Proyecto Fin de Carrera
Ingeniería en Informática

Gestión de comunicación en tiempo real en redes Ethernet conmutadas

José Javier Colomer Vieitez

Director: Klas Nilsson
Supervisores: Anders Blomdell y Sven Gestegård-Robertz
Ponente: José Luis Villarroel Salcedo

Departamentos de Ciencias de la Computación y de Control Automático
Universidad de Lund, Suecia

Departamento de Informática e Ingeniería de Sistemas
Centro Politécnico Superior
Universidad de Zaragoza

Zaragoza, Julio de 2011

Gestión de comunicación en tiempo real en redes Ethernet conmutadas
RESUMEN

Un sistema distribuido es una colección de computadoras que se comunican entre sí a través de una red de comunicaciones y cooperan hacia la consecución de un objetivo común, siendo percibidas por el usuario como un único sistema. Cuando un sistema distribuido lleva a cabo tareas para las que el tiempo de respuesta es crítico, se lo denomina sistema de tiempo real distribuido. Algunos ejemplos de aplicación de estos sistemas son: prevención de accidentes automovilísticos (e.g. frenos ABS y sistemas de regulación de velocidad o autocrucero), control remoto de robots a partir de la información recogida por sensores, implementación de sistemas de control industrial, etc.

A pesar de su extenso uso, hay una falta de especificaciones y técnicas estándar para comunicar de forma fácil y flexible los componentes de estos sistemas. Por ello, la mayor parte de los sistemas distribuidos emplean soluciones de comunicación a medida, específicamente diseñadas para funcionar en el entorno en que están situados los componentes del sistema, pero difícilmente aplicables a entornos de características diferentes.

El objetivo de este proyecto no es proponer un nuevo estándar de comunicación para dichos sistemas, sino desarrollar una solución de comunicación en tiempo real que pueda ser fácilmente desplegada en una amplia variedad de entornos, liberando así de la tarea de diseñar una infraestructura de comunicación adaptada a cada situación específica.

Este proyecto continúa la línea de trabajo iniciada por los departamentos de Ciencias de la Computación y de Control Automático de la Universidad de Lund, cuyo trabajo previo llevó al diseño e implementación de una herramienta para generar automáticamente infraestructuras de comunicación entre aplicaciones (LabComm) y a la especificación de un protocolo de comunicación en tiempo real basado en reserva de ancho de banda en redes Ethernet conmutadas (ThrottleNet).

La realización de este proyecto ha conllevado el diseño e implementación de dos sistemas independientes:

- Una aplicación interactiva que facilite el proceso de aprendizaje de LabComm. En esencia, esta aplicación será un simulador de transmisiones de datos que mostrará las virtudes y forma de uso del sistema LabComm.
- Una implementación de ThrottleNet, a partir de la especificación proporcionada por estos departamentos.

Agradecimientos

A mi director de proyecto, Klas Nilsson, por toda su ayuda y por haberme dado la posibilidad de trabajar en temas que me han parecido fascinantes.

A Sven, por haberme propuesto la aventura de implementar ThrottleNet.

A Anders Blomdell, por su interminable paciencia, por todo lo que me ha enseñado y por haberme demostrado que "todo es posible en Linux, si sabes dónde buscar en el núcleo".

A José Luis Villarroel, por su orientación y consejos como ponente.

A mis amigos de Lund, por haberme regalado dos años inolvidables.

A mis amigos del CPS, por las risas, los muses y los buenos ratos que han ayudado a superar la carrera.

A todos mis amigos, por estar siempre a mi lado y ser como sois.
Especialmente, a Violeta y a Clara, porque me conocen como nadie y son las dos hermanas que nunca he tenido.

También a mi familia, por todo su apoyo y cariño.

Y, por supuesto, a mi peludo amigo Pitt.

Pero, sobre todo, a mis padres. Por vuestra infinita paciencia, cariño y dedicación, que me han acompañado siempre. De no ser por vosotros no habría terminado cuerdo este invierno, aunque en el proceso os volviera yo locos a vosotros.

Muchas gracias a todos vosotros, y a todos aquellos que olvido nombrar.

Índice general

1. Introducción **1**
- 1.1. Objetivos y alcance del proyecto . . . 3
- 1.2. Análisis de los problemas planteados . . . 5
  - 1.2.1. Facilitar el proceso de aprendizaje de LabComm . . . 5
  - 1.2.2. Comunicación en red en tiempo real: posibles alternativas 7
- 1.3. Estructura del documento . . . 8

2. Facilitando el proceso de aprendizaje de LabComm **11**
- 2.1. Interactuando con el simulador . . . 12
- 2.2. El cliente . . . 18
- 2.3. El servidor . . . 19
- 2.4. Comunicación cliente-servidor . . . 20
- 2.5. Ejecución de simulaciones . . . 22
  - 2.5.1. Aislamiento de otros usuarios . . . 22
  - 2.5.2. Un marco para la ejecución de simulaciones . . . 24
  - 2.5.3. Comunicación productor-consumidor . . . 27
- 2.6. Detalles técnicos de la solución . . . 28
  - 2.6.1. *Applets* no firmados . . . 28
  - 2.6.2. Recarga dinámica de clases . . . 28

3. ThrottleNet: comunicación en red en tiempo real **31**
- 3.1. Implementación como *driver* . . . 33
- 3.2. Racionamiento de tráfico . . . 34
  - 3.2.1. Tráfico saliente . . . 35
  - 3.2.2. Tráfico entrante . . . 36
- 3.3. GlobeThrottle . . . 37
  - 3.3.1. Conexión a la red ThrottleNet . . . 37
  - 3.3.2. Gestión del ancho de banda . . . 38
  - 3.3.3. Organizador de tráfico NRT . . . 40
- 3.4. Tipos de tráfico en una red ThrottleNet . . . 44
  - 3.4.1. Tráfico RT . . . 45

3.4.2. Tráfico NRT 46
3.5. Tratamiento de ARP en situaciones de alta carga NRT 48

4. Gestión del proyecto **49**
4.1. Metodologías de desarrollo y gestión del tiempo 49
4.1.1. El simulador 49
4.1.2. La implementación de ThrottleNet 51
4.2. Tamaño del proyecto 55

5. Conclusiones **57**
5.1. Contribuciones 57
5.2. Cumplimiento de objetivos 58
5.2.1. El simulador 58
5.2.2. La implementación de ThrottleNet 59
5.3. Trabajo futuro 60
5.3.1. Posibles mejoras para el simulador 60
5.3.2. Posibles mejoras para ThrottleNet 60
5.3.3. Pasos hacia la solución integrada 61
5.4. Experiencia personal 61

A. La herramienta de simulación **63**
A.1. Transferencia de ficheros mediante ObjectStreams 63
A.2. Comunicación productor-consumidor 66
A.3. Common mistakes when using the simulator 70
A.3.1. Producer code 70
A.3.2. Consumer code 71

B. La implementación de ThrottleNet **73**
B.1. Racionamiento de tráfico 73
B.1.1. Procesamiento de tráfico saliente 73
B.1.2. Procesamiento de tráfico entrante 75
B.2. Gestión de ancho de banda 77
B.3. Interacción con la interfaz RT vía ioctl 81

C. El sistema LabComm **95**
C.1. Componentes del sistema LabComm 95
C.2. El lenguaje LabComm 95
C.3. Ejemplos de protocolos de comunicación 97
C.3.1. Ejemplo 1 97
C.3.2. Ejemplo 2 97

D. A LabComm tutorial **99**
D.1. Introduction 99
D.2. Step one: Generate the marshalling routines 101
D.3. Step two: Use the marshalling routines 102
D.4. Protocol definition for Example 1 105

Capítulo 1

Introducción

Los sistemas de tiempo real distribuidos requieren de soluciones de comunicación en red en tiempo real apropiadas. Por motivos de coste y flexibilidad, sería deseable que dichas redes no requiriesen de *hardware* especial extra (e.g. ProfiBus, FlexRay, TTP y similares) y que sus protocolos básicos fueran abiertos para permitir adaptaciones a una amplia variedad de dispositivos y sistemas. Aunque los aspectos puramente técnicos (sincronización de relojes, compresión, planificación, etc) de dichas redes y sistemas han sido ampliamente investigados, existen ciertos aspectos en los que es necesario profundizar:

- Comunicación en tiempo real empleando *hardware* estándar y de bajo coste.
- Eficiencia en tiempo de ejecución dentro de los nodos que se comunican.
- Intercambio de mensajes fuertemente tipados, para asegurar una correcta interpretación de los datos intercambiados.

Con el fin de dar solución a los aspectos arriba mencionados, los departamentos de Ciencias de la Computación y de Control Automático de la Universidad de Lund han desarrollado LabComm[1] y ThrottleNet[2].

LabComm es un sistema diseñado para generar automáticamente una infraestructura de comunicación entre aplicaciones basada en paso de mensajes fuertemente tipados. Esta infraestructura consiste en rutinas de codificación/decodificación de mensajes (de ahora en adelante, rutinas de serialización[3]) que permiten generar e interpretar mensajes fuertemente tipados. Como los mensajes están tipados, la aplicación receptora siempre sabe cómo reconstruir el mensaje recibido (i.e. sabe

cómo interpretar el contenido del mensaje, los datos intercambiados).

El usuario ha de proporcionar a LabComm una especificación de los tipos de datos que se van a transmitir. A partir de esta especificación (que llamaremos protocolo), LabComm generará automáticamente las rutinas de serialización requeridas. De esta manera LabComm permite generar mecanismos de comunicación de manera sencilla y transparente al usuario, consiguiendo abstracción del contenido de los mensajes, comunicación fuertemente tipada e independencia de los lenguajes en que están escritas las aplicaciones que se comunican.
El Apéndice C de este documento ofrece una descripción más detallada del sistema LabComm.

ThrottleNet es una especificación de un protocolo de comunicación en tiempo real basado en reserva de ancho de banda en redes Ethernet conmutadas. ThrottleNet planifica la transmisión de los mensajes para asegurar su entrega a tiempo. El comportamiento inherentemente impredecible de las redes de comunicación, que imposibilita conocer el peor tiempo de entrega de un mensaje (a causa de las colisiones entre mensajes y de los retardos de retransmisión que estas colisiones conllevan), hace impracticable esta aproximación al problema. A fin de conseguir un escenario libre de colisiones, ThrottleNet impone restricciones acerca de cómo han de estar los nodos conectados a la red y de cómo se han de comunicar. ThrottleNet asume una topología de estrella (Figura 3.1) en la que cada nodo está conectado a los demás a través de un enlace *full-duplex* a un conmutador. La red queda entonces dividida en un conjunto de dominios de colisión aislados, cada uno de ellos conteniendo únicamente a un nodo que puede enviar y recibir datos a y de los otros nodos simultáneamente. Para garantizar la ausencia de colisiones hay que impedir el uso excesivo de los enlaces físicos hacia el conmutador, de ahí la necesidad de reservar ancho de banda para poder comunicarse.

