno u otro no modifique las llamadas que se intercambian los módulos Link y Forwarding.

Solo se ha implementado con TCP, de forma que si se intenta utilizar cualquiera de los otros saltará la excepción correspondiente.

Las tareas de este módulo son:

- Crear y mantener las conexiones abiertas, tanto en sentido entrante (cuando el nodo actúa como servidor) como en sentido saliente (cuando el nodo actúa de cliente).
- Enviar mensajes a la red.
- Quedarse bloqueado a la espera de recibir un mensaje de la red, identificar donde acaba el paquete, comprobar si es del protocolo RELOAD y, si es así, pasarlo al nivel superior como un *array* de bytes.
- Cerrar las conexiones.

Una de las características de este módulo es que una ejecución de la aplicación no crea una única instancia de Link, sino que crea tantas como conexiones existan o, lo que es lo mismo, tantas como *sockets* haya corriendo.

En particular, existen tantos objetos Link Interface como conexiones salientes (clientes) tenga abiertas el nodo, cada una con un objeto Socket. Por el contrario, un único objeto Link Interface (servidor) con la clase ServerSocket en su interior gestionará las conexiones entrantes.

Esta capa es igualmente responsable de la cabecera Framing Header, pero debido a que TCP no saca partido de ella, se ha decidido posponer su implementación.

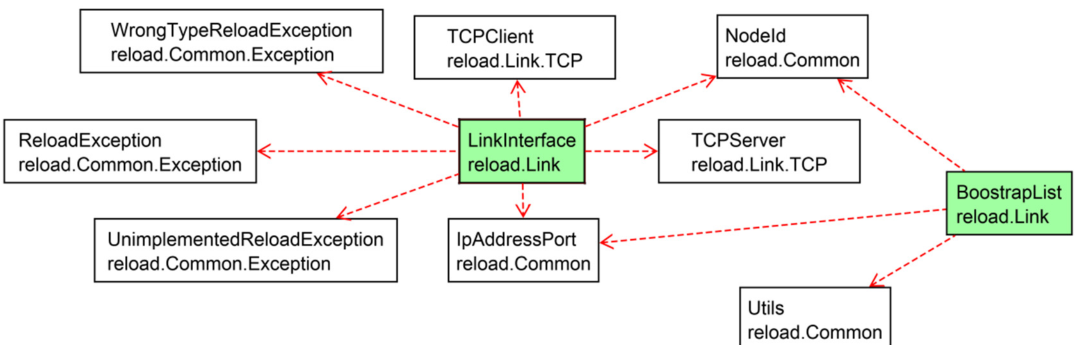
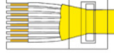
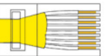


FIGURA 18: Diagrama XML de Link

Como puede verse en el diagrama, la capa de enlace no guarda relación con otros módulos, sino que únicamente se comunica con el paquete Common y con otras clases de su mismo nivel.





4.4 CAPA DE INTERRED: FORWARDING & LINK MANAGEMENT

Forwarding es el encargado de reenviar los mensajes a otros nodos de la red. Es quizá el módulo con mayor complejidad, pues maneja más de un centenar de hilos con todos los enlaces establecidos. También es el encargado de responder a mensajes como Ping, Attach…, e igualmente de mantener el estado de aquellas comunicaciones para las cuales está esperando respuesta.

Esta capa recibe los mensajes de Link, creando un objeto Message que le permite obtener las cabeceras del paquete. En base a las direcciones destino, comprueba si el paquete es para este nodo. Si lo es, lo entrega a la capa Message Transport y, en caso contrario, tras hacer pequeñas modificaciones en la cabecera, lo entrega a la capa de enlace para que sea reenviado.

Este módulo se encarga de:

- Consultar al complemento de topología para saber cuál es el siguiente salto del mensaje.
- Mantener los 128 hilos –en Chord– que atiendan de la forma más transparente posible a cada una de las conexiones con los *fingers*; y un mínimo de 6 hilos que hagan lo propio con los nodos vecinos (se requieren, como mínimo, tres sucesores y otros tres predecesores).
- Almacenar las referencias los objetos Link Interface, tantas como clases Socket o ServerSocket de Java haya en funcionamiento.
- Gestionar el hilo que realiza las tareas de “cliente” propias de los mensajes que pertenecen a Forwarding & Link Management (consultar figura 10).
- Llamar a Message Transport para instanciar objetos Message y entregar a este nivel la ejecución si el mensaje es para este nodo.

Clases relevantes:

- ForwardingAndLinkInterface: la interfaz de esta clase tiene los métodos relativos a las peticiones (*requests*) de las que esta clase es responsable, como Attach y Ping. También tiene métodos importantes, como *crear conexión*, para lo cual registra el nuevo nodo en Topology Plugin y crea la conexión llamando a Link; *enviar*, que es capaz de enviar un mensaje eligiendo automáticamente qué *peer* es el adecuado para el siguiente salto, aunque también permite seleccionar el nodo en particular al que queremos entregar el paquete; y *cerrar*, que cierra una conexión.

- ForwardingCheck: tiene métodos de uso general en las clases internas del módulo. Por el momento, su único método es *comprobar mensaje*, que identifica para qué nodo es el mensaje.

- ReloadThread: es el hilo “servidor” que recorre todos los niveles de RELOAD, en ocasiones normales con Chord hay un total de 134. En esta capa realiza tareas esenciales como tener el bucle infinito que aparece en todo servidor de comunicaciones, dentro del cual se recibe una petición y se envía la correspondiente respuesta. Cada hilo se asocia a un determinado objeto de la capa Link, por lo que cada uno de ellos recibirá los mensajes siempre del mismo vecino. Estos hilos actúan como un servidor de baja complejidad, puesto que reciben siempre peticiones básicas (mensajes Request) a la que responden con una única respuesta (mensajes Answer).

- Otras menos relevantes son **SendData**, que almacena información pendiente de enviar en un hilo que aún no se ha creado; **ForwardingLoop**, que crea nuevos hilos dinámicamente según se van necesitando e **Inicialization**, que contiene una serie de sentencias que se ejecutan de forma previa al arranque de la aplicación.

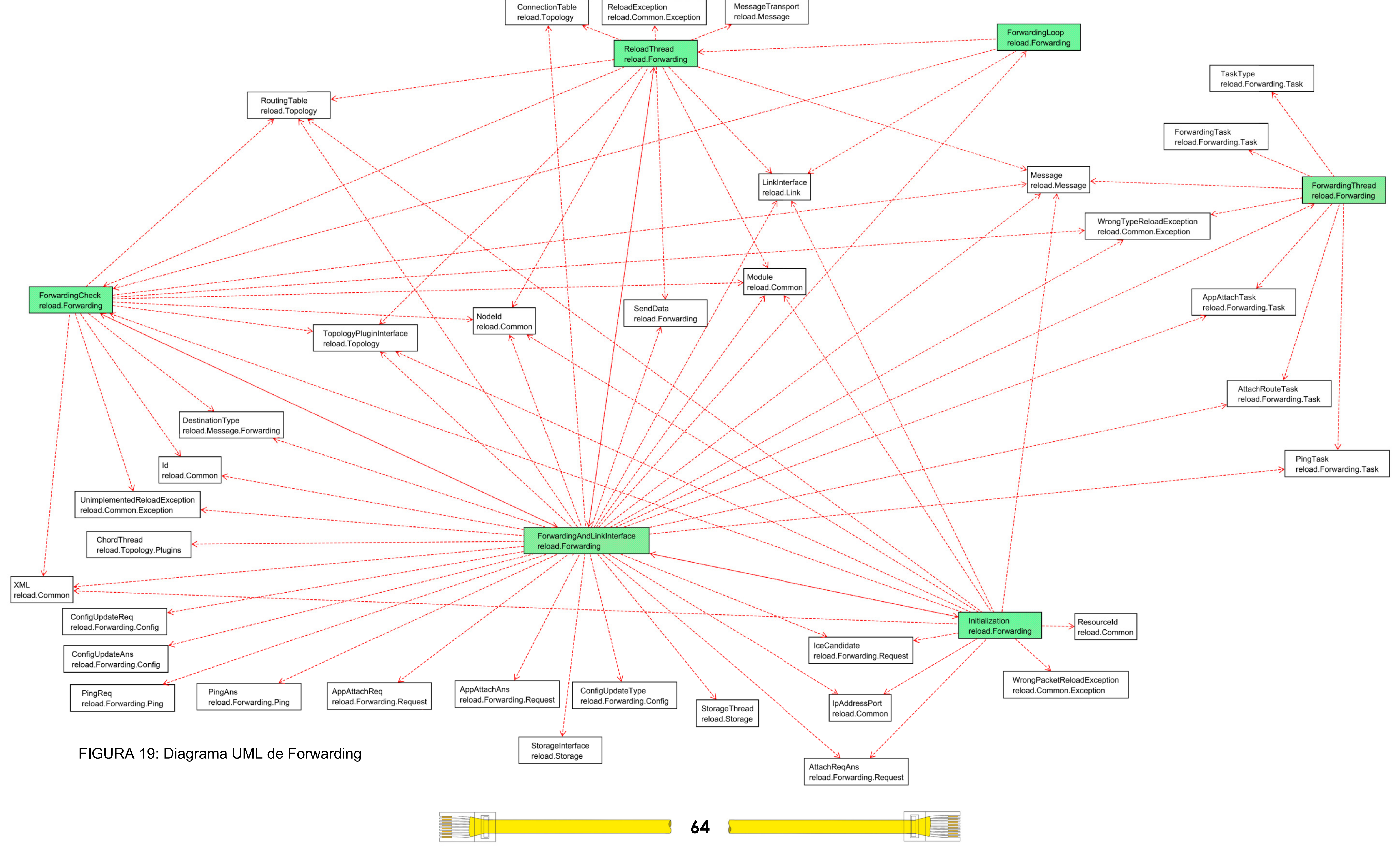


FIGURA 19: Diagrama UML de Forwarding

4.5 CAPA DE TRANSPORTE: MESSAGE TRANSPORT

Message Transport es el responsable del mensaje de RELOAD. Es un módulo sencillo con tres partes bien diferenciadas:

- **Message Structure** es aquella que contiene toda la estructura del mensaje. Está codificada en más de 40 clases, repartidas en tres paquetes: Forwarding Header, Message Contents y Security Block.

- **Message Transport** es la interfaz típica del módulo, análoga a LI, a FALMI, a TPI y a SI, en la cual se atienden peticiones de niveles inferiores y superiores. Consta de un *switch* central, que a partir del código enviará el contenido del mensaje al módulo pertinente. Excepcionalmente, no es la única interfaz en Message Transport.

- **Message** se trata de un objeto que puede ser instanciado por cualquier módulo, por lo que puede considerarse otra interfaz. Crea un nuevo mensaje, por lo que internamente contiene un Message Structure, pero no todos los campos de la estructura son accesibles desde Message.

Las decenas de clases que conforman la estructura del mensaje están codificadas tal cual se definen en el borrador del IETF. En dicho documento aparecen definidas con una sintaxis tipo C, por lo que el trabajo de esta parte ha consistido en portar del idioma C a Java.

Cada estructura de C se codifica en una clase diferente, esto es, se instancia en un objeto de Java. Cada clase tiene como atributos los distintos campos de la estructura –utilizando, siempre que sea posible, tipos básicos para codificarlos– e implementa, como mínimo, dos constructores. El primero de ellos se usa para construir un mensaje que se va a enviar por la red, por lo que sus argumentos coinciden con los atributos de la clase. El segundo constructor se utiliza para los paquetes que llegan desde la red hacia el nodo, por lo que tiene como único

argumento un *array* opaco de bytes, que debe ser descodificado para que los atributos puedan rellenarse con los valores obtenidos.

Message nace de la necesidad de tener un objeto cuyos atributos son determinados campos del mensaje de RELOAD. Ante una limitación bien conocida de Java: que solo permite devolver un único objeto en cada método, se decidió implementar una clase que contuviera como atributos el cuerpo del mensaje (MESSAGE BODY) y el código (MESSAGE CODE). Sin embargo, se ha generalizado de manera que ahora cuenta con un mayor número de campos y las diferentes partes de RELOAD lo intercambian y lo utilizan en sus métodos.

Cuando Forwarding & Link Managment recibe un paquete de la red, debe examinar algunos campos del mensaje, tales como la DESTINATON LIST. Si este nodo es el responsable del primer ID de la lista, pasa el hilo de ejecución al nivel superior, a Message Transport; en caso contrario, reenvía el mensaje al nodo apropiado. Sin embargo, para poder tomar la decisión es necesario obtener algunos campos de la estructura del mensaje: es ahí donde entra en juego Message. Cualquier capa que lo requiera puede crear un objeto Message.

En cualquier caso, esta decisión no afecta a la modularidad del diseño, puesto que cualquier instancia de la clase Message es equivalente a una consulta que un módulo realiza a Message Transport. Responde, más bien, a la realidad de que RELOAD tiene una estructura que es común para todos los niveles.

En esta clase destaca:

- MessageTransport: tiene dos métodos importantes: *enviar*, que es llamado cuando otro módulo quiere entregar un mensaje a la red, y cuya petición se derivará al método *enviar* de Forwarding; y *procesar*, cuando esta capa recibe un paquete de la red, detectará el tipo de mensaje, lo pasará al módulo encargado y, si es una petición, le devolverá la respuesta.

ErrorUnknownKind reload.Common.Error
ErrorGenerationCounterTooLow reload.Common.Error
ErrorDataTooOld reload.Common.Error
ErrorResponse reload.Message.MessageCont
StorageInterface reload.Storage
Module reload.Common
KindArray reload.Common
TopologyThread reload.Topology
KindId reload.Common
ErrorForbidden reload.Common.Error
ForwardingThread reload.Forwarding
ErrorNotFound reload.Common.Error
TopologyPluginInterface reload.Topology
StorageThread reload.Storage
ForwardingAndLinkInterface reload.Forwarding
MessageTransport reload.Message
RoutingTable reload.Topology
UnimplementedReloadException reload.Common.Exception
Id reload.Common
WrongTypeReloadException reload.Common.Exception
NodeId reload.Common
GenericCertificate reload.Message.Security
SignerIdentity reload.Message.Security
DestinationData reload.Common
CertificateType reload.Message.Security
SignerIdentityType reload.Message.Security
HashAlgorithm reload.Message.Security.TLS
Message reload.Message
SignatureAlgorithm reload.Message.Security.TLS
SignatureAndHashAlgorithm reload.Message.Security.TLS
ReloadError reload.Common.Error
Signature reload.Message.Security
Destination reload.Common
DestinationType reload.Common
ReloadException reload.Common.Exception
SignerIdentityValue reload.Message.Security
MessageContents reload.Message.MessageCont
XML reload.Common
ForwardingHeader reload.Message.Forwarding
SecurityBlock reload.Message.Security
MessageStructure reload.Message
Utils reload.Common
WrongLengthReloadException reload.Common.Exception

FIGURA 20: Diagrama UML de Message Transport (no todas las estructuras)

4.6 CAPA DE TRANSPORTE: STORAGE

Storage debe proporcionar un almacenamiento para los datos, así como la capacidad para obtenerlos en las consultas que se le hagan. Esta información se almacena en memoria RAM.

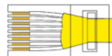
Sus clases principales son:

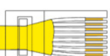
- StorageInterface: gestiona las peticiones propias de la capa de almacenamiento. A través de esta interfaz se proporciona acceso desde fuera del módulo a las tablas y listas que se enumeran a continuación.

- SingleValueTable: esta es la tabla que se utiliza si el Data Model en uso es el de valor individual. Si el uso define que este Data Model es único para toda la aplicación, las otras dos tablas estarán siempre vacías. Sus métodos permiten añadir, eliminar y obtener datos. Cada entrada de la tabla relaciona un Kind-ID con el tiempo de almacenamiento, el tiempo de vida, un *flag* que indica si el dato existe y el valor almacenado en sí mismo.

- ArrayTable: esta tabla se utiliza si el Data Model es *array*. Es idéntica al caso del valor individual, salvo por la presencia de un índice numérico. Varias entradas de la tabla pueden tener el mismo Kind-ID, siempre y cuando tengan diferentes índices. Dado el Kind-ID y el índice, el valor a devolver es único.

- DictionaryTable: análogamente a los casos anteriores, cuando la información almacenada tiene un diccionario como DataModel, se almacena en esta tabla. La clave que indexa los valores es una secuencia de bytes.





- KindModelList: almacena el Data Model para cada Kind-ID. También permite asignar un mismo Data Model para toda la aplicación, evitando tener que codificar cada relación de forma repetitiva.

- ResourceReplicaList: mantiene la información acerca de qué recursos tienen almacenado un determinado Kind-ID y si esos datos son una réplica.

- KindCounterList: almacena el dato Generation Counter para cada Kind-ID conocido.

- DataStructure: se instancia para formar un objeto que contiene todos los campos de un valor almacenado. De esta forma es posible devolver una entrada completa de cualquiera de las tablas antes mencionadas en su método *getData()*.

Los datos se guardan utilizando estructuras de datos sencillas de Java. Se ha considerado que no era necesario utilizar tablas Hash porque dado que lo que almacenamos es información básica y no objetos, otros elementos como las listas de Java son válidos y cumplen su objetivo con una menor carga computacional.

Kind-ID*	Resource-ID*	Storage Time	Life Time	Exists	Value
1	20	12:57 06/07/2013	86400 s	True	ABCD
1	80	15:21 06/07/2013	86400 s	True	LMNO
1	64	16:48 06/07/2013	86400 s	True	WXYZ
2	20	18:57 06/07/2013	86400 s	True	1234

FIGURA 21: Ejemplo de tabla de almacenamiento de tipo Single Value

Cada una de las columnas de la tabla se codifica con un ArrayList, por lo que cada vez que se añade una nueva entrada (fila) a la tabla, se incrementa en una unidad el tamaño de cada uno de los *arrays*.

En este ejemplo, existen dos índices (*): el Kind-ID y el Resource-ID. Si realizamos una obtención de la tabla con ambos índices, el resultado será único (o nulo). Si realizamos una búsqueda solo con uno de ellos (solo con el Kind-ID o solo con el Resource-ID) obtendremos, en general, múltiples filas de la tabla.

Sin embargo, no es necesario que los valores almacenados en la tabla sean tipos básicos de Java. Es posible utilizar objetos siempre y cuando estos tengan una representación numérica única e inequívoca. Por ejemplo, el objeto Kind-ID es un entero de 32 bits y el Resource-ID es un entero de una longitud variable.

La tabla se recorre con iteradores, por lo que para buscar un determinado Kind-ID, se obtiene su valor entero (un Integer) y se compara con los valores enteros de cada uno de los Kind-IDs almacenados en la lista, dando en su caso una serie de aciertos. Para el caso de los Resource-ID, esta clase se tiene un método *equals()* que compara longitud y contenido de los identificadores de recurso en cada iteración.

Si hacemos un *getData()* con argumento Kind-ID 1, obtendremos las primeras tres filas de la tabla. Si lo hacemos con el Resource-ID 20, obtendremos las filas primera y última. Un *getData()* que contenga como atributos un Kind-ID y un Resource-ID da un resultado único (o nulo).

Solo es posible obtener información de la tabla buscando a través de los índices. Si bien el diseño de esta estructura de almacenamiento permitiría de forma análoga buscar cualquier otro campo, debido a los requisitos de RELOAD, se ha implementado únicamente esta forma de búsqueda.

A continuación se presenta un esquema UML del presente módulo.

Tal cual se desprende de la siguiente figura, Storage tiene relación con los paquetes de Message y de Topology, además del paquete Common, que es común para toda la implementación.

SingleValueTable reload.Storage

ResourceReplicaList reload.Storage

DataValue reload.Storage.Data

ResourceKindList reload.Storage

KindId reload.Common

DictionaryTable reload.Storage

WrongLengthReloadException reload.Common.Exception

Utils reload.Common

DictionaryKey reload.Storage.Data

TaskType reload.Storage.Task

StorageTask reload.Storage.Task

Message reload.Message

ReloadException reload.Common.Exception

ResourceId reload.Common

DataStructure reload.Common

StorageThread reload.Storage

ArrayTable reload.Storage

JoiningStoreTask reload.Storage.Task

WrongTypeReloadException reload.Common.Exception

AppFetchTask reload.Storage.Task

AppStoreTask reload.Storage.Task

Module reload.Common

Opaque reload.Common

ErrorDataTooOld reload.Common.Error

StorageInterface reload.Storage

DataModel reload.Common

StatKindResponse reload.Storage.Data

StoredDataSpecifier reload.Storage.Data

Signature reload.Message.Security

StoredDataValue reload.Storage.Data

StoredData reload.Storage.Data

FindReq reload.Storage.Data

ArrayEntry reload.Storage.Data

SignerIdentity reload.Message.Security

StatReq reload.Storage.Data

StoredMetaData reload.Storage.Data

Destination reload.Message.Forwarding

FindKindData reload.Storage.Data

FindAns reload.Storage.Data

StatAns reload.Storage.Data

DictionaryEntry reload.Storage.Data

ErrorNotFound reload.Common.Error

FetchAns reload.Storage.Data

StoreReq reload.Storage.Data

ErrorUnknownKind reload.Common.Error

KindModelList reload.Storage

SignerIdentityValue reload.Message.Security

NodeId reload.Common

FetchKindResponse reload.Storage.Data

ErrorGenerationCounterTooLow reload.Common.Error

FetchReq reload.Storage.Data

SignatureAlgorithm reload.Message.Security.TLS

StoreAns reload.Storage.Data

StoreKindResponse reload.Storage.Data

SignerIdentityType reload.Message.Security

StoreKindData reload.Storage.Data

RoutingTable reload.Topology

HashAlgorithm reload.Message.Security.TLS

KindCounterList reload.Storage

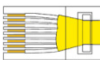
SignatureAndHashAlgorithm reload.Message.Security.TLS

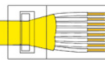
TopologyPluginInterface reload.Topology

UnimplementedReloadException reload.Common.Exception

ErrorForbidden reload.Common.Error

FIGURA 22: Diagrama UML de Storage





4.7 CAPA DE ENCAMINAMIENTO: TOPOLOGY PLUGIN

Dado que los complementos deben ser intercambiables, la base del diseño del Topology Plugin es una interfaz genérica en la que encajen las diferentes topologías. En esta interfaz –Topology Plugin Interface– se han incluido aquellos atributos y métodos comunes, de tal forma que pueda ser llamada por los otros módulos con independencia del algoritmo en cuestión que se esté usando.

Topology Plugin define las siguientes clases de forma genérica:

- RoutingTable: tabla de rutas, que en el caso de Chord está formada por la tabla de vecinos y la tabla de *fingers*. Permite insertar *peers* y también eliminarlos, amén de realizar consultas sobre ellos.

- ConnectionTable: tabla de conexiones en la que se mantiene un estado de todas las conexiones activas, tanto si estas son entrantes como si son salientes. Igualmente, se permite la consulta y modificación de los datos.

- Routing: el encaminamiento permite la consulta del siguiente salto; para ello, debe devolver el nodo elegido de entre los que conforman la tabla de rutas.

- TopologyPluginInterface es la interfaz ya comentada que proporciona acceso a todas las funciones desde otros módulos. Esta interfaz proporciona un acceso directo a las tres clases anteriores. Tiene como métodos los mensajes propios de Topology Plugin: Update, Join, Leave, Route Query y Probe.

Todo complemento ha de implementar el equivalente de estas clases en cada uno de los algoritmos implementados. Para el caso de Chord, debe existir una interfaz de acceso (ChordInterface), una clase con la tabla de rutas de Chord

(ChordRoutingTable), otra con la tabla de conexiones (ChordConnectionTable) y una última con el encaminamiento del algoritmo (ChordRouting). La implementación redirige las llamadas a las clases genéricas de Topology Plugin a las clases equivalente del complemento en uso.