Aunque estas dos técnicas han sido desarrolladas independientemente hasta ahora, la posibilidad de complementarlas para tratar de dar solución a las necesidades anteriormente mencionadas es la base de este proyecto, que se ha desarrollado en colaboración con los departamentos anteriormente citados.

1.1. Objetivos y alcance del proyecto

Pretendemos ahora combinar las técnicas anteriormente mencionadas con el fin de lograr una solución integrada para gestionar comunicación en tiempo real en redes Ethernet conmutadas. El objetivo final de esta solución es ser capaces de desplegar una red de tiempo real en la que se puedan transmitir datos a través de servicios. Un servicio es una abstracción usada para referirnos a un canal de transmisión unívocamente identificado que garantiza la entrega a tiempo de los mensajes que transmite, los cuales están codificados de acuerdo con un protocolo LabComm conocido en la red. Al ser este protocolo conocido, cualquier nodo que desee utilizar un servicio sólo ha de conseguir la definición del protocolo LabComm para ese servicio y utilizar LabComm para generar las rutinas de serialización correspondientes. Hecho esto, el nodo puede enviar/recibir datos a través del servicio sin necesidad de saber cómo se formatearán los datos transmitidos, pero con la seguridad de que serán correctamente interpretados y llegarán a tiempo a su destino.

Podemos implementar servicios si combinamos LabComm y ThrottleNet: ThrottleNet proporcionará canales de transmisión de tiempo real y LabComm serializará[3] los mensajes transmitidos. Un servicio implementado así garantizará que los mensajes sean entregados a tiempo y correctamente interpretados y será, por tanto, un medio fiable de comunicación en tiempo real.
La red de tiempo real anteriormente citada funcionará en base al protocolo ThrottleNet y será llamada, por tanto, red ThrottleNet, mientras que a sus nodos nos referiremos como nodos ThrottleNet.

A continuación se exponen los objetivos detallados de este proyecto, con los que se espera contribuir al desarrollo de la solución integrada antes nombrada:

1. Ayudar a los nuevos usuarios de LabComm a comprender qué puede hacer y cómo funciona. Una aplicación interactiva será diseñada para permitir experimentar de forma educativa con esta aproximación a la comunicación basada en paso de mensajes. Esta aplicación deberá cumplir con los siguientes requisitos:

    1.1 Permitir a los usuarios diseñar simulaciones personalizadas que utilicen LabComm para la transmisión de datos. Los usuarios podrán ejecutarlas y examinar sus resultados.

1.2 Minimizar las gestiones de configuración de la aplicación por parte de los usuarios. La aplicación no debería requerir de configuración alguna para poder funcionar.

2. Diseñar e implementar una red de tiempo real eficiente que pueda ser fácilmente desplegada en *hardware* estándar de bajo coste y en una amplia gama de sistemas. La red deberá cumplir con los siguientes requisitos:

   2.1 Gestionar el ancho de banda como un recurso distribuido compartido que puede ser reservado por los nodos ThrottleNet para sus comunicaciones, sean éstas de tiempo real (comunicación RT) o no (comunicación NRT).

   2.2 Emplear técnicas de racionamiento de tráfico (*traffic throttling*) para impedir que los nodos ThrottleNet se excedan en su uso del ancho de banda. Este racionamiento se efectuará de forma local a cada nodo de acuerdo con el ancho de banda que tenga el nodo asignado.

   2.3 Facilitar canales de comunicación adecuados para la comunicación que no sea de tiempo real (comunicación NRT).

   2.4 Proporcionar canales de comunicación apropiados para la comunicación que sí sea de tiempo real (comunicación RT, los anteriormente mencionados servicios). A fin de reflejar las diferentes prioridades y necesidades de estos servicios, estos canales podrán ser configurados en cuanto a la máxima frecuencia a la que pueden mandar mensajes y a cuán largos estos mensajes pueden ser, dentro de lo razonable. Esta configuración dará como resultado una reserva de ancho de banda por canal.

   2.5 Ser capaz de detectar qué nodos en la red continúan conectados con el fin de poder descartar mensajes dirigidos a nodos que ya no estén conectados a la red y así evitar ese consumo de ancho de banda.

   2.6 Ofrecer una perspectiva amigable (para los humanos) de los nodos ThrottleNet en la red y de los servicios a los que está suscrito cada nodo.

Una vez cumplidos estos objetivos, una implementación de una red ThrottleNet estará lista para ser usada. No obstante, la previamente mencionada abstracción de "servicio" no estará disponible en esta red de tiempo real, puesto que ThrottleNet no formatea por sí solo los datos que transporta. Por

consiguiente, el último paso para obtener la solución integrada será combinar LabComm y ThrottleNet para que todos los datos enviados a través de la red ThrottleNet estén estructurados de acuerdo con un formato bien conocido en la red. Este paso no será acometido en este proyecto sino que queda pendiente como trabajo futuro. Más detalles en la sección 5.3.3.

1.2. Análisis de los problemas planteados

1.2.1. Facilitar el proceso de aprendizaje de LabComm

Aceleraremos el proceso de aprendizaje de LabComm ofreciendo a los usuarios una aplicación *software* interactiva que permita experimentar con LabComm mediante simulaciones. Como LabComm es usado para codificar y decodificar datos enviados a través de un flujo (*stream*) de datos unidireccional, hemos decidido que el simulador presente un escenario con un productor y un consumidor. No obstante, el interés aquí no radica en cómo sincronizar al productor y al consumidor sino más bien en cómo ellos codifican y decodifican los datos que intercambian usando LabComm. Es vital que el usuario pueda modificar los agentes que controlan las simulaciones, pues esto permite estudiar las variaciones en los resultados obtenidos y las posibilidades de aplicación. Se pueden procesar los datos de diferentes maneras antes de su transmisión y después de su recepción (e.g. si un sensor proporciona medidas de temperatura en grados Fahrenheit pero el robot que las utiliza requiere que estén en grados Celsius, esta conversión puede ser realizada de forma automática por las rutinas de decodificación). Este simulador proporciona una plantilla de protocolo LabComm que especifica los datos que serán transmitidos y plantillas de código que implementan un productor y un consumidor. El usuario podrá modificar, compilar y ejecutar el código fuente de estos tres elementos como desee, consiguiendo así un control total de la simulación.

Sería difícil satisfacer el requisito 1.2 si escribiéramos este *software* como una aplicación estándar y la ofreciéramos para su descarga. El usuario tendría que afrontar todo tipo de problemas, entre los que se incluyen: adquirir un compilador de LabComm, aprender a usarlo, configurar sus compiladores[1] para colaborar con el simulador, etc.

[1]LabComm puede generar rutinas de serialización en código Java, C, C# y Python. Si el usuario quisiera escribir código C para su productor y código Python para su consumidor, los compiladores de estos dos lenguajes habrían de ser puestos a disposición del simulador para poder automatizar el proceso de compilación de las simulaciones.

Por tanto, esta aplicación será diseñada como una aplicación Web, lo que permitirá que:

- La aplicación ejecutándose en el lado del cliente sea ligera (e.g. un *applet*), no requiera configuración alguna y toda la sobrecarga de generación de código y manejo de ficheros se traslade al servidor.
- El usuario no se sienta desprotegido mientras utilice la aplicación, puesto que los *applet*s Java[4][5] son ejecutados bajo restricciones de seguridad muy severas[1][6].

Finalmente, con el fin de mostrar que LabComm puede trabajar sobre cualquier flujo de datos unidireccional sin importar su naturaleza (ya sea una *pipe*, un *socket*, etc.), el simulador permite decidir si el productor y el consumidor se ejecutarán en la misma máquina (ya sea la del cliente o el servidor) o si lo harán en máquinas diferentes. Ello muestra como LabComm es aplicable a prácticamente cualquier tipo de comunicación.

Se ha elegido Java para escribir esta aplicación por las siguientes razones:

- El sistema LabComm (nos referimos al sistema en sí, no a las rutinas de serialización generadas para comunicar aplicaciones) está escrito completamente en Java, y escribir esta aplicación en Java facilita la integración de ambas partes.
- La funcionalidad de carga de clases de Java hace que sea muy sencillo ejecutar las simulaciones ¡e incluso cargar el código (binarios) del servidor en el cliente!
- Crear una aplicación ligera con una interfaz gráfica que resulte agradable al usuario es muy sencillo en Java, gracias a la clase Applet[4][5] y a las librerías Swing[7].

[1]El modelo de cajón de arena de Java está basado en una defensa en tres capas: verificación de bytecodes previa a su ejecución, uso de un cargador de clases específico para impedir intentos de sustitución de clases Java esenciales por parte del *applet* y un *security manager* que genera excepciones de seguridad al más mínimo comportamiento sospechoso por parte del *applet*.

1.2.2. Comunicación en red en tiempo real: posibles alternativas

Pese a que ya existe al menos una solución para la comunicación en tiempo real sobre *hardware* Ethernet estándar, RTnet[8], este proyecto opta por implementar el protocolo ThrottleNet. Los motivos para esto son varios:

- **Facilidad de despliegue**: RTnet sólo está disponible para Xenomai[9] y RTAI[10] mientras que ThrottleNet puede ser implementado para una variedad mayor de plataformas, como sistemas UNIX y similares (incluyendo Mac OS y distribuciones Linux) e incluso Windows.

- **Eliminar limitaciones de los protocolos de paso de testigo**. Por su propia naturaleza, estos protocolos limitan la eficiencia de aquellas redes en las que existen nodos con velocidades de procesamiento o transmisión de datos muy dispares, pues los nodos más rápidos ven limitada la velocidad a la que pueden trabajar debido al uso de técnicas de control de flujo[11]:

  - Si existen nodos que procesan datos a velocidades muy diferentes, los nodos más rápidos tendrán que disminuir su velocidad de transmisión para no saturar a los nodos más lentos.
  - Si existen conmutadores en la red con diferentes velocidades de transmisión, los conmutadores más rápidos no trabajarán al máximo de sus capacidades, sino a una velocidad apropiada para los conmutadores más lentos. Ello supone un desaprovechamiento de los recursos disponibles.

  RTnet es un protocolo de paso de testigo y, por tanto, se ve afectado por estas limitaciones. ThrottleNet, en cambio, no se ve afectado porque no está basado en paso de testigo.

Como queremos que la red no sólo sea eficiente sino que además pueda ser fácilmente desplegada en una amplia variedad de plataformas *hardware* y *software* (Objetivo 2 del proyecto), optaremos por implementar ThrottleNet.

ThrottleNet será instalado en cada nodo de la red para que el racionamiento de tráfico se efectúe de forma local a cada nodo. Aunque una implementación de espacio de usuario sería más sencilla y directa, se ha elegido

implementarlo como una aplicación de espacio de núcleo para lograr una mayor eficiencia[1] y facilitar la portabilidad a otros sistemas operativos en los que ThrottleNet habrá de ser desplegado (e.g. Xenomai y VxWorks[12]). Por ello, escribiremos ThrottleNet completamente en C como un *driver* de red de Linux[13].