Características de Chord:

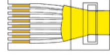
- El encaminamiento es sencillo: se toma como siguiente salto el nodo de la tabla de rutas más cercano al destino, sin pasarse. Si no existe ninguno, se escogerá el nodo inmediatamente posterior.

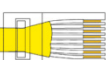
- Un *peer* será responsable de los recursos cuyo ID sea superior al ID del nodo anterior e inferior o igual al propio. Por lo que es inmediato de identificar si un Resource-ID pertenece a un Node-ID dado, siempre que se tenga información sobre el Node-ID previo.

- Cuando un nodo recibe una petición para almacenar de un recurso del que es responsable, deberá almacenar dos réplicas en sus dos primeros sucesores, para asegurarnos redundancia en la red.

RELOAD utiliza algoritmos modificados y personalizados, por lo que no es posible conocer la forma en la que estos se deberían implementar, aun conociendo sus diseños originales. Es previsible que futuros documentos recojan propuestas para la adaptación de nuevas redes *peer-to-peer* a este protocolo.

A continuación tenemos un diagrama UML que recoge las relaciones de este módulo con otros. Hay que especificar que en dicho diagrama solo se recogen las relaciones de la interfaz genérica TopologyPluginInterface con el resto de clases. No se muestran las relaciones que el algoritmo de la red en uso (ChordInterface o cualquier otro) podría tener y que, naturalmente, involucraría más relaciones de las aquí expuestas.





ReloadException
reload.Common.Exception

RoutingTable
reload.Topology

ChordRoutingTable
reload.Topology.Plugins.Chord

ProbeAns
reload.Topology.Plugins

ProbeInformationData
reload.Topology.Plugins

Id
reload.Common

ResourceId
reload.Common

ProbeInformation
reload.Topology.Plugins

NodeId
reload.Common

TopologyPluginInterface
reload.Topology

ChordInterface
reload.Topology.Plugins

ConnectionTable
reload.Topology

ChordThread
reload.Topology.Plugins

ProbeInformationType
reload.Topology.Plugins

IpAddressPort
reload.Common

ProbeReq
reload.Topology.Plugins

Module
reload.Common

TopologyThread
reload.Topology

Message
reload.Message

ChordConnectionTable
reload.Topology.Plugins.Chord

FIGURA 23: Diagrama UML de Topology Plugin

4.8 CAPA DE APLICACIÓN: USO

Por encima de todos los módulos de RELOAD, se ha creado una interfaz general: Reload Interface. Esta interfaz enmascara todos los niveles inferiores, por lo que esta clase es la única manera que tienen los usos de comunicarse con el resto de RELOAD.

Este proyecto no se marca como objetivo la implementación de ningún uso en particular, por lo que los programadores que utilicen este código deberán ser capaces de acoplar cualquier tipo de uso que se defina para RELOAD sobre la mencionada interfaz.

Sin embargo, y por necesidades de las pruebas, se ha desarrollado una versión básica de SIP basada en el borrador de internet del IETF [SIP13]. El fin de este uso es obtener dónde se encuentra la persona a la que se desea telefonear. Los *peers* almacenan en la red su localización (su Node-ID) en el Resource-ID que corresponde a su AOR. Cualquier otro nodo que lo desee, si conoce su AOR, será capaz de obtener el Node-ID y de establacer una conexión directa con él mediante un App Attach.

Así pues, en esta capa tenemos las siguientes clases:

- ReloadInterface: inicializa los módulos del nivel inferior y tiene una pequeña consola que permite teclear los comandos. Sus métodos son genéricos para cualquier uso: *store*, *fetch*, *remove*, *app_attach*…

- SipUsage: convierte los AOR en Resource-ID y realiza llamadas a los métodos de ReloadInterface para almacenar y obtener información, para lo cual utiliza sus estructuras propias. Sus métodos son similares a los de la interfaz: *store*, *fetch*, *remove*, *sip_attach* (llama a *app_attach*)…

- Main: es la única clase ejecutable de la aplicación. Llama a SipUsage.

Main
reload
{main}

DictionaryKey
reload.Common

DestinationData
reload.Common

DestinationType
reload.Common

SipRegistration
reload.SIP

Destination
reload.Common

SipRegistrationData
reload.SIP

SipUsage
reload

SipRegistrationType
reload.SIP

NodeId
reload.Common

Opaque
reload.Common

DataStructure
reload.Common

ResourceId
reload.Common

DataModel
reload.Common

KindId
reload.Common

MessageTransport
reload.Message

ReloadInterface
reload

StorageInterface
reload.Storage

ReloadException
reload.Common.Exception

Module
reload.Common

KindModelList
reload.Storage

DictionaryTable
reload.Storage

FIGURA 24: Diagrama UML de uso: SIP

Se puede observar en el diagrama cómo SipUsage únicamente accede a determinadas clases comunes (paquete Common) y a las propias estructuras que se definen en la documentación de su uso (paquete SIP).

El método *main()* puede estar dentro de la clase SipUsage o ser una clase aparte. En este segundo caso, solamente se relaciona con SipUsage.

Por su parte, ReloadInterface tiene acceso a Common y a sus dos niveles inferiores: los paquetes Storage y Message.

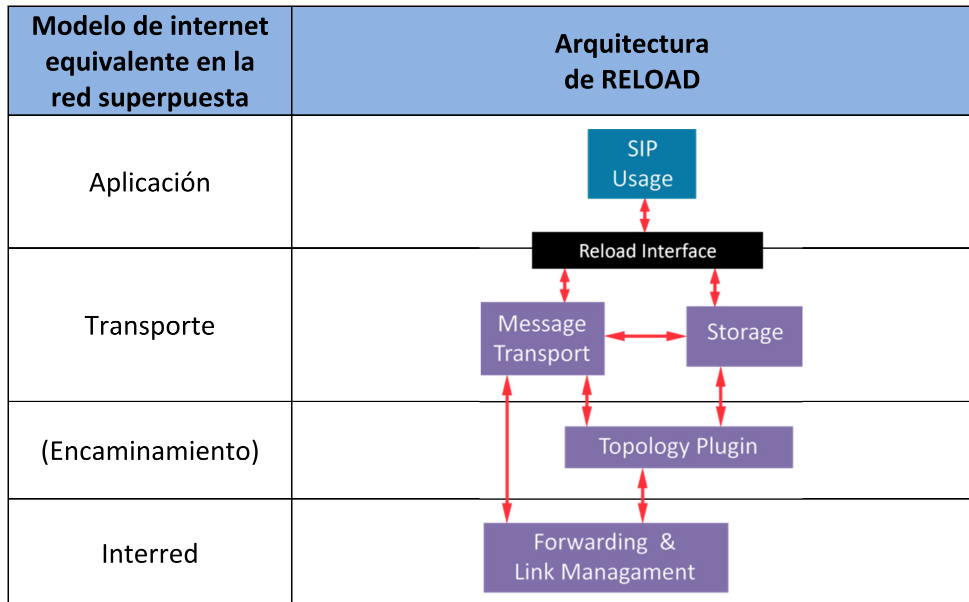


FIGURA 25: Arquitectura de RELOAD implementada

En esta figura se establece cómo se ha adaptado la estructura de RELOAD en esta implementación. Si se desea, puede compararse con la arquitectura original de la figura 3. El uso solo tiene relación con Reload Interface y, a su vez, la interfaz se comunica con Storage y Message Transport. Con esto se amplía y se detalla la arquitectura original de RELOAD sin que exista modificación en su esencia, pues dicha arquitectura no definía las relaciones entre el uso y los niveles inferiores.

4.9 CLASES COMUNES

Existen una serie de clases comunes a todos los niveles de RELOAD.

4.9.1. Excepciones

Se han creado una serie de excepciones que ayuden a depurar el código. ReloadException es la clase principal, que hereda directamente de Exception. El resto de clases heredan de ReloadException y son:

- NotTrueNorFalseReloadException.
- UnimplementedReloadException.
- WrongLengthReloadException.
- WrongPacketReloadException.
- WrongTypeReloadException.

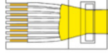
4.9.2. Errores

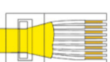
Se han codificado los errores definidos por la especificación, puede encontrarse la tabla completa de errores en el apartado 14.9 del borrador de internet. ReloadError es un error genérico que hereda de Exception, mientras que todos los demás errores heredan de ReloadError.

4.9.3. Tareas

Las tareas son las funciones que la aplicación realiza como "cliente", es decir, envía peticiones (Request) para recibir respuestas (Answer). Estas peticiones pueden ser requeridas por el uso o por la propia aplicación. Con el objeto de aumentar el número de peticiones simultáneas que pueden realizarse, se han creado tres hilos: ForwardingThread, TopologyThread y StorageThread.

Cada uno de los hilos está formado internamente por una cola en la que se almacenan las tareas pendientes que ese módulo tiene. Cada tarea realiza una





o más peticiones, de las que espera una determinada respuesta para actuar en consecuencia. El tiempo que pasa entre una petición y su respuesta el hilo queda bloqueado. Cuando todas las peticiones de una tarea han quedado respondidas, dicha tarea se desencola, dando paso a la siguiente.

4.9.4. Otros

- XML: se encarga del archivo de configuración: “overlay.xml”.

- ID: los identificadores del protocolo son KindId, ResourceId y NodeId. Id es una clase de la que heredan NodeId y ResourceId, y que tiene el contenido del identificador –una cadena de bytes– además de métodos que permiten hacer operaciones matemáticas entre dos identificadores. KindArray permite gestionar una serie de KindId concatenados.

- Direcciones: determinados campos como las direcciones IP y los puertos, y el tipo de dirección –IPv4 o IPv6– se han codificado conforme al borrador. Están incluidas en este módulo común para que puedan ser accedidas desde cualquier nivel.

- Clases auxiliares: Utils, Opaque y Algorithm realizan operaciones de uso frecuente con *arrays* y estructuras. Para una mayor eficiencia computacional se ha preferido codificar unas nuevas en lugar de usar aquellas predefinidas por Java.

- Códigos: MessageCode y ErrorCode son enumeraciones que asignan una cifra a cada tipo de mensaje (según aparece en la figura 10) y a cada tipo de error definido en RELOAD (según se especifica en punto 4.9.2). Estas asignaciones vienen recogidas en el borrador.

- Module: es la clase que permite las llamadas de los módulos entre sí. Module tiene como atributos públicos y estáticos a las cuatro interfaces principales: MT, SI, TPI y FALMI.

4.10 FUNCIONAMIENTO GENERAL

La figura que sigue es un resumen del funcionamiento general de la implementación. Este diagrama es un esquema muy simplificado en el que se representa cómo se comunican los diferentes módulos entre sí.

Main
{main}

SipUsage
-ReloadApi

ReloadInterface

Message Transport

MessageTransport

StorageInterface
+KindModel
+KindCounter
+ResourceReplica
+ResourceKind
+SingleValueTable
+ArrayTable
+DictionaryTable

Message

ForwardingAndLinkInterface
+SendData
-LinkApi [] (client)
-LinkApi (server)

TopologyPluginInterface
+ConnectionTable
+RoutingTable
-ChordApi

ChordInterface

1..*

LinkInterface
-TCPClient
-TCPServer

Topology Plugin

FIGURA 26: Diagrama UML simplificado que muestra una visión general

4.11 COMENTARIOS ACERCA DEL DESARROLLO

Se ha considerado relevante referirse a los aspectos que han supuesto una mayor dificultad en el desarrollo del programa, así como qué decisiones fueron tomadas con el objetivo de atajarlos.

4.11.1. Dificultad en la comprensión

El aspecto que ha sido más complicado de entender es la parte de almacenamiento. El borrador no deja suficientemente claro qué parámetros deben almacenarse. La información que se obtiene es únicamente a través de las estructuras que son enviadas en los mensajes de almacenamiento.

4.11.2. Dificultad en el diseño

En el diseño de la aplicación, decidir el número de hilos y las funciones que debía tener cada uno de ellos ha supuesto un reto. Debido a la inexperiencia previa que se tenía con redes *peer-to-peer*, no resultó fácil decidir el número exacto, de manera que existiera un compromiso entre el funcionamiento en paralelo de los módulos y que el acceso simultáneo a los recursos no supusiera un problema.

Finalmente, la solución fue separar los hilos en función de su rol, dependiendo de si actuaban realizando funciones de servidor o de cliente. Mientras tenemos un hilo “servidor” por cada conexión establecida, solamente se necesitan tres hilos “cliente”: uno por cada tarea (ver 4.9.3). Uno de los hilos gestiona las tareas de almacenamiento (Storage), otro las de Forwarding & Link Management y un tercero lleva las tareas del complemento de topología (Topology Plugin). La elección de estos tres módulos se debe a que cada uno de ellos tiene asociado una parte de mensajes de RELOAD, como se puede observar en la figura 10. La funcionalidad de los hilos será, por tanto, enviar las peticiones de los mensajes que son responsables para recibir respuestas.

4.11.3. Dificultad en la implementación

El aspecto de RELOAD que ha supuesto mayor dificultad a la hora de implementar ha sido el protocolo de Chord. Ha resultado complicado identificar de forma inequívoca cuándo un nodo es un sucesor y cuándo es un predecesor, especialmente en redes con pocos nodos.

Para solventar este problema se han desarrollado dos algoritmos basados en distancias. Obtener la distancia entre dos nodos nos permite tomar decisiones acerca de su posición en la red. Debido a la topología circular de la red a nivel lógico, un nodo puede ser indistintamente sucesor y predecesor.

El primero de los algoritmos está basado en distancias absolutas. Supongamos que tenemos un nodo con Node-ID A y otro con B, la distancia se calcula restando A y B y aplicando sobre esta resta un valor absoluto, por lo que: dist(A, B) = dist(B, A). Para identificar a los vecinos, se divide el anillo en dos, en el punto donde se encuentra el propio nodo que realiza el cálculo.

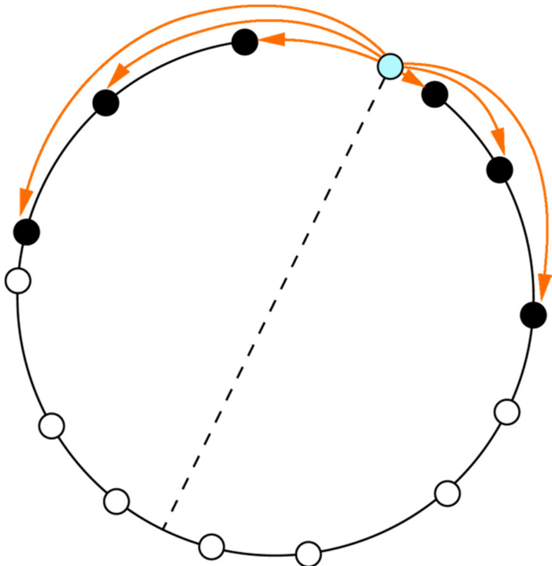


FIGURA 27: Red Chord en la que se buscan vecinos con distancias absolutas

De entre los candidatos, se localizan los nodos más cercanos (menor distancia absoluta) y se los etiqueta como sucesores o predecesores en función de la parte del anillo en la que estén.

El segundo algoritmo se basa en distancias positivas, se calcula la distancia de cada nodo siempre en sentido de las agujas del reloj. Por lo que, salvo que ambos nodos estén separados exactamente 180º, dist(A, B) ≠ dist(B, A).

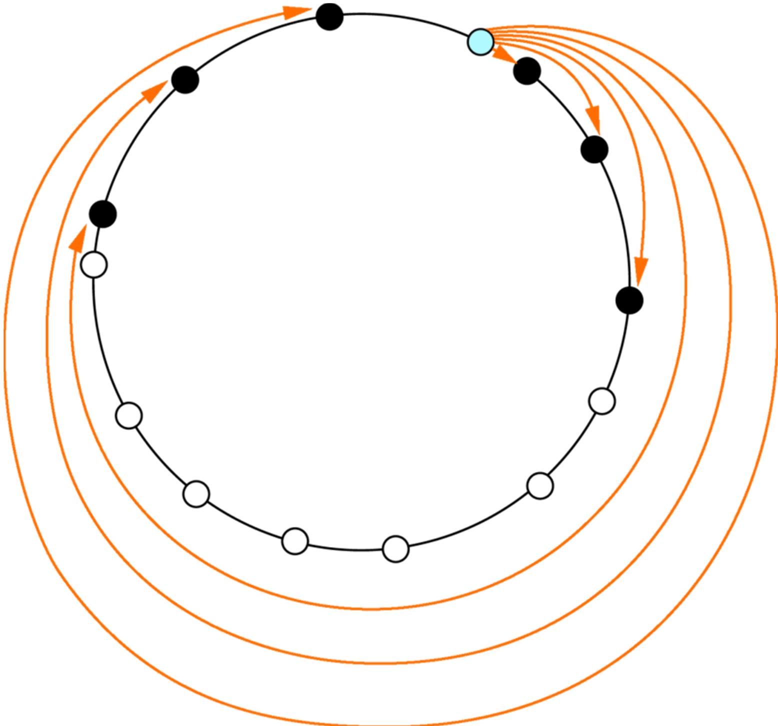
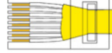
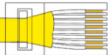


FIGURA 28: Red Chord en la que se buscan vecinos con distancias positivas

Si se utiliza este algoritmo para ordenar todos los nodos sobre los que tenemos conocimiento, se debe calcular la distancia positiva entre nuestro nodo y cada uno de los candidatos. Los tres con distancia menor serán los sucesores y los tres con mayor distancia serán los predecesores.





Capítulo

Resultados

5.1 ESCENARIO PARA LAS PRUEBAS

A continuación, se van a documentar diferentes pruebas, funcionando en una red de doce nodos. Se ha escogido esta cantidad superior a diez nodos porque se ha considerado que es un caso no favorable, donde el número de *peers* es suficiente para rellenar la tabla de vecinos. Tendremos típicamente uno o dos *fingers* y habrá nodos que no estarán conectados entre sí, por lo que los mensajes deberán ser encaminados por la red. Aunque hubiera podido ser deseable, un mayor número de nodos era inviable, debido al gran número de máquinas virtuales requeridas.

En la primera prueba todos los nodos están en una misma red local. Se probarán diferentes aspectos de la implementación, como la inicialización de los nodos –cuando acceden a la red por primera vez y se registran en ella–, el almacenamiento o la forma en la que intercambian la información de encaminamiento –mediante mensajes Update–. Para demostrar el funcionamiento, se han realizado capturas de Wireshark.

Una segunda prueba consistirá en comprobar el funcionamiento de la implementación en la internet real. Para ello, se han establecido algunos ordenadores en diferentes lugares geográficos y, aunque estos pertenecen a redes con direccionamiento privado, se han mapeado los puertos necesarios que requiere la aplicación. En esta segunda prueba, se muestra el registro de tres nodos diferentes en la red, desde el punto de vista del nodo *bootstrap*.

En la lectura del TFG, el próximo día 4 de julio de 2013, se realizarán pruebas con una aplicación real. Se ha modificado la aplicación TextClient de Emmanuel Proulx [PROU07], que utiliza la biblioteca Jain-SIP [JSIP], para crear un cliente SIP de funcionalidad mínima que intercambia mensajes de texto entre dos terminales. Se ha conseguido adaptar dicho cliente a RELOAD, por lo que el programa desarrollado ahora funciona como un auténtico software P2P-SIP.

5.2 PRUEBAS EN RED LOCAL

NOTA: A pesar de que todos los nodos que forman parte de la red se identifican únicamente por su Resource-ID / Node-ID, se mencionan sus direcciones IP para facilitar la comprensión de las figuras que acompañan al texto.

5.2.1. Registro en la red

Cuando un nodo accede a la red –Joining Node (JN), con IP 192.168.1.15–, se conecta a un nodo *bootstrap*, del que previamente conoce su dirección IP, en este caso: 192.168.1.70. Es el quinto nodo en iniciar, por lo que contactará con otros 4 *peers*, a los que añade en la tabla de vecinos.

Time	Source	Destination	Protocol	Length	Info
0.022979	192.168.1.15	192.168.1.70	RELOAD	1269	Attach Request
0.158876	192.168.1.70	192.168.1.15	RELOAD	1304	Attach Response
0.199647	192.168.1.13	192.168.1.15	RELOAD	1304	Update Request
0.255607	192.168.1.15	192.168.1.13	RELOAD	1246	Update Response
0.399271	192.168.1.15	192.168.1.13	RELOAD	1268	Attach Request
0.420789	192.168.1.13	192.168.1.15	RELOAD	1286	Attach Response
0.443881	192.168.1.15	192.168.1.13	RELOAD	1268	Attach Request
0.462110	192.168.1.13	192.168.1.15	RELOAD	1286	Attach Response
0.478593	192.168.1.15	192.168.1.14	RELOAD	1268	Attach Request
0.562027	192.168.1.14	192.168.1.15	RELOAD	1286	Attach Response
0.625787	192.168.1.15	192.168.1.13	RELOAD	1261	Join Request
0.643610	192.168.1.13	192.168.1.15	RELOAD	1245	Join Response
0.772796	192.168.1.13	192.168.1.15	RELOAD	1453	Store Request
0.786422	192.168.1.15	192.168.1.13	RELOAD	1261	Store Response
0.791981	192.168.1.13	192.168.1.15	RELOAD	1453	Store Request
0.797329	192.168.1.15	192.168.1.13	RELOAD	1261	Store Response
0.811386	192.168.1.13	192.168.1.15	RELOAD	1320	Update Request
0.844412	192.168.1.15	192.168.1.13	RELOAD	1246	Update Response
1.005322	192.168.1.15	192.168.1.13	RELOAD	1250	Update Request
1.011821	192.168.1.13	192.168.1.15	RELOAD	1246	Update Response
1.030923	192.168.1.15	192.168.1.14	RELOAD	1250	Update Request
1.040872	192.168.1.14	192.168.1.15	RELOAD	1246	Update Response
1.045741	192.168.1.15	192.168.1.12	RELOAD	1250	Update Request
1.053522	192.168.1.12	192.168.1.15	RELOAD	1246	Update Response
1.063275	192.168.1.15	192.168.1.70	RELOAD	1250	Update Request
1.112422	192.168.1.70	192.168.1.15	RELOAD	1250	Update Request
1.118390	192.168.1.15	192.168.1.70	RELOAD	1246	Update Response
1.123321	192.168.1.70	192.168.1.15	RELOAD	1246	Update Response
1.140494	192.168.1.15	192.168.1.13	RELOAD	1454	Store Request
1.166302	192.168.1.13	192.168.1.15	RELOAD	1311	Store Response

FIGURA 29: Mensajes de inicialización de un nodo

Tras esto, envía un mensaje de Attach al Resource-ID correspondiente –su propio Node-ID más uno–, el cual será encaminado a través del *bootstrap* hasta su Admitting Peer (AP), cuya dirección IP es la 192.168.1.13.

Tras recibir el Attach, se forma una conexión directa entre el Joining Node y el Admitting Peer. Se envían mensajes directos de Update entre estos dos nodos, gracias a esto el JN recibe información de los vecinos del AP, que también serán los suyos. A continuación, realiza Attach a los otros 3 nodos que existen en la red (dos de los cuales son encaminados a través del AP) y, como ya está preparado para formar parte de la red, envía un Join a su Admitting Peer.

Tras esto, el JN envía tres mensajes de Store en los cuales envía la información de tres Resource-ID de los que ahora será responsable el AP. Posteriormente, el JN envía al AP un Update que le indica que ya es un nodo más de la red.