Esta implementación de ThrottleNet utiliza un GlobeThrottle (ver Sección 3.3) para tareas de gestión de ancho de banda. GlobeThrottle también ha sido escrito en C por motivos de reutilización de código, pero será diseñado como una aplicación de espacio de usuario con el fin de mostrar cómo el racionamiento de tráfico y su planificación también pueden hacerse en espacio de usuario de forma eficiente.

1.3. Estructura del documento

Esta memoria consta de otros cuatro capítulos que resumen el proyecto en su totalidad:

- El capítulo 2 presenta la herramienta de simulación diseñada para facilitar el aprendizaje de LabComm.
- En el capítulo 3 se describe cómo funciona ThrottleNet y se ofrecen detalles específicos del diseño e implementación realizados.
- En el capítulo 4 explica cómo se ha gestionado el proyecto en cuanto a la administración del tiempo disponible y a las metodologías utilizadas.
- Finalmente, el capítulo 5 analiza el cumplimiento de los objetivos planteados, expone las conclusiones alcanzadas y enumera las principales líneas de trabajo que este proyecto deja abiertas.

Por último, los Apéndices ofrecen información adicional que permite cubrir más en detalle algunos aspectos críticos de este trabajo, en particular:

- En el Apéndice A se ofrece información relativa a la herramienta de simulación web.
- El Apéndice B detalla cómo se efectúan algunas tareas críticas en ThrottleNet como, por ejemplo, la gestión de ancho de banda.

[1] Una implementación de espacio de usuario también funcionaría, pero se vería sometida a un mayor número de cambios de contexto y por tanto no sería tan eficiente como su equivalente de espacio de núcleo.

- El Apéndice C ofrece una descripción más profunda de LabComm y muestra el lenguaje que LabComm proporciona para la especificación de protocolos de comunicación junto con algunos ejemplos.
- Por último, en el Apéndice D se incluye un tutorial para LabComm escrito al comienzo del proyecto que pretende ofrecer una primera aproximación a aquellos que desconocen LabComm por completo.

Capítulo 2

Facilitando el proceso de aprendizaje de LabComm

A fin de asistir el proceso de aprendizaje de LabComm, se ha decidido diseñar un simulador de transmisiones de datos que utilice LabComm para codificar la información enviada. El simulador presenta un escenario con un productor y un consumidor y permite al usuario modificar estos para experimentar con diferentes tratamientos de los datos transmitidos.

Como ya se explicó en la Sección 1.2.1, esta aplicación se ha diseñado siguiendo una arquitectura cliente-servidor para facilitar el manejo del simulador y disminuir la carga de trabajo en la computadora del usuario. El cliente de esta aplicación es un *applet* que el usuario ejecuta en su navegador para configurar el simulador como desee. La más notable de estas opciones de configuración es la posibilidad de elegir los computadores en que queremos que se ejecuten el productor y el consumidor. Estos equipos pueden ser tanto aquel en que se ejecuta el cliente como aquel en que se ejecuta el servidor, por tanto ambos subsistemas han sido diseñados para ser capaces de ejecutar simulaciones de forma autónoma o cooperando el uno con el otro.
La Figura 2.1 muestra la arquitectura de la aplicación diseñada junto con sus posibles casos de uso, en función de la localización del cliente y del consumidor. Más adelante, se ilustrarán los componentes del cliente y del servidor y cómo se ejecutan las simulaciones para cada posible combinación de localizaciones del productor y del consumidor.

Naturalmente, además de la funcionalidad compartida para ejecutar simulaciones, cada subsistema añade funcionalidad propia relacionada con las tareas que le corresponden a causa de su localización (en el cliente o en el servidor). La Sección 2.1 explica cómo el usuario interacciona con la aplicación

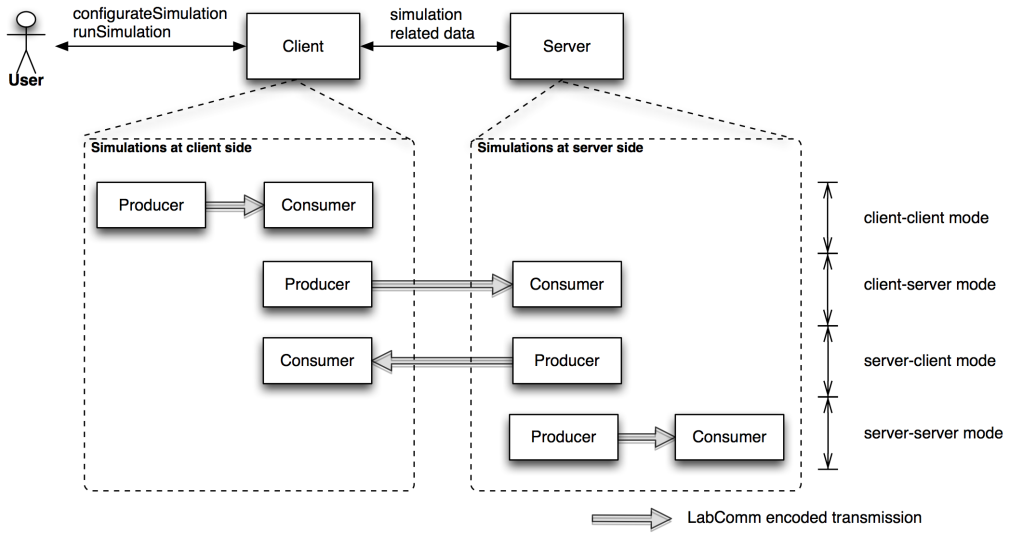

Figura 2.1: Arquitectura del simulador, ilustrando los diferentes modos de trabajo que ofrece en cuanto a la localización del productor y del consumidor (La notación "productor-consumidor" es usada: "cliente-servidor" indica un escenario en que el productor corre en el cliente y el consumidor en el servidor).

y las Secciones 2.2, 2.3 y 2.4 describen al cliente, al servidor y la comunicación entre ambos, respectivamente. Por último, la Sección 2.5 detalla cómo se ejecutan las simulaciones y la Sección 2.6 reseña detalles técnicos de la implementación.

2.1. Interactuando con el simulador

Dado que el cliente reside en la computadora del usuario, su principal función es ofrecer a este una interfaz gráfica desde la que poder manejar de forma cómoda e intuitiva la aplicación. Esta interfaz (ver Figura 2.2) ofrece multitud de opciones para configurar a medida cada simulación. Entre estas opciones están:

- Determinar el computador en que se ejecutarán el productor y el consumidor durante la simulación. Como las únicas opciones son el cliente o el servidor, hay cuatro posibles combinaciones entre las que elegir. La notación "producer - consumer"será utilizada (e.g. "server - client" especifica una simulación en la que el productor residirá en el servidor mientras que el consumidor se ejecutará en el cliente).

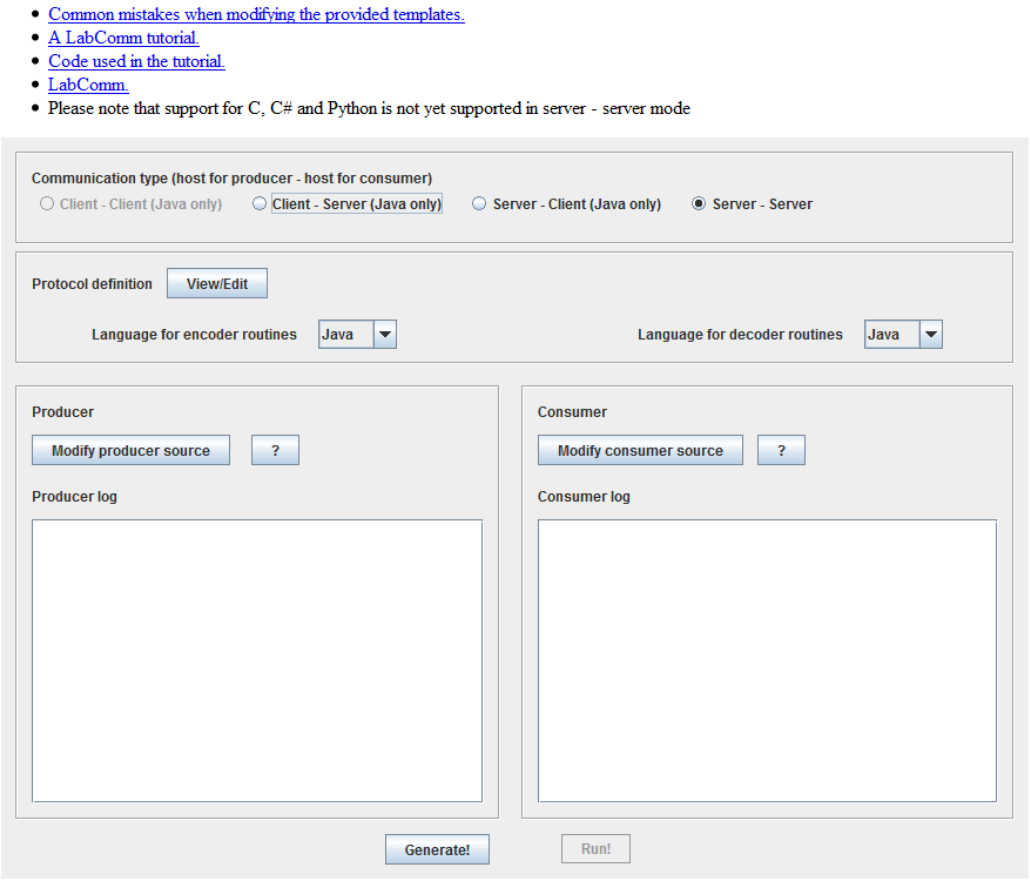

Figura 2.2: Interfaz gráfica ofrecida al usuario para configurar y ejecutar las simulaciones. Permite elegir dónde se ejecutaran el productor y el consumidor, examinar y modificar el código fuente del protocolo LabComm, del productor y del consumidor y enviar peticiones de compilación y ejecución al servidor.

- Examinar la plantilla del protocolo de comunicación proporcionada (ver Figura 2.3) y permitir modificarla para añadir nuevos tipos de datos, modificar los ya existentes o incluso eliminarlos.

- Elegir los lenguajes de programación[1] en que el productor y el consumidor estarán escritos (ver Figura 2.4 para el caso del productor, el caso del consumidor es análogo). La herramienta de simulación configurará el compilador de LabComm para que éste genere las rutinas de serialización en los lenguajes elegidos.

[1] LabComm puede generar rutinas de serialización[3] en Java, C, C# y Python.

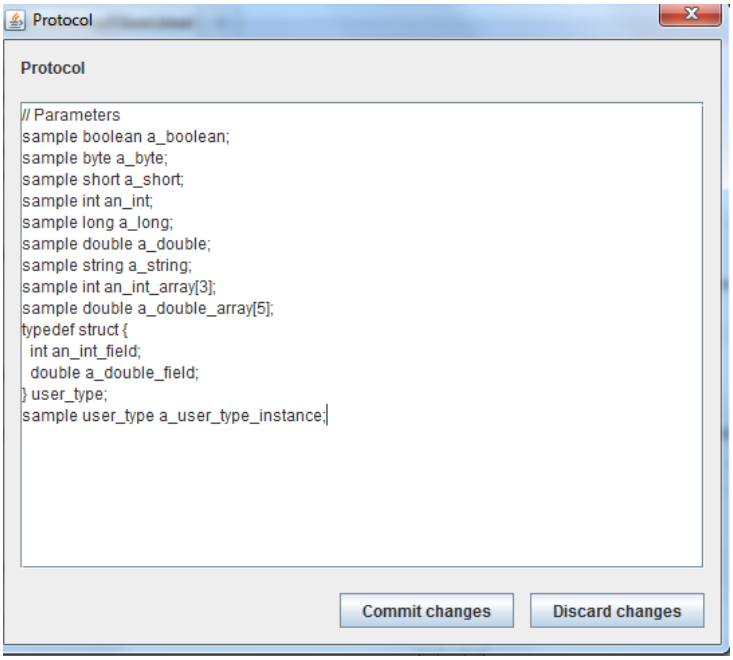

Figura 2.3: Un cuadro de texto que muestra el protocolo LabComm empleado en la simulación. El texto del cuadro puede ser modificado por el usuario y enviado de vuelta al servidor.

- Mostrar las plantillas proporcionadas para el productor y el consumidor y permitir modificarlas (ver Figura 2.5 para el caso del productor, el caso del consumidor es análogo), de manera que puedan hacer uso de los cambios hechos en el protocolo de comunicación para enviar nuevos tipos de datos o modificar la forma en que están procesando los datos que envían o reciben.

- Visualizar información relativa a la ejecución del productor y del consumidor a través de sus ventanas de registro de mensajes.

- Emitir peticiones de compilación al servidor. En caso de haber fallos de compilación, la interfaz gráfica mostrará un mensaje de error que contendrá los errores hallados (ver Figura 2.6) mientras que si no hay errores se permitirá continuar y ejecutar la simulación (ver Figura 2.7).

- Ejecutar la simulación. Tanto en caso de éxito como de error, el resultado de la simulación será mostrado en las ventanas de registro de mensajes del productor y del consumidor. La Figura 2.8 muestra el resultado de una simulación ejecutada sin contratiempos, mientras que las Figuras 2.9 y 2.10 muestran ejemplos de posibles problemas que

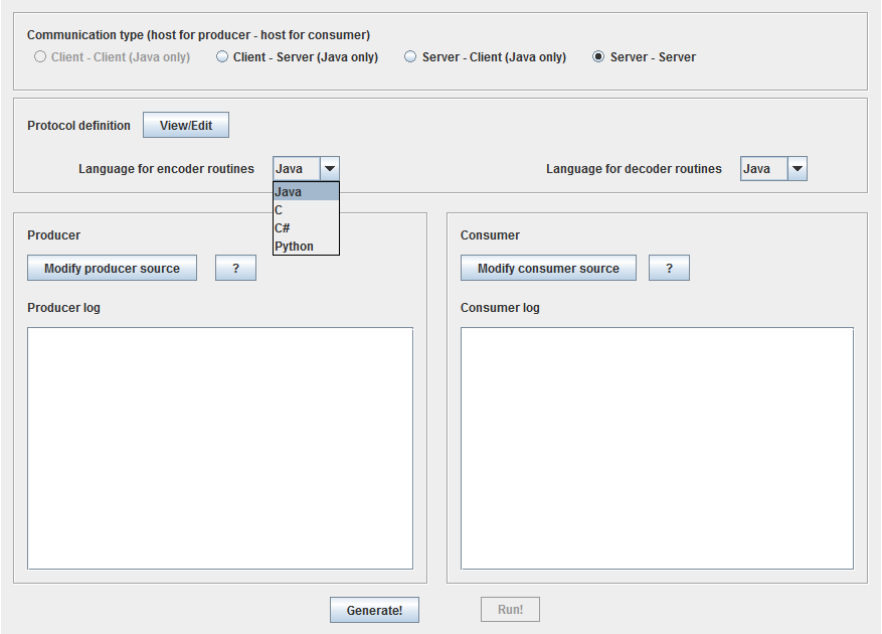

Figura 2.4: Selección del lenguaje de programación en que el productor está implementado. El compilador de LabComm también producirá las rutinas de serialización en ese lenguaje. Las mismas opciones se ofrecen para el consumidor.

pueden ocurrir durante la simulación, debidos a una incorrecta modificación del productor y del consumidor por parte del usuario. Así, y dado que se dan indicaciones acerca de cómo modificar el productor y el consumidor, se espera que el usuario aprenda a través de la experimentación cómo el productor y el consumidor utilizan LabComm para comunicarse.

Cuando el usuario carga el *applet* en su navegador, este utiliza un *socket* para conectarse al servidor desde el que ha sido descargado[1] e inicia una sesión para el usuario. A cada nueva sesión se le asigna un directorio propio que contiene una copia de todas las plantillas de código fuente empleadas en la simulación. Siempre que un usuario modifica un fichero y lo envía de vuelta al servidor, este fichero es depositado en el directorio que corresponde a la sesión del usuario, evitando así interferencias con los demás usuarios.

[1]Dado que se trata de un *applet* no firmado, sólo puede crear conexiones hacia el servidor desde el que ha sido descargado. Ello causará algunos inconvenientes, más información en la Sección 2.6.

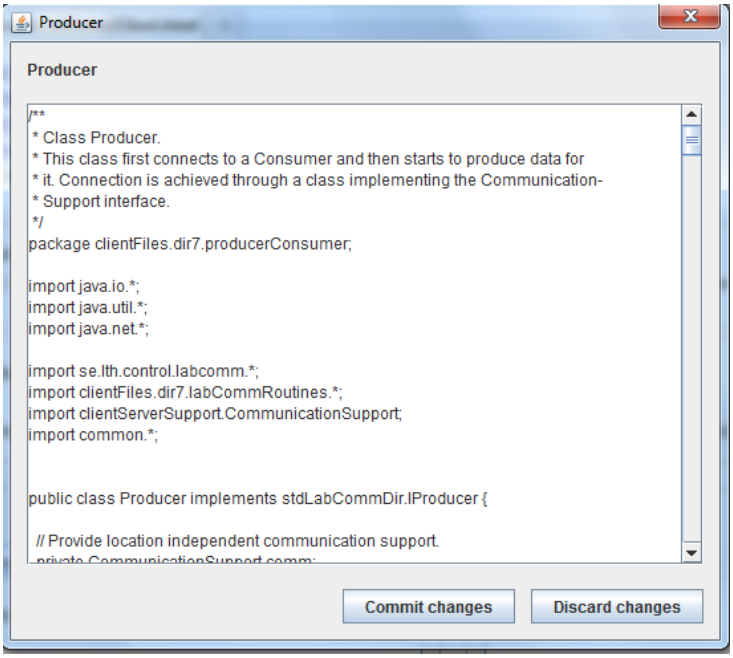

Figura 2.5: Un cuadro de texto mostrando el código fuente del productor. El texto del cuadro puede ser modificado por el usuario y enviado de vuelta al servidor. El consumidor se muestra en una ventana similar.

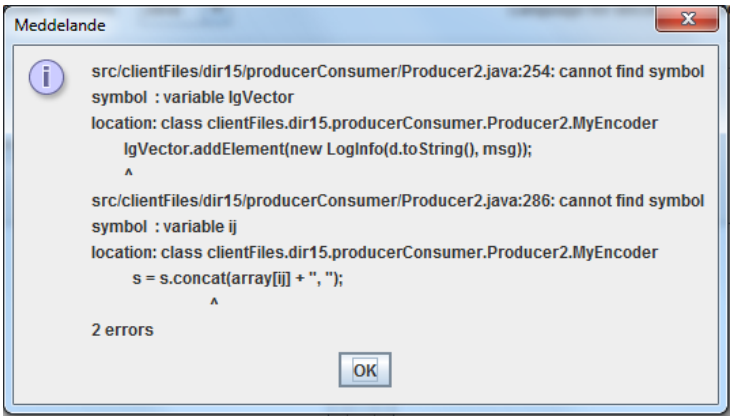

Figura 2.6: Un mensaje de error que lista errores de compilación encontrados.

Los usuarios pueden ver, modificar y enviar el código de la simulación de vuelta al servidor cuantas veces deseen. Cuando emitan una orden de compilación al servidor, este les responderá de un modo u otro en función de si se han dado fallos de compilación o no. Si los ha habido, le serán mostrados al usuario en un diálogo (ver Figura 2.6) mientras que, en caso

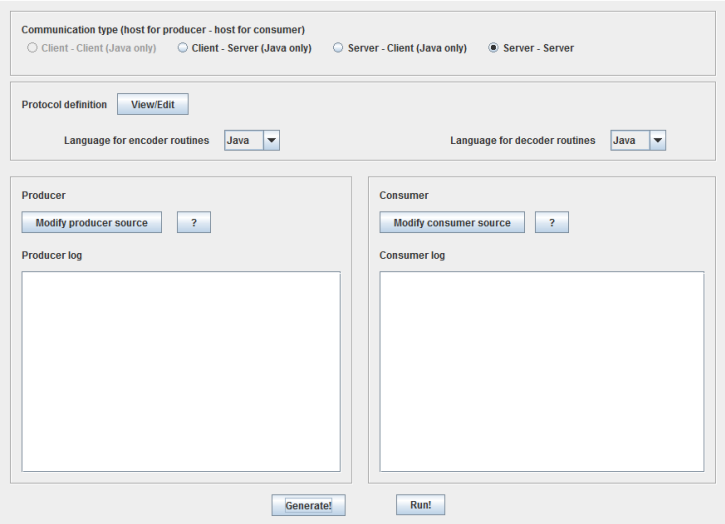

Figura 2.7: Una compilación satisfactoria activa el botón "Run", empleado para ejecutar simulaciones (comparar con las Figuras anteriores, en las que el botón está desactivado).