Ahora, el AP envía Updates a toda su tabla de encaminamiento para indicarles que ya está listo para encaminar para ellos. Por último, SIP registra su AOR en la red mediante un Store (que el responsable de almacenar dicha información sea el AP es pura coincidencia y podría ser cualquier otro *peer* de la red).

5.2.2. Obtención y eliminación de datos

Con todos los nodos ya iniciados, se ha guardado en la red un mapeo de un AOR en otro AOR, para lo cual se envía un mensaje Store y se recibe la correspondiente respuesta (primeros dos paquetes de la figura).

Time	Source	Destination	Protocol	Length	Info
8.7396110	192.168.1.15	192.168.1.2	RELOAD	1350	Store Request
8.7410840	192.168.1.2	192.168.1.15	RELOAD	1261	Store Response
10.674067	192.168.1.15	192.168.1.13	RELOAD	1279	Fetch Request
10.691594	192.168.1.13	192.168.1.15	RELOAD	1471	Fetch Response
15.695138	192.168.1.15	192.168.1.16	RELOAD	1279	Fetch Request
15.706263	192.168.1.16	192.168.1.15	RELOAD	1453	Fetch Response
23.638374	192.168.1.15	192.168.1.13	RELOAD	1330	Store Request
23.655468	192.168.1.13	192.168.1.15	RELOAD	1297	Store Response

FIGURA 30: Almacenamiento y obtención de información

A continuación, mediante la consola hemos realizado un par de consultas de diferentes AOR. Se pueden observar las peticiones Fetch y las respuestas, que nos facilita, en este caso, el Node-ID del AOR deseado. La respuesta de este Fetch se muestra en pantalla por consola.

Por último, hemos realizado la eliminación de nuestro AOR que, como se ve en los dos últimos paquetes, se realiza con un Store que machaca la información almacenada, pues no existe un mensaje específico para eliminaciones.

5.2.3. Réplicas

Siendo ya parte de la red, un *peer* (cuya dirección IP 192.168.1.19) desea almacenar su información en un Resource-ID que nos pertenece.

Time	Source	Destination	Protocol	Length	Info
1.016677	192.168.1.1	192.168.1.15	RELOAD	1490	Store Request
1.024823	192.168.1.15	192.168.1.1	RELOAD	1329	Store Response
1.025991	192.168.1.15	192.168.1.2	RELOAD	1453	Store Request
1.029554	192.168.1.2	192.168.1.15	RELOAD	1261	Store Response
1.034978	192.168.1.15	192.168.1.16	RELOAD	1453	Store Request
1.043227	192.168.1.16	192.168.1.15	RELOAD	1261	Store Response

FIGURA 31: Almacenamiento de réplicas

En primer lugar, recibimos un Store, pero no procede del nodo esperado. Esto es así porque al estar en una red con bastantes nodos, se da la situación de que para el nodo que genera el paquete no somos su vecino ni su *finger*, de modo que el paquete se encamina por la red, en este caso, con un salto intermedio, pasando por el peer con dirección IP 192.168.1.1. Como hay encaminamiento simétrico recursivo, la respuesta se devuelve al mismo nodo que nos lo entregó.

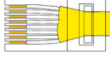
Debemos realizar dos réplicas de esta información, por lo que se la pasamos a nuestros dos primeros sucesores. Nuestro sucesor inmediato es aquel con IP 192.168.1.2, de modo que se lo entregamos directamente. El siguiente sucesor es el 192.168.1.16, por lo que el proceso de entrega es similar.

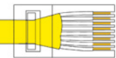
5.3 MENSAJES

5.3.1. Tráfico Join

```
⊞ Frame 39: 1261 bytes on wire (10088 bits), 1261 bytes capt ace 0
⊞ Ethernet II, Src: Microsof_1d:3f:17 (00:03:ff:1d:3f:17), 03:ff:1b:3f:17)
⊞ Internet Protocol Version 4, Src: 192.168.1.15 (192.168 92.168.1.13)
⊞ Transmission Control Protocol, Src Port: 1072, Dst Por 5, Len: 1207
⊟ REsource LOcation And Discovery
  ⊟ ForwardingHeader: Join Request
      relo_token (uint32): 0xd2454c4f
      overlay (uint32): 0x9aa32b8d
      configuration_sequence (uint16): 22
      version (uint8): Unknown (0x0a)
      ttl (uint8): 30
    ⊞ fragment (uint32): 0xc0000000 (Fragment) (Last)
      length (uint32): 1207
      transaction_id (uint32): 0x6263860067e15292
      max_response_length (uint32): 0
        [Response length not restricted]
      via_list_length (uint16): 0
      destination_list_length (uint16): 18
      options_length (uint16): 0
    ⊟ destination_list (Destination<18>): 1 elements
      ⊟ Destination: node
          type (DestinationType): node (0x01)
          length (uint8): 16
          node_id (NodeId): 5454063d9deb301470b24ef2b607a00d
  ⊟ MessageContents
      message_code (uint16): 15 (join_req)
    ⊟ message_body (JoinReq<16>)
        length (uint32): 16
      ⊟ JoinReq
          joining_peer_id (NodeId): d7ae7ea9bf56ebdc60d222a046176d05
        ⊟ overlay_specific_data (opaque<0>)
            length (uint16): 0
    ⊞ extensions (0 elements)
  ⊟ SecurityBlock
    ⊟ certificates (GenericCertificate<1098>): 2 elements
        length (uint16): 1098
      ⊞ GenericCertificate
      ⊞ GenericCertificate
    ⊟ signature (Signature)
      ⊟ algorithm (SignatureAndHashAlgorithm)
          hash (HashAlgorithm): SHA1 (2)
          signature (SignatureAlgorithm): RSA (1)
      ⊟ identity (SignerIdentity)
          identity_type (SignerIdentityType): cert_hash (1)
          length (uint16): 18
        ⊞ identity (SignerIdentityValue[18])
      ⊞ signature_value (opaque<0>)
```

FIGURA 32: Detalle de un mensaje Join Request





En esta captura se analiza el paquete Join Request del apartado 5.2.1: *Registro en la red*. Se trata del undécimo paquete de la figura 29.

Los dos nodos que se intercambian este mensaje son:

- el nodo origen, con ID 0xd7ae7ea9bf56ebdc60d222a046176d05 y dirección IP 192.168.1.15,
- y el destino, con Node-ID 0x5454063d9deb301470b24ef2b607a00d y dirección IP 192.168.1.13,

no existiendo nodos intermedios al tratase de un mensaje Join.

En Forwarding Header aparecen los valores anteriormente comentados. La versión de RELOAD es $0x0A_{16} = 10_{10}$ = **1.0**, el mensaje se dirige a un Node-ID (el Admitting Peer), el código del mensaje es el 15 (Join Request) y en su contenido consta únicamente del Node-ID del Joining Node.

5.3.2. Tráfico Fetch

A continuación se muestra el primero de los paquetes Fetch Request de la figura 30, en el apartado 5.2.2 de la sección anterior: *Obtención y eliminación de datos*.

Se puede observar cómo en el campo destination_list, el destino es un Resource-ID, por lo que será el Node-ID responsable de ese recurso quien recibirá el mensaje. El código de mensaje es el número 33 (Join Request) y en el interior del cuerpo del mensaje se advierte de nuevo el Resource-ID, siendo una petición simple que cuenta únicamente con un Kind-ID (el número 1: "SIP-REGISTRATION"). Un generation_counter de 0 indica que se admite cualquier valor, por reciente o antiguo que sea. Por último, no se especifican claves de diccionario (0 keys), puesto que el uso SIP detalla que las peticiones se deben hacer vacías, de manera que en la respuesta se devolverán todos los valores almacenados en ese recurso.

```
⊟ REsource LOcation And Discovery
  ⊟ ForwardingHeader: Fetch Request
      relo_token (uint32): 0xd2454c4f
      overlay (uint32): 0x9aa32b8d
      configuration_sequence (uint16): 22
      version (uint8): Unknown (0x0a)
      ttl (uint8): 30
    ⊞ fragment (uint32): 0xc0000000 (Fragment) (Last)
      length (uint32): 1225
      transaction_id (uint32): 0xca8d233f5d4d762c
      max_response_length (uint32): 0
        [Response length not restricted]
      via_list_length (uint16): 0
      destination_list_length (uint16): 19
      options_length (uint16): 0
    ⊟ destination_list (Destination<19>): 1 elements
      ⊟ Destination: resource
          type (DestinationType): resource (0x02)
          length (uint8): 17
        ⊟ resource_id (ResourceId<16>)
            length (uint8): 16
            data (bytes): 5ca28cc8a9fed53e91d0604016ef8229
  ⊟ MessageContents
      message_code (uint16): 9 (fetch_req)
    ⊟ message_body (FetchReq<33>)
        length (uint32): 33
      ⊟ FetchReq
        ⊟ resource (ResourceId<16>)
            length (uint8): 16
            data (bytes): 5ca28cc8a9fed53e91d0604016ef8229
        ⊟ specifiers(StoredDataSpecifier<14>): 1 elements
            length (uint16): 14
          ⊟ StoredDataSpecifier
              kind (KindId): 1 (SIP-REGISTRATION)
              generation_counter (uint64): 0
              length (uint16): 0
            indices(0 keys)
    ⊞ extensions (0 elements)
  ⊟ SecurityBlock
    ⊟ certificates (GenericCertificate<1098>): 2 elements
        length (uint16): 1098
      ⊞ GenericCertificate
      ⊞ GenericCertificate
    ⊟ signature (Signature)
      ⊞ algorithm (SignatureAndHashAlgorithm)
      ⊞ identity (SignerIdentity)
      ⊞ signature_value (opaque<0>)
```

FIGURA 33: Detalle de un mensaje Fetch Request

5.3.3. Tráfico Store

Por último, se analizará un mensaje Store Request del apartado 5.2.3: *Réplicas*.

Se trata del tercer paquete de la figura 31, en el cual se realiza una petición para almacenar una réplica.

```
⊟ REsource LOcation And Discovery
  ⊞ ForwardingHeader: Store Request
  ⊟ MessageContents
      message_code (uint16): 7 (store_req)
    ⊟ message_body (StoreReq<208>)
        length (uint32): 208
      ⊟ StoreReq
        ⊞ resource (ResourceId<16>)
          replica_number (uint8): 1
        ⊟ kind_data (StoreKindData<186>): 1 elements
            length (uint32): 186
          ⊟ StoreKindData
              kind (KindId): 1 (SIP-REGISTRATION)
              generation_counter (uint64): 1
            ⊟ values (StoredData<170>): 1 elements
                length (uint32): 170
              ⊟ StoredData
                  length (uint32): 166
                  storage_time (uint64): Jan  1, 1970 00:00:00.000000000 UTC
                  lifetime (uint32): 0
                ⊟ value (DictionaryEntry)
                  ⊟ key (DictionaryKey)
                      length (uint16): 16
                      NodeId: 3aeaf64a17c82131ea98dc1e01ab8946
                  ⊟ value (DataValue) (DataValue)
                      exists (Boolean): True
                      length (uint32): 124
                    ⊟ SipRegistration
                        type (SipRegistrationType):sip_registration_route(2)
                        length (uint16): 121
                      ⊟ data (SipRegistrationData)
                        ⊟ contact_prefs (opaque<99>)
                            length (uint16): 99
                            data (string): (& (sip.audio=TRUE)\n(sip//r
                                           =msg-taker)\n(sip.automa//xed)\n
                        ⊟ destination_list (Destination<18>): 1 elements
                            length (uint16): 18
                          ⊟ Destination: node
                              type (DestinationType): node (0x01)
                              length (uint8): 16
                          node_id (NodeId): 3aeaf64a17c82131ea98dc1e01ab8946
              ⊟ signature (Signature)
                ⊞ algorithm (SignatureAndHashAlgorithm)
                ⊞ identity (SignerIdentity)
                ⊟ signature_value (opaque<0>)
                    length (uint16): 0
    ⊞ extensions (0 elements)
  ⊞ SecurityBlock
```

FIGURA 34: Detalle de un mensaje Store Request

Aunque no aparece por cuestiones de espacio, el destino de este mensaje es un nodo, el Node-ID en el que deseamos almacenar la réplica. El código del mensaje es 7 (Store Request) y en el contenido se incluye el Resource-ID donde está almacenado el contenido original (octava línea de la figura), recurso del que, naturalmente, el nodo receptor de este mensaje no será responsable. También puede observarse que el campo replica_number es 1. En las peticiones de almacenamiento, un valor de 0 del número de réplica indica que se trata de un mensaje original (no réplica); en este caso el número 1 que indica que es una réplica en el primer sucesor.