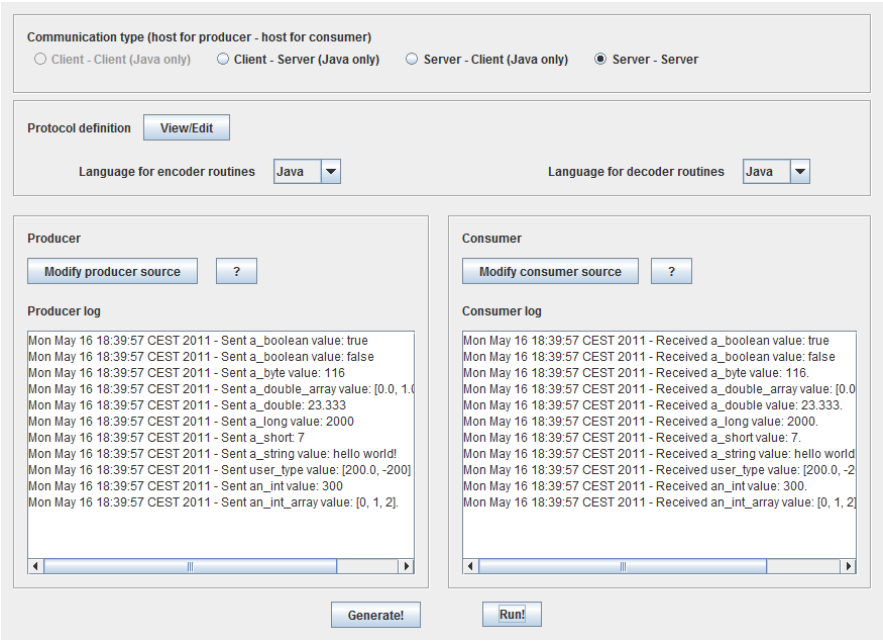

Figura 2.8: En ausencia de contratiempos, las ventanas de registro de mensajes del productor y del consumidor listan todos los eventos de transmisión ocurridos, junto con un *timestamp*.

contrario, se concederá al usuario permiso para ejecutar la simulación (i.e.

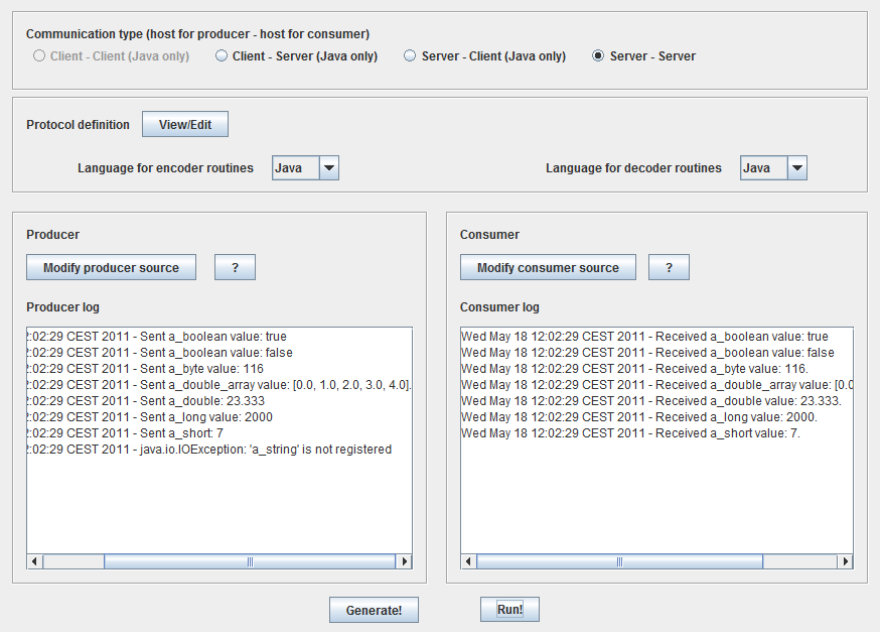

Figura 2.9: La simulación no pudo completarse porque el productor no dispone de la rutina de codificación necesaria para muestras de datos de tipo "a_string".

se habilitará el botón "Run" de la parte inferior de la interfaz gráfica, ver Figura 2.7). Por supuesto, siempre que el usuario modifique el código de la simulación y lo reenvíe al servidor, estos permisos de ejecución le serán retirados y sólo podrán ser recuperados tras una compilación satisfactoria.

2.2. El cliente

El cliente de la aplicación consta de las siguientes clases Java:

- Una interfaz gráfica de usuario embebida como un panel en un ***applet***. El usuario emplea esta interfaz para interactuar con la aplicación y obtener información de las simulaciones.

- Un agente para procesar las peticiones de los usuarios (**UserReqProcessor**) relacionadas con la configuración y ejecución de la simulación. Estas peticiones se han descrito con anterioridad y no merece la pena repetirlas. Este agente llevará a cabo aquellas tareas que puedan ser efectuadas localmente (e.g. ejecutar una simulacion en el cliente) y remitirá aquellas que deban ser efectuadas en el servidor al **ClientCommProtocol**.

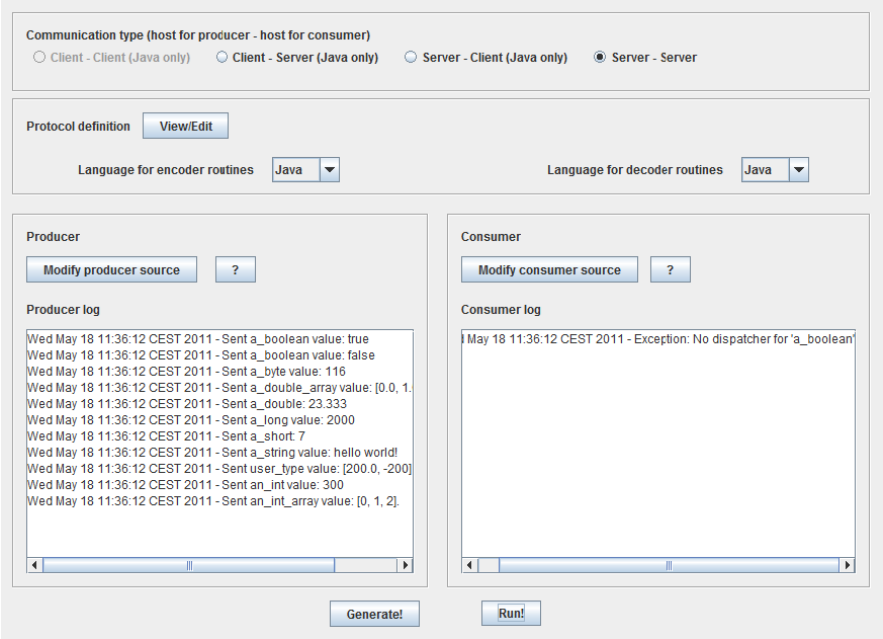

Figura 2.10: La simulación no pudo completarse porque el consumidor no dispone de la rutina de decodificación necesaria para muestras de datos de tipo “a_boolean”.

- Un agente de comunicaciones (**ClientCommProtocol**) que se comunica con su homólogo en el lado del servidor. Obedece órdenes del **ClientReqProcessor** y le transmite la información que recibe del servidor como respuesta a sus peticiones. Esta información puede variar desde una cadena de texto que describe un fallo de compilación a un fichero de código fuente.

2.3. El servidor

El servidor de la aplicación tiene una estructura muy similar a la del cliente pero ofrece además clases adicionales que implementan la funcionalidad extra requerida en el servidor. Consta de las siguientes clases Java:

- Un servidor (**Server**) que espera peticiones de conexión en un *socket*. Este servidor crea un **ServerReqProcessor** en un hilo de ejecución independiente (*thread*) para atender cada nueva conexión.
- Agentes que atienden las peticiones de cada cliente (**ServerReqProcessors**). Estos agentes se comunican con el cliente utilizando agentes de comunicaciones (**ServerCommProtocol**) y llevan a cabo aquellas tareas que

han de efectuarse en el servidor. Usan **FileHandlers** para gestionar los ficheros de cada sesión (ficheros de código fuente, clases compiladas, etc) y pueden compilar código bajo demanda del usuario.

- Una colección de compiladores compartida por todos los **ServerReqProcessors**. Esta incluye un **JavaCompiler** y un **LabCommCompiler**, implementados como monitores[14] para asegurar que son usados en exclusión mutua. Como esta versión inicial del simulador sólo puede trabajar con productores y consumidores escritos en Java, añadir más compiladores queda como una posible mejora futura para este simulador (ver Sección 5.3.1).

2.4. Comunicación cliente-servidor

La Figura 2.11 ofrece una visión general de los agentes que conforman el cliente y el servidor y de la comunicación que mantienen entre ellos.

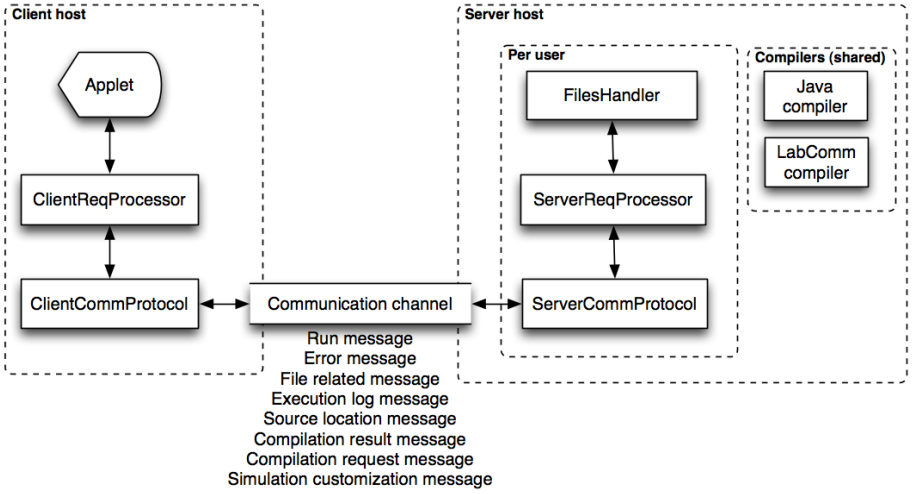

Figura 2.11: Estructura de los subsistemas cliente y servidor y de su comunicación basada en paso de mensajes.

Los mensajes intercambiados por el cliente y el servidor pueden clasificarse de acuerdo con su función:

- Obtener ficheros del servidor y remitirlos de vuelta.
- Mensajes relacionados con el proceso de compilación. Existen varios tipos, dependiendo de su propósito: emitir una orden de compilación

al servidor o transmitir al cliente un informe de la compilación (ya sea confirmando una compilación exitosa o listando errores de compilación encontrados).

- Mantener al servidor informado de las opciones de configuración elegidas por el usuario (e.g. si el usuario decide que el productor ha de estar implementado en C#, cuando quiera que el usuario solicite el código fuente del productor para verlo/modificarlo, este debería recibir una plantilla de productor en C# en vez de una en Java o C).

- Determinar la localización de aquellas clases que han de ser cargadas por el cliente para poder ejecutar la simulación. Más información en la Sección 2.5.2.

- Solicitar la ejecución de la simulación (total o parcialmente) en el servidor y obtener como resultado un **ExecutionLog**[1].

Estos mensajes están implementados como clases Java, cada una de las cuales define un único tipo de mensaje. En tiempo de ejecución, instancias de estas clases son intercambiadas entre el cliente y el servidor utilizando ObjectStreams[15], que son flujos de transmisión apropiados para enviar y recibir objetos. Antes de ser transmitido a través de un ObjectStream, un objeto ha de ser serializado (convertido en una secuencia de bytes) y, una vez recibido, ha de ser deserializado (recuperado a partir de esa secuencia). Esto se puede conseguir de forma transparente al programador si se implementa la interfaz Serializable[2][16][17] en aquellas clases Java que definen los mensajes.

[1] **ExecutionLog** es una clase Java diseñada para contener un registro (*log*) de todos los eventos de codificación/decodificación de mensajes utilizando LabComm que han ocurrido durante la transmisión, junto con un *timestamp* para cada uno de ellos.

[2] Un lector atento puede estar confuso: si la meta es comunicar al productor y al consumidor al tiempo que evitamos invertir mucho tiempo en crear el protocolo de comunicación, ¿por qué emplear la solución presentada (implementar mensajes como clases Java serializables y enviarlos a través de ObjectStreams) si LabComm podría ser usado? Esta es una muy buena crítica y la única respuesta es que, en el momento de acometer este proyecto, el autor tenía una experiencia práctica muy limitada con LabComm (¡recordemos que este simulador no existía por aquel entonces!) y no era por tanto plenamente consciente de todas las posibilidades que ofrece. Ahora, si el autor se viera enfrentado a la misma tarea (o a una similar), él optaría directamente por una implementación basada en LabComm, pues desarrollar este simulador le ha dado experiencia con LabComm. Ahora él sabe que LabComm permitiría un desarrollo mucho más rápido que escribir una implementación completa en Java, porque aunque en ambos casos el programador está obligado a definir las estructuras de datos para los mensajes, en Java también está obligado a escribir el código que gestiona el envío y recepción de mensajes mientras que LabComm hace esto por el programador.

Si definimos una superclase abstracta para esos mensajes que implemente la interfaz Serializable, podremos definir toda una jerarquía de clases de mensajes con diferentes propósitos y todas ellas serán serializables.
Como los Arrays[18] de Java son también objetos, esta aplicación implementa la transferencia de ficheros a partir de la transmisión de Arrays de *bytes*. Es así como se transmiten los ficheros de código entre el cliente y el servidor. La clase utilizada para enviar y recibir ficheros se llama **FileTransfer** y su código está disponible en el Apéndice A.1.

Es muy conveniente implementar los mensajes como clases, pues ello permite enviar mensajes muy complejos sin necesidad de diseñar un intrincado protocolo de comunicación. Además, reconocer los mensajes entrantes es muy sencillo, pues el operador **instanceof**[19] de Java puede ser usado para determinar la clase del mensaje que hemos recibido y así interpretarlo correctamente.

2.5. Ejecución de simulaciones

2.5.1. Aislamiento de otros usuarios

En circunstancias normales, se espera que múltiples usuarios interaccionen con la aplicación, ya sea enviando ficheros modificados al servidor, solicitando compilaciones y ejecuciones de simulaciones, etc.
Hemos de garantizar que las acciones de cada usuario no afecten en modo alguno a los demás usuarios, pues si no estos experimentarán un comportamiento inesperado y errático por parte de la aplicación. En otras palabras, cada usuario debería tener la impresión de que es el único que está trabajando con el simulador. Para ello, impondremos acceso en exclusión mutua a todos los recursos compartidos (e.g. los compiladores mencionados en la Sección 2.3 están implementados como monitores[14]) y nos aseguraremos de que cada usuario pueda acceder únicamente a los ficheros pertenecientes a su sesión.
Al inicio de cada sesión, se creará un subdirectorio para cada usuario bajo los directorios de ficheros de código fuente y binarios localizados en el directorio raíz del servidor web. Cada uno de estos nuevos directorios recibirá un nombre basado en un identificador de sesión único, para garantizar la segura identificación de los ficheros de cada usuario. El directorio de ficheros de código fuente será inicialmente provisto de una copia de todos los ficheros de código usados en la simulación (protocolo, productor y consumidor). Siempre que el usuario modifique uno de estos, la versión enviada al servidor

será emplazada en su directorio de ficheros fuente y siempre que el usuario solicite una compilación, los ficheros resultantes se colocarán en su directorio de binarios. Naturalmente, estos directorios son debidamente borrados al término de cada sesión. La Figura 2.12 muestra una estructura de directorios simplificada para ilustrar dónde se colocarían los ficheros en una sesión con identificador 2.

```
.
|-- bin
|   |-- clientFiles
|   |   `-- dir2
|   |       |-- labCommRoutines
|   |       `-- producerConsumer
|   |-- clientSide
|   |-- common
|   |-- messages
|   |-- serverSide
|-- src
    |-- clientFiles
    |   `-- dir2
    |       |-- labCommRoutines
    |       |-- producerConsumer
    |       `-- protocol.lc
    |-- clientSide
    |-- common
    |-- messages
    |-- serverSide
```

Figura 2.12: Estructura de directorios simplificada para mostrar dónde colocar los ficheros de una sesión con identificador 2. Todos los elementos del directorio de ficheros fuente tienen su equivalente en el directorio de binarios como resultado del proceso de compilación. La única excepción es el protocolo LabComm (.lc), pero esto es así porque la definición del protocolo no es de ninguna utilidad en el directorio de binarios. Lo que necesitamos son las rutinas de serialización obtenidas al compilar el protocolo, que serán colocadas en el directorio "./bin/clientFiles/dir2/labCommRoutines".

2.5.2. Un marco para la ejecución de simulaciones

Idealmente, un marco para la ejecución de simulaciones debería simplificar las tareas de inicializar el productor y el consumidor, configurarlos para que se comuniquen con independencia del equipo en que se estén ejecutando (ya sea el cliente o el servidor), llevar a cabo la simulación y devolver sus resultados (en forma de uno o dos ExecutionLogs). Para esto, la clase **ExecutionHandler** ha sido escrita. El uso de esta clase es realmente sencillo, dado que sólo requiere de cierta información de inicialización que le permita autoconfigurarse y, hecho esto, se le puede pedir que ejecute simulaciones. La Figura 2.13 muestra cómo se usa esta clase.

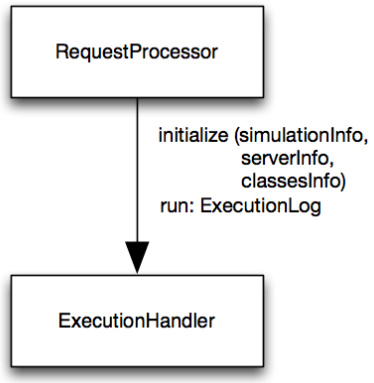

Figura 2.13: Un ClientReqProcessor o un ServerReqProcessor configura un ExecutionHandler, lo utiliza para ejecutar una simulación y obtiene un ExecutionLog (o dos) como resultado.

Un ExecutionHandler requiere de la siguiente información para autoconfigurarse:

- Configuración de la simulación: computadoras (cliente o servidor) en que se van a ejecutar el productor y el consumidor.
- Información para establecer la conexión: nombre del servidor y puerto en que la aplicación en el servidor (ya sea una aplicación productora o una consumidora) estará escuchando. Como un ExecutionHandler puede ser creado por un ClientReqProcessor o por un ServerReqProcessor, es necesario informar al ExecutionHandler de si ha sido creado en el cliente o en el servidor. Esta información es necesaria a la hora de conectar el productor y el consumidor para ejecutar la simulación y para devolver los ExecutionLogs al cliente.

- Información para cargar las clases (i.e. URLs[20]) de las clases que implementan el productor, el consumidor o ambos).

Una vez inicializado, el ExecutionHandler determinará qué clases (productor y/o consumidor más las rutinas de serialización necesarias) deben ser cargadas y si esta carga ha de hacerse de forma local o remota (desde el servidor). En cualquier caso, un cargador de clases Java (Java ClassLoader[21]) será usado para esta tarea. Una vez el productor, el consumidor o ambos hayan sido cargados, el ExecutionHandler los configurará para que sepan cómo conectarse el uno al otro en función del equipo en que se vaya a ejecutar cada uno de ellos. Por supuesto, tanto el productor como el consumidor son ejecutados en *threads* independientes durante la simulación.
Al término de la simulación el resultado de la misma, en forma de uno o dos ExecutionLogs, es devuelto a la clase que configuró al ExecutionHandler para ejecutar la simulación.

Las Figuras 2.14, 2.15 y 2.16 ilustran cómo proceden los ExecutionHandlers ante cada posible escenario de simulación. En todos ellos, las simulaciones comienzan bajo demanda del usuario y terminan al entregar los resultados de la simulación al *applet* para que los muestre al usuario. Los pasos intermedios varían en todos los casos, como puede verse en las Figuras.

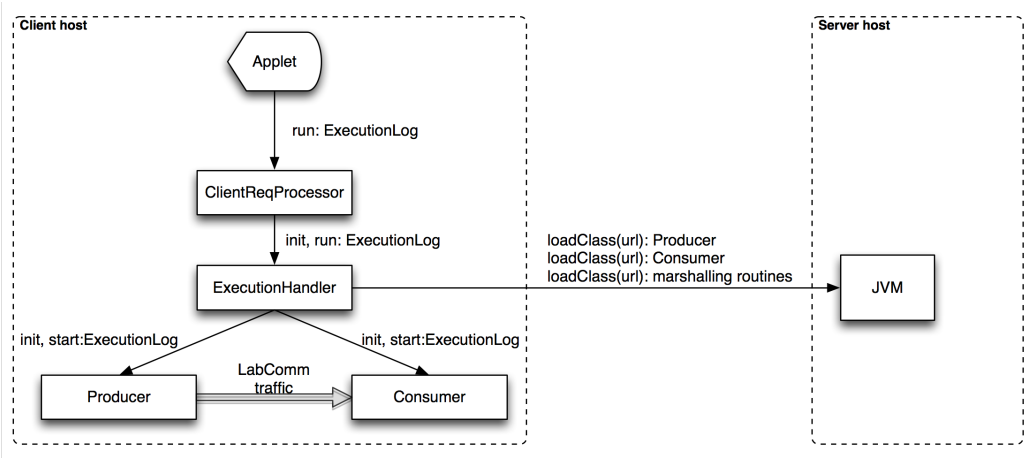

Figura 2.14: Un ExecutionHandler lleva a cabo una simulación en el cliente. Ello requiere cargar el productor y el consumidor desde el servidor. Los resultados son entregados directamente al ClientReqProcessor, que los remitirá al *applet*.