Puede comprobarse que se trata de un Kind-ID "SIP-REGISTRATION", dado que se está utilizando SIP y este uso considera un único identificador de Kind. El generation_counter es 1, este campo indica aproximadamente el número de veces que este valor ha cambiado. En este caso, el valor ha sido escrito una sola vez en el nodo responsable.

En StoredData tenemos el contenido que se desea almacenar. El campo storage_time no está relleno, puesto que la temporización no está codificada, por lo que al enviar el campo a cero, identifica la fecha del 1 de enero de 1970 (el tiempo de almacenamiento es el número de segundos a partir de esa fecha).

Vemos que será almacenada una única entrada del tipo diccionario, con una clave y un valor. El uso de SIP define que la clave será el identificador de nodo donde se encuentra la persona que podrá ser llamada (en este caso, el Node-ID: 0x3aeaf64a17c82131ea98dc1e01ab8946), mientras que el valor será una estructura con datos adicionales de SIP, en la que se especifica información del protocolo como los métodos que soporta, si admite vídeo, movilidad…, y el esquema en uso (SIP o SIPS).

Como ya se comentó en el apartado 2.4 (Mensajes en RELOAD), en el campo Security Block se envían dos certificados obtenidos del fichero de configuración y una firma vacía, al no estar aún implementada la seguridad.

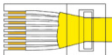
5.4 PRUEBAS EN INTERNET

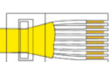
En esta prueba, todos los nodos están en diferentes redes locales, la mayoría detrás de un *router* con NAT. Se pretende comprobar cómo el funcionamiento es correcto cuando los diferentes nodos están distribuidos por internet.

No.	Time	Source	Destination	Protocol	Length	Info
219	7.1903720	2.137.174.160	192.168.1.1	RELOAD	1269	Attach Request
1155	7.5107330	192.168.1.1	2.137.174.160	RELOAD	1268	Attach Response
1220	7.6438240	192.168.1.1	2.137.174.160	RELOAD	1256	Update Request
1227	7.7867770	2.137.174.160	192.168.1.1	RELOAD	1246	Update Response
1353	8.0621200	2.137.174.160	192.168.1.1	RELOAD	1261	Join Request
1592	8.1542760	192.168.1.1	2.137.174.160	RELOAD	1245	Join Response
1627	8.3973760	192.168.1.1	2.137.174.160	RELOAD	1272	Update Request
1629	8.5266930	2.137.174.160	192.168.1.1	RELOAD	1246	Update Response
4661	80.095194	87.221.218.195	192.168.1.1	RELOAD	1269	Attach Request
4710	80.111015	192.168.1.1	2.137.174.160	RELOAD	1287	Attach Request
4712	80.215722	2.137.174.160	192.168.1.1	RELOAD	1286	Attach Response
4823	80.258199	192.168.1.1	87.221.218.195	RELOAD	1286	Attach Response
4844	81.119234	2.137.174.160	192.168.1.1	RELOAD	1286	Attach Request
5010	81.182719	192.168.1.1	2.137.174.160	RELOAD	1286	Attach Response
5041	81.496829	87.221.218.195	192.168.1.1	RELOAD	1250	Update Request
5177	81.544866	192.168.1.1	87.221.218.195	RELOAD	1246	Update Response
10936	142.205086	88.6.233.110	192.168.1.1	RELOAD	1269	Attach Request
11063	142.247858	192.168.1.1	88.6.233.110	RELOAD	1268	Attach Response
11102	142.304610	192.168.1.1	88.6.233.110	RELOAD	1288	Update Request
11108	142.352769	88.6.233.110	192.168.1.1	RELOAD	1246	Update Response
11191	142.598939	88.6.233.110	192.168.1.1	RELOAD	1268	Attach Request
11248	142.620188	192.168.1.1	2.137.174.160	RELOAD	1286	Attach Request
11264	142.755350	2.137.174.160	192.168.1.1	RELOAD	1286	Attach Response
11378	142.795462	192.168.1.1	88.6.233.110	RELOAD	1286	Attach Response
11393	142.904134	88.6.233.110	192.168.1.1	RELOAD	1268	Attach Request
11451	142.925368	192.168.1.1	87.221.218.195	RELOAD	1286	Attach Request
11463	143.019384	87.221.218.195	192.168.1.1	RELOAD	1286	Attach Response
11576	143.057271	192.168.1.1	88.6.233.110	RELOAD	1286	Attach Response
11614	143.357057	88.6.233.110	192.168.1.1	RELOAD	1261	Join Request
11755	143.408290	192.168.1.1	88.6.233.110	RELOAD	1245	Join Response
11793	143.465269	192.168.1.1	88.6.233.110	RELOAD	1304	Update Request
11799	143.514665	88.6.233.110	192.168.1.1	RELOAD	1246	Update Response
14648	171.347317	192.168.1.1	87.221.218.195	RELOAD	1279	Fetch Request
14650	171.730709	87.221.218.195	192.168.1.1	RELOAD	1453	Fetch Response
15195	171.903028	192.168.1.1	88.6.233.110	RELOAD	1269	AppAttach Request
15253	172.590135	88.6.233.110	192.168.1.1	RELOAD	1269	AppAttach Response

FIGURA 35: Mensajes en la internet real (captura desde el nodo *bootstrap*)

RELOAD no puede funcionar detrás de NAT sin el módulo de Transversal NAT. Esto es debido a que los mensajes Attach y App Attach llevan la IP del nodo que generó el mensaje, y los traductores de direcciones de los *routers* solo modifican la dirección del paquete IP, no la del mensaje de RELOAD. Por ello, y con el fin de realizar esta prueba, se ha programado un sencillo módulo de NAT.





Capítulo

Conclusiones

6.1 CONCLUSIONES GENERALES

En esta memoria se analiza una de las primeras implementaciones de RELOAD, el protocolo que se va a convertir en el nuevo estándar para redes *peer-to-peer*. La RFC 6940 implicará una transformación importante de internet, no solo a la hora de permitir la creación de programas que funcionen de forma distribuida. Su principal desafío es la redefinición de protocolos ampliamente aceptados por la industria que, si se llegan a adaptar, supondrán el cambio progresivo del modelo cliente-servidor a un nuevo paradigma que minimizará la necesidad de servidores centralizados.

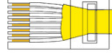
Con este trabajo se ha pretendido demostrar cómo es posible realizar una implementación sencilla extrayendo los aspectos principales del borrador, a la vez que se propone un diseño modular que permitiría, incluso, que dos módulos codificados por diferentes empresas funcionasen conjuntamente.

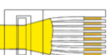
6.1.1. Funcionamiento

En lo que respecta al rendimiento, en una red pequeña de 10 o 12 nodos, el uso de la CPU está como máximo al 2,1% durante el registro en la red. Tras esto, el uso cae a un valor muy cercano al 0%. El consumo de memoria RAM varía entre 23,8 y 25,8 MB, alcanzando un valor máximo en la inicialización.

Estos valores incluyen el consumo de recursos de la máquina virtual de Java. Esta prueba se realizó con un procesador Intel Core i7 950 Quad-core con una velocidad de reloj de 3 GHz, funcionando bajo el sistema operativo Microsoft Windows 7.

Se considera que se han cumplido los objetivos previstos del Trabajo Fin de Grado, al haber conseguido una implementación que ha funcionado incluso por encima de las estimaciones iniciales.





Aunque se trata de una versión *alpha* que, por sus características, no puede garantizar el funcionamiento con otras implementaciones, creemos que puede servir de referencia a otros programadores, por su correcto diseño y su código sencillo. Por ello, la idea es liberar el código después del verano, e ir actualizando periódicamente el programa conforme se vaya completando, hasta que se convierta en una aplicación completamente funcional.

Aún queda mucho trabajo por delante: optimizar el código y corregir errores, pues en una aplicación de semejante extensión, siempre se encuentran aspectos que mejorar.

6.1.2. La implementación en cifras

- Más de medio millón de caracteres (554 kilobytes), sin contar código reutilizado.
- 11.025 líneas de código fuente, sin contar comentarios, y otras 7.609 líneas en banco.
- 163 clases de Java.
- 28 paquetes.
- Coste estimado de 40.000 euros, según el presupuesto anexo.
- Este documento impreso [electrónico en esta versión en PDF] cuenta con 18.007 palabras.

6.1.3. Documentación

La API de las interfaces de los módulos, generada mediante la utilidad Javadoc, se encuentra disponible en:

http://www.relod.net

6.2 CONCLUSIONES PERSONALES

6.2.1. Aportaciones profesionales

Ha sido un trabajo intenso y complejo, que me ha permitido un ligero acercamiento al mundo empresarial, al trabajar en el desarrollo de una aplicación real con las tecnologías actuales. Durante el desarrollo, he contactado con profesionales que estaban trabajando con RELOAD, algunos de ellos colaboradores del IETF, de los que he recibido ayuda y consejos.

Otro aspecto a destacar ha sido poder introducirme en el mundo de las redes P2P. Es de prever que este modelo adquiera una importancia relevante en los próximos tiempos, y tener este tipo de conocimientos es un rasgo diferenciador.

Todo ello me ha supuesto una experiencia enriquecedora, pero también algunos momentos duros en los que el siguiente a paso a dar era incierto. He aprendido mucho de mis propios errores y este proyecto supone un punto de inflexión en mi formación como ingeniero.

6.2.2. Mejoras propuestas

Una de las mejoras que se proponen es establecer de forma clara y unívoca la forma en que el complemento de topología puede comunicarse con el resto de los módulos. El borrador define únicamente Chord como algoritmo DHT, aunque tiene como requisito la obligación de funcionar con cualquier otro tipo de red superpuesta estructurada o desestructurada.

Aunque el programador pueda hacer todo el esfuerzo para que este módulo se comporte como un auténtico complemento (*plugin*), que se pueda intercambiar sin modificar el resto del código, solo se puede garantizar que la implementación está diseñada de una forma correcta cuando se realice un segundo algoritmo y se ponga a funcionar en lugar del primero.

Para evitar esta situación, convendría especificar las llamadas que un módulo externo puede hacer a Topology Plugin. Si bien, algunas como obtener el siguiente salto para encaminar el paquete son evidentes, otras como la forma de manejar y acceder a los vecinos y *fingers* son menos inmediatas, puesto que la topología define el número de nodos y cómo estos se organizan.

En este sentido, es especialmente importante lograr un acuerdo en el módulo de Topology Plugin, pues fijar una API normalizada es una forma de garantizar que nuevos algoritmos puedan funcionar con implementaciones preexistentes.

En otro orden de cosas, el borrador no define una capa de enlace propia en la red superpuesta. Simplemente se limita a indicar que el nivel de Link podrá ser TLS o DTLS, pero luego, incoherentemente, define una cabecera extra en ese mismo nivel: Framing Header. Por si no es suficiente, a la capa inmediatamente superior la denomina Forwarding & Link Management, cometiendo el error de incluir la palabra Link en una capa que ya no es de enlace, sino de interred.

Se propone la creación de un nuevo nivel intermedio entre TLS y Forwarding, llamado: “Link Layer”, y modificar el nombre de “Forwarding & Link Management” por simplemente “Forwarding Management”. Link se encargaría de las tareas propias que le asigna el documento: control de congestión, semifiabilidad y temporización, así como de incluir la mencionada cabecera Framing Header.