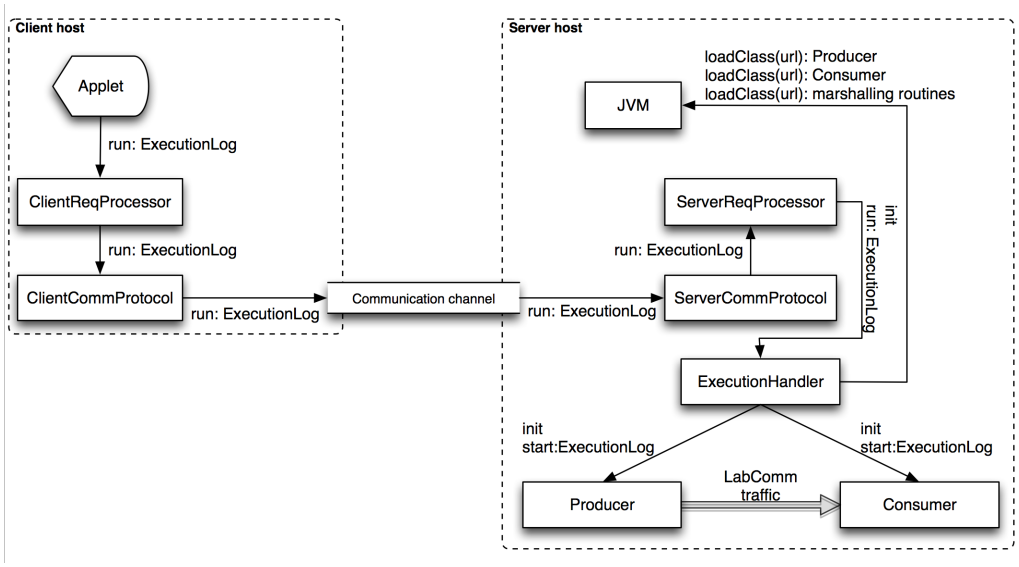

Figura 2.15: Un ExecutionHandler ejecuta toda la simulación en el servidor. La carga de clases se efectúa de forma local. Cuando la simulación termina y el ServerReqProcessor obtiene los ExecutionLogs, utiliza el ServerCommProtocol para enviarlos al cliente. Una vez recibidos en el cliente, se le entregan al *applet* para que los muestre en la interfaz gráfica de usuario.

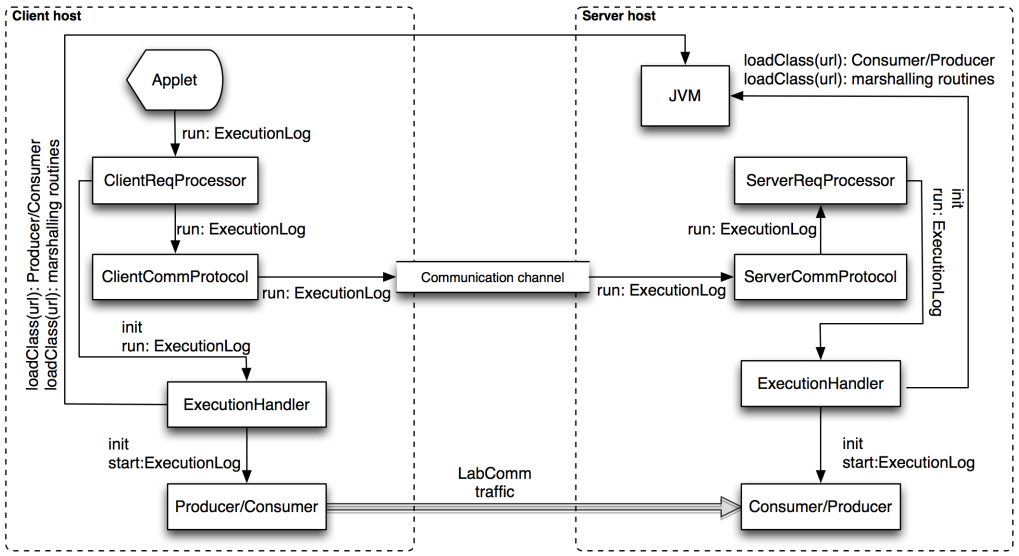

Figura 2.16: Dos ExecutionHandlers, uno sito en el cliente y otro en el servidor, colaboran para llevar a cabo la simulación. En este caso, parte de los resultados será generada en el cliente y parte vendrá del servidor.

2.5.3. Comunicación productor-consumidor

El productor y el consumidor utilizan un flujo de datos para transmitir los mensajes codificados con LabComm. Como el productor y el consumidor pueden ejecutarse en la misma máquina o en máquinas distintas, los comunicaremos usando un *socket*, pues estos son válidos en ambas situaciones. Sin embargo, establecer conexiones empleando *sockets* puede ser complicado a causa de todas las posibles localizaciones del productor y del consumidor y de que la única dirección fija y conocida es la del servidor. Si empleamos una aproximación cliente-servidor para conectar al productor y al consumidor, podemos forzar a que la aplicación que se ejecute en el servidor actúe como un servidor y a que la aplicación que se ejecute fuera del servidor haga el papel de cliente. Si tanto el productor como el consumidor se ejecutan en el servidor, la decisión de cuál toma el papel de servidor sería arbitraria. Como será explicado más a fondo en la Sección 2.6.1, no se pueden llevar a cabo simulaciones en modo cliente-cliente en esta versión del simulador, por lo que este esquema solucionaría el problema de conectar al productor y al consumidor.

La implementación es la siguiente: el productor y el consumidor desconocen su papel en la comunicación (cliente/servidor) y se conectan el uno al otro a través de una interfaz Java llamada **CommunicationSupport**. Esta interfaz define dos métodos: *connect()* y *getSocket()*. Dos clases implementan esta interfaz:

- **ClientSupport** toma el papel de cliente en la conexión. Cuando su método *connect()* es invocado, este intenta conectar a un puerto específico de un computador determinado.
- **ServerSupport** actúa como un servidor. Cuando su método *connect()* es invocado, comienza a escuchar en un puerto, esperando peticiones de conexión.

Cuando un ExecutionHandler va a inicializar a un productor/consumidor, evalúa si ese agente tomará el papel de cliente o de servidor. Dependiendo de esto, creará un objeto ClientSupport o un objeto ServerSupport y se lo pasará al constructor del agente. Tanto productores como consumidores acceden al objeto ClientSupport/ServerSupport a través de la interfaz CommunicationSupport y por tanto no son conscientes de cómo se establece la conexión. Estos simplemente invocan el método *connect()* de la interfaz y esperan a que termine. Cuando eso ocurre, una conexión ha sido establecida entre el productor y el consumidor. Entonces, utilizarán el método *getSocket()* de

CommunicationSupport para obtener el *socket* que los conecta y lo utilizarán para enviar/recibir los datos codificados con LabComm.

2.6. Detalles técnicos de la solución

Determinadas características de diseño del lenguaje Java han impuesto restricciones que han tenido un impacto en la implementación y funcionalidad del simulador. Estas son:

2.6.1. *Applets* no firmados

Uno de los motivos de que los *applets* sean tan usados es que no entrañan riesgos de seguridad. Ello es así porque, o bien el *applet* ha sido revisado y señalado como inocuo por una autoridad certificadora (*applets* firmados/fiables) o bien el *applet* es ejecutado bajo severas restricciones de seguridad (*applets* no firmados/no fiables). El *applet* usado por el simulador no está firmado y por tanto está afectado por estas restricciones de seguridad. Una de dichas restricciones impide al *applet* establecer conexiones a otros ordenadores que no sean aquel del que el *applet* se ha descargado. Ello **imposibilita ejecutar el simulador en modo cliente-cliente**, pues no es posible establecer conexiones dentro del cliente. Sin embargo, esto no supone un gran inconveniente, porque el motivo de querer usar todos estos modos (cliente-cliente, servidor-cliente, etc) era demostrar que LabComm puede trabajar sobre cualquier tipo de flujo, al margen de si este flujo es local a la computadora o entre diferentes computadoras, y ello ha quedado demostrado al no ocasionar LabComm fallo alguno en ninguno de los otros modos de trabajo del simulador.

2.6.2. Recarga dinámica de clases

Para poder ejecutar simulaciones cuando el usuario lo solicite, las clases que intervienen en la simulación han de ser cargadas de forma dinámica. Los ClassLoaders[21] de Java facilitan esta tarea, pero desconocer cómo un ClassLoader funciona puede traer consigo efectos inesperados; cuando un ClassLoader recibe una petición de carga de una clase, primero comprueba si la clase ha sido previamente cargada o si alguno de los padres del ClassLoader (hay una jerarquía parental) puede cargarla. Sólo si ninguna de estas opciones es posible intentará el ClassLoader cargar la clase. Esto es problemático, pues se espera que los usuarios del simulador sigan un proceso iterativo de

modificar, compilar y ejecutar el código de la simulación hasta que obtengan los resultados deseados. Si ignoráramos cómo trabajan los ClassLoaders, cuando un usuario intentara compilar y ejecutar una simulación más de una vez se encontraría con que el agente modificado (ya sea el productor o el consumidor) seguiría mostrando el mismo comportamiento que antes, pues la última versión de esa clase no habría sido cargada por la Máquina Virtual de Java (JVM).

La solución inmediata a este problema sería Java Reflection[22][23], pero esta no es posible a causa de las restricciones de seguridad aplicadas a los *applets* no firmados. Estos deben siempre utilizar su propio AppletClassLoader (una subclase de ClassLoader) y no pueden instanciar ningún otro ClassLoader, lo que hace que esta solución no sea aplicable a nuestro problema. Sin embargo, podemos evitar el problema si **versionamos el productor y el consumidor**; cada vez que el usuario modifique uno de estos, su número de versión será actualizado por el servidor. Si concatenamos este número al nombre de la clase, conseguiremos que la JVM cargue cualquier versión modificada del productor y del consumidor y conseguiremos que la simulación transcurra de acuerdo con las expectativas del usuario.

Capítulo 3

ThrottleNet: comunicación en red en tiempo real

Como se ha explicado anteriormente, la fiabilidad de una red ThrottleNet depende de un uso controlado del ancho de banda disponible en la red. Es por ello que los nodos ThrottleNet no emplean directamente su enlace físico al conmutador para comunicarse con los demás nodos. En su lugar, establecen conexiones lógicas[1] unidireccionales hacia aquellos nodos a los que desean enviar datos. Estas conexiones lógicas transmiten datos a través de los enlaces físicos que conectan los nodos al conmutador (ver Figura 3.1) y tienen un ancho de banda asignado. Por tanto, la carga de tráfico en cada enlace físico puede ser calculada como la suma de todos los anchos de banda asignados a las conexiones que transmiten datos a través del enlace. La red evita situaciones que conduzcan a un uso excesivo del ancho de banda denegando arbitrariamente aquellas peticiones de establecimiento de conexiones que requieran más ancho de banda que el que hay disponible en el enlace físico sobre el que pretenden transmitir (ver Sección 3.3.2).

Ahora es momento de considerar cómo se transmiten datos a través de las conexiones lógicas. Si utilizásemos TCP/UDP y/o IP, el protocolo ARP[24] sería empleado para determinar la dirección física de los receptores de los mensajes. Esto puede plantear problemas, dado que ARP genera mensajes multidifusión (*broadcast*) que consumen grandes cantidades de ancho de banda y pueden incrementar notablemente el tiempo de entrega de los mensajes. Dada la importancia de cumplir con los plazos de entrega, ThrottleNet no utilizará TCP/UDP ni IP para transmitir datos, sino que trabajará directa-

[1] Utilizamos el término "conexión lógica" en vez de "servicio" para reflejar que estas conexiones no transmiten datos codificados utilizando LabComm.