Esto no supondría ningún cambio en la práctica, solamente es una aclaración teórica y conceptual que ayuda a entender el funcionamiento de RELOAD.

En el caso de que una nueva implementación se desarrollase con esta estructura de capas propuesta, sería imposible diferenciarla de cualquier otra que utilice la arquitectura anterior. La interoperabilidad estaría garantizada pues, a fin de cuentas, los mensajes que se intercambian los nodos son los mismos, y esta modificación solo afecta al diseño interno de la aplicación.

A continuación se presenta una imagen con la nueva arquitectura propuesta:

Modelo de internet	Modelo de internet equivalente en la red superpuesta	Arquitectura modificada de RELOAD
Aplicación	Aplicación	SIP Usage · ShaRe Usage
	Transporte	Message Transport · Storage
	(Encaminamiento)	Topology Plugin
	Interred	Forwarding Management
	Enlace	Link Layer
Transporte	TLS · DTLS	
Interred		
Enlace		

FIGURA 36: Nueva estructura de capas propuesta para RELOAD

6.2.3. Críticas

En general, la estructura del borrador es complicada, pues no queda claro a qué módulo pertenecen los mensajes, ni realmente cómo es la organización por capas. En ese sentido, otros protocolos como P2PP son más sencillos de entender, e incluso la mayoría de RFC consultadas ha resultado de menor dificultad.

Se va a citar un ejemplo en el cual ocurren una serie de llamadas poco intuitivas entre clases, en un mensaje completamente habitual: un Ping Request. Cuando un nodo recibe un mensaje en la capa de enlace de la red superpuesta, se lo pasa a Forwarding, para que este decida si le pertenece o si debe reenviarlo a otro nodo. Para ello consulta a la capa Message, pues por sí mismo no es capaz de entender la estructura, ya que Forwarding no tiene ningún tipo de cabecera asociada. Tras hacer esta comprobación y verificar que efectivamente es suyo, lo pasa definitivamente a la capa superior, que es de nuevo Message.

Message analiza el paquete, ve que se trata de un Ping Request y obtiene que el responsable (consultar figura 10) es Forwarding, por lo que le pasa el MESSAGE BODY a este módulo, que podrá ver el contenido del mensaje. Este genera la contestación, por lo que se enviará por el camino opuesto: Forwarding → Message → Forwarding (consulta de nuevo a Message) → Link.

Si ya es fácil constatar que no es normal que una capa tan relevante como Forwarding & Link Management no tenga ninguna cabecera asociada (podría perfectamente tener acceso a los datos que le permitieran decidir sobre si ha de reenviar el mensaje o no), lo que no resulta en absoluto entendible es que Forwarding entregue un mensaje a una capa superior, que inmediatamente vuelve a él, generando un extraño tráfico en zigzag.

En este gráfico puede observarse el mensaje Ping Request, así como las capas responsables de las distintas secciones del mensaje.

Capa						
FORWARDING & LINK MANAGMENT					PING REQ	
MESSAGE TRANSPORT				HEADER	BODY	FOOTER
FORWARDING & LINK MANAGMENT				RELOAD MESSAGE		
OVERLAY LINK LAYER			FH	RELOAD MESSAGE		
		TLS HEAD.	ENCRYPTED APPLICATION DATA			
	TCP HEAD.	TCP DATA				
IP HEADER	IP DATA					

FIGURA 37: Desglose del paquete Ping Request

REFERENCIAS

[RELO13] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset and H. Schulzrinne, *REsource LOcation And Discovery (RELOAD) Base Protocol draft-ietf-p2psip-base-26*, February 24, 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-base-26

[SIP13] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset and H. Schulzrinne, *A SIP Usage for RELOAD draft-ietf-p2psip-sip-09*, February 25, 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-sip-09

[P2PSIP] Peer-to-Peer Session Initiation Protocol (p2psip), (undated). [Online]. Viewed 2013 May 13. Available: http://datatracker.ietf.org/wg/p2psip/charter/

[BYK08] John F. Buford, Heather Yu and Eng Keong Lua, *P2P Networking and Applications*. Burlington, Massachusetts: Elsevier, 2008, pp. 29-31.

[VVN12] D. Vivekanandreddy, Allamprabhu Vastrad and R. M. Nareshkumar, “Implementation of a novel optimized trust based search approach for the peer to peer (P2P) platform”, *World Journal of Science and Technology*, vol. 2, no. 10, pp. 129-132, 2012.

[SGH09] Tallat M. Shafaat, Ali Ghodsi and Seif Haridi, “Dealing with network partitions in structured overlay networks”, *Peer-To-Peer Networking and Applications*, vol. 2. no. 4, pp. 334-347, 2009.

[MBCGG09] Isaias Martinez-Yelmo, Alex Bikfalvi, Ruben Cuevas, Carmen Guerrero and Jaime Garcia, “H-P2PSIP: Interconnection of P2PSIP domains for Global Multimedia Services based on a Hierarchical DHT Overlay Network”, *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 53, no. 4, pp. 556-568, 2009.

[Chord] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. August 27, 2001. [Online]. Available: http://pdos.csail.mit.edu/papers/chord:sigcomm01

[wikiChord] Chord (peer-to-peer). (undated). *Wikipedia, The Free Encyclopedia*. [Online]. Viewed 2013 Feb 12. Available: http://en.wikipedia.org/wiki/Chord_(peer-to-peer)

[Kadem] Petar Maymounkov and David Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric", *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pp. 53-65, 2002.

[wikiKadem] Kademlia. (undated). *Wikipedia, The Free Encyclopedia*. [Online]. Viewed 2013 Jun 10. Available: http://en.wikipedia.org/wiki/Kademlia

[P2PP] S. Baset, H. Schulzrinne and M. Matuszewski, *Peer-to-Peer Protocol (P2PP) draft-baset-p2psip-p2pp-01*, November 19, 2007. [Online]. Available: http://tools.ietf.org/html/draft-baset-p2psip-p2pp-01

[DisCo13] A. Knauf, G. Hege and M. Waehlisch, *A RELOAD Usage for Distributed Conference Control (DisCo) draft-ietf-p2psip-disco-00*. October 9, 2012. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-disco-00

[ShaRe13] A. Knauf, T C. Schmidt, G. Hege and M. Waehlisch, *A Usage for Shared Resources in RELOAD (ShaRe) draft-ietf-p2psip-share-01*. February 24, 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-share-01

[CoAP13] J. Jimenez, J. Lopez-Vega, J. Maenpaa and G. Camarillo, *A Constrained Application Protocol (CoAP) Usage for REsource Location And Discovery (RELOAD) draft-jimenez-p2psip-coap-reload-03*. February 18, 2013. [Online]. Available: http://tools.ietf.org/html/draft-jimenez-p2psip-coap-reload-03

[SNMP13] Y. Peng, W. Wang, Z. Hao and Y. Meng, *An SNMP Usage for RELOAD draft-peng-p2psip-snmp-05*. October 18, 2012. [Online]. Available: http://tools.ietf.org/html/draft-peng-p2psip-snmp-05

[MACA13] J. Maenpaa and G. Camarillo, *Service Discovery Usage for REsource LOcation And Discovery (RELOAD) draft-ietf-p2psip-service-discovery-08*. February 23, 2013. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-service-discovery-08

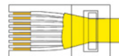
[PRJ12] M. Petit-Huguenin, J. Rosenberg and C. Jennings, *A Usage of Resource Location and Discovery (RELOAD) for Public Switched Telephone Network (PSTN) Verification draft-petithuguenin-vipr-reload-usage-04*. March 12, 2012. [Online]. Available: http://tools.ietf.org/html/draft-petithuguenin-vipr-reload-usage-04

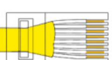
[MMR10] R. Mahy, P. Matthews and J. Rosenberg, *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*, RFC 5766, April 2010.

[PETI12] Marc Petit-Huguenin, [P2PSIP] 2nd RELOAD interoperability testing event, *IETF Discussion Archive*, November 2012. [Online]. Viewed 2013 May 15. Available: http://www.ietf.org/mail-archive/web/p2psip/current/msg06325.html

[PROU07] Emmanuel Proulx, An Introduction to the JAIN SIP API, *Oracle*, October 2007. [Online]. Viewed 2013 May 6. Available: http://www.oracle.com/technetwork/articles/entarch/introduction-jain-sip-090386.html

[JSIP] Jsip, *Java.net*, (undated). [Online]. Viewed 2013 May 6. Available: https://jsip.java.net/





PLANIFICACIÓN

El trabajo se ha desarrollado entre el 1 agosto de 2011 y el 15 de junio de 2013.

	E	F	M	A	M	J	J	A	S	O	N	D
2011								½	1	0	0	0
2012	0	1	1	1	0	½	½	½	1	1	1	1
2013	1	½	½	½	½	1						

Se especifican aquellos meses en los que se ha desarrollado un trabajo a jornada completa (126 horas mensuales), aquellos con una dedicación a media jornada (53 horas mensuales) y aquellos en los que, por cuestiones personales o académicas, no se ha podido dedicar tiempo al proyecto.

La duración total son 22,5 meses. Sin contar los meses sin actividad, se ha trabajado durante 17,5 meses, y las horas dedicadas han sido 1700 (equivalentes a 13,5 meses a jornada completa).

En el siguiente diagrama de Gantt, se ve el tiempo dedicado a las actividades:

Documentación	01/08/2011	30/09/2011
Codificación de estructuras	01/09/2011	30/09/2011
Diseño Forwarding	01/02/2012	30/03/2012
Diseño y programación Link	15/02/2012	14/03/2012
Diseño y codificación Message	02/04/2012	30/04/2012
Diseño Topology, codificación Chord	01/06/2012	14/09/2012
Diseño y programación Storage	03/09/2012	30/11/2012
Codificación Forwarding y tareas	01/11/2012	31/12/2012
Codificación Reload Interface y SIP	01/01/2013	31/01/2013
Pruebas y corrección errores	01/01/2013	29/03/2013
Redacción memoria	18/02/2013	15/06/2013
Redacción artículo de investigación	01/05/2013	31/05/2013

GANTT project

agosto 2011

1 2 3 4 5 78 9 10 11 12 15 16 17 18 19 22 23 24 25 26 29 30 31

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

GANTT project

septiembre 2011

1 2 3 5 6 7 8 9 12 13 14 15 16 19 20 21 22 23 26 27 28 29 30

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

GANTT project

febrero 2012

1 2 3 6 7 8 9 10 13 14 15 16 17 20 21 22 23 24 27 28 29

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

GANTT project

marzo 2012

1 2 45 6 7 8 9 12 13 14 15 16 19 20 21 22 23 26 27 28 29 30

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

GANTT project

abril 2012

2 3 4 5 6 89 10 11 12 13 16 17 18 19 20 23 24 25 26 27 30

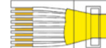
Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

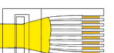
GANTT project

junio 2012

1 2 4 5 6 7 8 9 11 12 13 14 15 18 19 20 21 22 25 26 27 28 29

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13





julio 2012

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

agosto 2012

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

septiembre 2012

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

octubre 2012

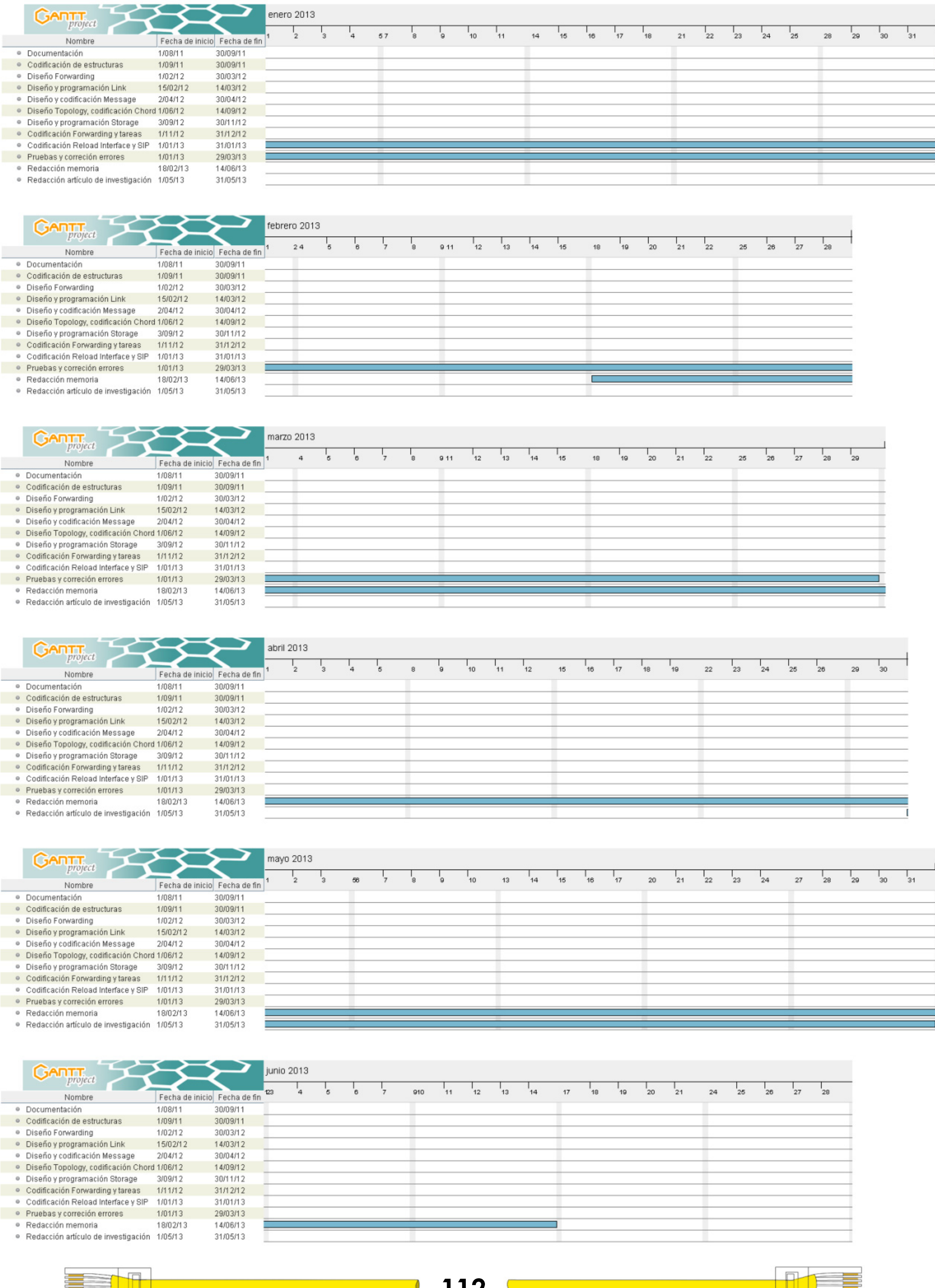
Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

noviembre 2012

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13

diciembre 2012

Nombre	Fecha de inicio	Fecha de fin
Documentación	1/08/11	30/09/11
Codificación de estructuras	1/09/11	30/09/11
Diseño Forwarding	1/02/12	30/03/12
Diseño y programación Link	15/02/12	14/03/12
Diseño y codificación Message	2/04/12	30/04/12
Diseño Topology, codificación Chord	1/06/12	14/09/12
Diseño y programación Storage	3/09/12	30/11/12
Codificación Forwarding y tareas	1/11/12	31/12/12
Codificación Reload Interface y SIP	1/01/13	31/01/13
Pruebas y correción errores	1/01/13	29/03/13
Redacción memoria	18/02/13	14/06/13
Redacción artículo de investigación	1/05/13	31/05/13


GANTT project
enero 2013
febrero 2013
marzo 2013
abril 2013
mayo 2013
junio 2013
Nombre
Fecha de inicio
Fecha de fin
Documentación 1/08/11 30/09/11
Codificación de estructuras 1/09/11 30/09/11
Diseño Forwarding 1/02/12 30/03/12
Diseño y programación Link 15/02/12 14/03/12
Diseño y codificación Message 2/04/12 30/04/12
Diseño Topology, codificación Chord 1/06/12 14/09/12
Diseño y programación Storage 3/09/12 30/11/12
Codificación Forwarding y tareas 1/11/12 31/12/12
Codificación Reload Interface y SIP 1/01/13 31/01/13
Pruebas y correción errores 1/01/13 29/03/13
Redacción memoria 18/02/13 14/06/13
Redacción artículo de investigación 1/05/13 31/05/13

PRESUPUESTO

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:
Marcos López Samaniego

2.- Departamento:
Ingeniería Telemática

3.- Descripción del Proyecto:

- Titulo	**Implementación del protocolo RELOAD**
- Duración (meses)	**17,5**
Tasa de costes Indirectos:	**20%**

4.- Presupuesto total del Proyecto (valores en Euros):
40.000,00 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	NIF	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)
López Samaniego, Marcos		Ingeniero técnico	13	2.155,51	28.021,67
Martínez Yelmo, Isaías		Ingeniero Senior	0,5	4.289,54	2.144,77
González Sánchez, Roberto		Ingeniero	0,5	2.694,39	1.347,20
		Hombres × mes = 14		**Total**	31.513,63

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1.575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)
Dedicación Marcos: 13,5 meses efectivos trabajados × 126 h/mes = 1701 h
1701 h / 131,25 h = 12,96 hombres × mes

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador sobremesa i7 950	1.200,00	100	17,5	60	350,00
Periféricos ordenador	900,00	100	17,5	60	262,50
Ordenador portátil Asus 1215N	450,00	50	11	60	41,25
Licencia Office Hogar y Empresa	222,31	100	11	30	81,51
Licencia Office Visio Estándar	329,75	100	11	30	120,91
Licencia Photoshop Elements	79,00	100	11	30	28,97
				Total	885,14

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable	
Ilustraciones y diseño 3D	Angelines Amaro Gómez, diseñadora	8 h × 40 €/h	320,00
Consultas programación	Daniel Escoz, ingeniero informático	1 h × 50 €/h	50,00
	Total		370,00

OTROS COSTES DIRECTOS DEL PROYECTO [e]

Descripción	Empresa	Coste imputable	
Conexión a internet 100/10 Mbps	Movistar	17,50 × 59,95 × 50%	524,56
Material de oficina			40,00
	Total		564,56

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas...

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	31513,63
Amortización	885,14
Subcontratación de tareas	370,00
Costes de funcionamiento	564,56
Costes Indirectos	6666,67
Total	40000,00

CONTENIDOS

Resumen **11**

1. Introducción **13**

1.1. Introducción al Trabajo Fin de Grado.. 15

1.2. Contenidos de la memoria 16

2. Estado del arte **17**

2.1. Redes P2P superpuestas 19

2.1.1. Chord 20

2.1.2. Kademlia 22

2.2. Introducción a RELOAD 23

2.3. Arquitectura de RELOAD 25

2.3.1. Direccionamiento 26

2.3.2. Capa de uso 27

2.3.3. Transporte de mensajes / Message Transport........................... 28

2.3.4. Almacenamiento / Storage 28

2.3.5. Complemento de topología / Topology Plugin 29

2.3.6. Capa de reenvío y manejo de enlaces / Forwarding & Link Management.. 30

2.3.7. Capa de enlace / Link 30

2.4. Mensajes en RELOAD 31

2.5. Otros aspectos de RELOAD 38

2.5.1. Encaminamiento 38

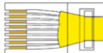
2.5.2. Clientes 39

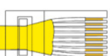
2.5.3. Chord en RELOAD 40

2.5.4. Seguridad 40

2.5.5. Transversal NAT 40

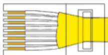
2.5.6. Herramientas de trabajo 41

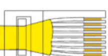




3. Objetivos 43
3.1. Objetivos Generales 45
3.2. Trabajo actual 46
3.3. Trabajos futuros 48

4. Diseño e implementación 51
4.1. Fases del desarrollo 53
4.1.1. Documentación 53
4.1.2. Codificación de estructuras 53
4.1.3. Diseño de Forwarding 53
4.1.4. Diseño y programación del nivel de enlace 54
4.1.5. Diseño y codificación de Message Transport 54
4.1.6. Diseño de Topology Plugin, codificación de Chord 54
4.1.7. Diseño y programación de Storage 55
4.1.8. Codificación de Forwarding y tareas 55
4.1.9. Codificación de Reload Interface y del uso (SIP) 55
4.2. Vista general 56
4.2.1. Organización del código 56
4.2.2. Llamadas entre los módulos 57
4.2.3. Aclaraciones 59
4.3. Capa de enlace (Link) 60
4.4. Capa de interred: Forwarding & Link Management 62
4.5. Capa de transporte: Message Transport 65
4.6. Capa de transporte: Storage 68
4.7. Capa de encaminamiento: Topology Plugin 72
4.8. Capa de aplicación: Uso 75
4.9. Clases comunes 78
4.9.1. Excepciones 78
4.9.2. Errores 78
4.9.3. Tareas 78
4.9.4. Otros 79
4.10. Funcionamiento general 80



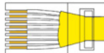


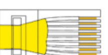
4.11. Comentarios acerca del desarrollo 81
4.11.1. Dificultad en la comprensión 81
4.11.2. Dificultad en el diseño..... 81
4.11.3. Dificultad en la implementación..... 82

5. Resultados 85
5.1. Escenario para las pruebas 87
5.2. Pruebas en red local 88
5.2.1. Registro en la red 88
5.2.2. Obtención y eliminación de datos..... 89
5.2.3. Réplicas 90
5.3. Mensajes 91
5.3.1. Tráfico Join 91
5.3.2. Tráfico Fetch..... 92
5.3.3. Tráfico Store 93
5.4. Pruebas en internet..... 96

6. Conclusiones..... 97
6.1. Conclusiones generales..... 99
6.1.1. Funcionamiento 99
6.1.2. La implementación en cifras..... 100
6.1.3. Documentación..... 100
6.2. Conclusiones personales..... 101
6.2.1. Aportaciones profesionales 101
6.2.2. Mejoras propuestas 101
6.2.3. Críticas..... 103

Referencias 105
Apéndices 109
Planificación 109
Presupuesto 113
Contenidos 115
Agradecimientos 119





AGRADECIMIENTOS

Quisiera agradecer, en primer lugar, a dos de las personas que me han prestado su ayuda y, sin las cuales, hubiera sido difícil sacar el trabajo adelante: Marc Petit-Huguenin, colaborador del IETF y una persona cordial y cercana; y Daniel Escoz Solana "Darkhogg", pequeño genio de la informática y programador de alta categoría.

A Angelines Amaro Gómez, diseñadora gráfica, por su trabajo desinteresado en los esquemas y por el fantástico modelado en tres dimensiones de la portada.

A mi *princess*, Beatriz Amaro Gómez, por estar siempre a mi lado y por los estupendos dibujos que adornan los títulos.

A Carlos López Infante "Clopezi", a mi padre y a la familia Amaro por permitirme realizar las pruebas utilizando sus ordenadores.

A Felis, por su apoyo moral y sus ronroneos.

A mi madre, por estar siempre ahí.

También y muy especialmente a la Fundación Yoga: a Pedro e Isabel, por su sustento incondicional en todos los sentidos, gracias al cual se ha podido editar esta memoria con semejante calidad.

A los profesores de toda la carrera, especialmente a José Manuel Ruiz de Marcos, por despertar mi interés en la teoría de la señal y enseñar con ese entusiasmo.

Y, por último, a los topánganos y al señor, por hacerme más fácil la carrera: Jose, Mari, Edu, Chus, al otro Jose y especialmente a Noe, por sus apuntes, y a Manu, por aguantar mis charlas de Sistemas Lineales y TDI.