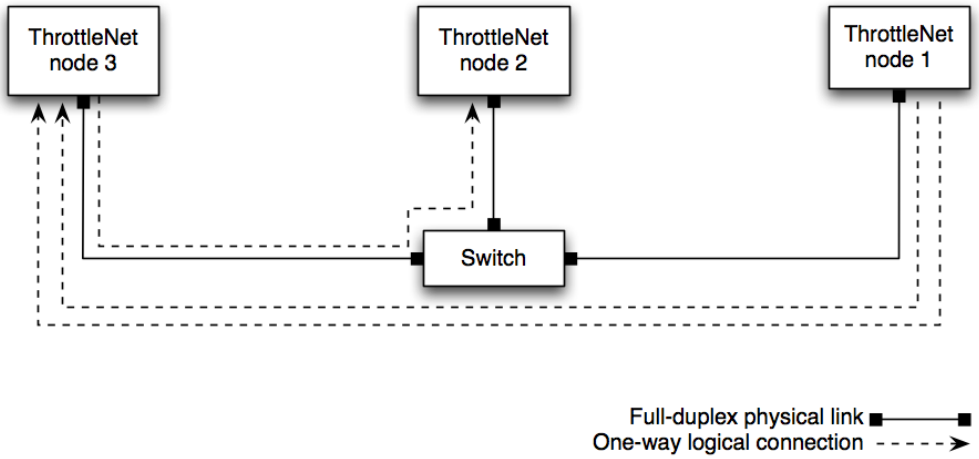

Figura 3.1: Topología en una red ThrottleNet. Los tres nodos están conectados mediante un enlace *full-duplex* al conmutador y algunos de ellos han establecido conexiones lógicas unidireccionales sobre los enlaces físicos. Es interesante notar que un nodo puede establecer dos conexiones lógicas hacia otro nodo, si es que necesita enviarle datos de diferente naturaleza o con diferentes necesidades temporales de entrega.

mente sobre Ethernet.

Este apartado consta de cinco secciones que tratan los aspectos más relevantes del diseño y la implementación elegidos:

- La Sección 3.1 describe cómo ThrottleNet emplea el *driver* Ethernet del sistema operativo para enviar y recibir tramas Ethernet.
- En la Sección 3.2 se explica cómo se lleva a cabo el racionamiento de tráfico.
- En la Sección 3.3 se describe a GlobeThrottle, que es el agente gestor del ancho de banda y quien proporciona los mecanismos de resolución de direcciones en una red ThrottleNet.
- La Sección 3.4 detalla los diferentes tipos de tráfico que pueden encontrarse en la red.
- Por último, la Sección 3.5 describe el tratamiento que la red ThrottleNet da a ARP[1] para evitar problemas en situaciones de alta carga.

[1]Nos referimos ahora al caso en que ARP es encapsulado como tráfico ordinario o NRT. Para más detalles, consultar la Sección 3.3.

Finalmente, puede ser interesante estudiar las Secciones 3.4.1 y 3.3.3, que detallan los mecanismos de resolución de direcciones para tráfico RT y NRT.

3.1. Implementación como *driver*

ThrottleNet será diseñado como un *driver* Linux que será instalado en cada nodo de la red para racionar su tráfico saliente. Aunque hayamos decidido trabajar directamente sobre Ethernet, sería interesante que la implementación de ThrottleNet conociera tan poco como fuera posible de los procesos de transmisión de tramas en Ethernet, a fin de mejorar la claridad y portabilidad de la implementación. Por ello, relegaremos las tareas de envío y recepción de tramas Ethernet al *driver* Ethernet del sistema operativo, que será quien interaccione con la tarjeta Ethernet de la computadora para enviar y recibir datos a través de la red.
Podemos considerar ThrottleNet como una capa intermedia entre el usuario y la capa de enlace de datos de la pila de protocolos:

- La interfaz del lado del usuario consiste en dos interfaces de red de Linux: **eth_rt** y **eth_nrt**. Estas interfaces permiten al usuario transmitir mensajes RT y NRT hacia los demás nodos.

- La interfaz dirigida hacia la capa de enlace de datos permite comunicar directamente con el *driver* Ethernet del sistema operativo y usarlo para enviar y recibir datos a través de la red. Esta interfaz se llama **eth_bridge**, pues sirve como un puente hacia el *driver* de la tarjeta Ethernet.

La interfaz **eth_bridge** y la interfaz del *driver* Ethernet (e.g. eth0) están conectadas por medio de un puente Linux. Un puente Linux[25] es un módulo que transmite tráfico transparentemente entre múltiples interfaces de red con independencia de los protocolos empleados en esas redes. Así, un puente Linux conecta dos segmentos Ethernet físicos para formar un segmento Ethernet lógico más grande. La Figura 3.2 ofrece una visión general de las interfaces del *driver* ThrottleNet e ilustra cómo ThrottleNet actúa de manera similar a una capa que separa al usuario de la capa de enlace de datos y cómo esta última se conecta a la capa física.

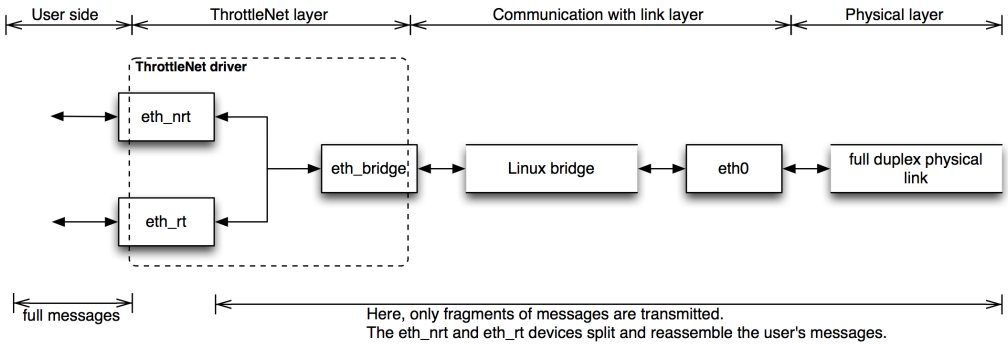

Figura 3.2: Visión general de las interfaces ofrecidas por el *driver* ThrottleNet. Esta Figura ilustra cómo ThrottleNet utiliza un puente Linux para acceder a la capa de enlace de datos y cómo el usuario no es consciente del racionamiento de tráfico que ThrottleNet efectúa, ya que este sólo envía y recibe mensajes completos.

3.2. Racionamiento de tráfico

La cantidad de ancho de banda asignada a cada conexión lógica queda especificada por la máxima longitud permitida para cada mensaje enviado a través de la conexión y por el mínimo período de espera obligatorio entre dos transmisiones de mensajes consecutivas: un nodo que envía datos utilizando una determinada conexión dispone de una ranura de transmisión de x octetos de longitud cada s nanosegundos. Se espera que los nodos intenten enviar mensajes demasiado grandes para encajar en estas ranuras, por tanto habrá que dividir esos mensajes en fragmentos de una longitud apropiada antes de transmitirlos (y planificar el envío de cada fragmento dentro de alguna de las ranuras de transmisión disponibles). Una vez el receptor haya recibido todos los fragmentos de un mensaje, los ensamblará para obtener el mensaje original y entregará este a las aplicaciones que lo esperen.

Así, el racionamiento de tráfico en una red ThrottleNet implica la realización de una serie de tareas de bajo nivel: fragmentación y reensamblado de mensajes, planificación de la transmisión de fragmentos de mensajes, etc. Estas tareas se ocultan mediante el uso de **colas de entrada** y **colas de salida** para procesar el tráfico entrante y saliente, respectivamente. Estas colas son tipos abstractos de datos esenciales en una red ThrottleNet, pues son necesarias para poder interactuar con las conexiones lógicas entre los nodos. Los puntos de entrada y salida de estas conexiones lógicas son, precisamente, estas colas de procesamiento de tráfico, que necesitan ser asignadas para

servir exclusivamente a una conexión. Esto es, si un agente[1] en la red produce datos para una conexión lógica y consume datos de otras dos conexiones lógicas, deberá usar una cola de salida y dos colas de entrada, cada una de las cuales dedicada exclusivamente a una de las conexiones mencionadas.

3.2.1. Tráfico saliente

Si un agente en una red ThrottleNet quiere enviar datos a través de una conexión lógica, lo hará utilizando la cola de salida que tenga asignada a esa conexión. Entre otros recursos, una cola de salida consta de:

- Un marcador de uso que indica si la cola está siendo usada, es decir, si ya hay en ella mensajes (o fragmentos de mensajes) pendientes de ser transmitidos.
- Una cola de fragmentos en la que están los fragmentos en que un mensaje ha sido dividido para su transmisión. Estos fragmentos son de tamaño igual o inferior al máximo tamaño de mensajes permitido para la conexión y se envían respetando el mínimo período de espera definido para la conexión.
- Una cola de mensajes, aún sin fragmentar, esperando a ser fragmentados y enviados.
- Un temporizador que se activa periódicamente para transmitir un fragmento o, en ausencia de fragmentos, fragmentar un mensaje pendiente de enviar.
- Medios para conseguir acceso en exclusión mutua a la propia cola de salida. Según la cola haya sido definida en espacio de usuario o de núcleo, se utilizará un semáforo[26] o un *spinlock*[27][13].

Cuando un agente trata de enviar un mensaje utilizando una cola de salida, pueden ocurrir dos cosas: si la cola está disponible, se la marca como ya no disponible, se fragmenta el mensaje y se inicia un temporizador para empezar la transmisión periódica de los fragmentos. Por otro lado, si la cola está siendo usada, el mensaje es puesto al final de una cola de mensajes. Periódicamente un temporizador se dispara para iniciar la transmisión de un fragmento o, si no hay ninguno, para sacar un mensaje de la cola de mensajes, fragmentarlo y colocar sus fragmentos en la cola de fragmentos. Cuando

[1] En una red ThrottleNet, tanto los nodos ThrottleNet como el GlobeThrottle pueden ser agentes.

el temporizador se vuelva a disparar, será uno de esos fragmentos el que se transmitirá. Finalmente, si no hay fragmentos por transmitir ni mensajes por fragmentar, el temporizador se desactiva y la cola se marca como disponible. El Apéndice B.1.1 contiene fragmentos de pseudocódigo que ilustran el procesamiento de tráfico saliente con mayor detalle.

3.2.2. Tráfico entrante

Cada vez que un mensaje[1] llega a través de una conexión lógica, es procesado utilizando la cola de entrada asignada a esa conexión. Entre otros recursos, las colas de entrada constan de:

- Una cola en la que se guardan los fragmentos recibidos. Cuando se dispone de todos los fragmentos de un mensaje, estos se reensamblan para formar el mensaje original.
- Un *buffer* que almacena el último mensaje reensamblado, para que pueda ser recogido por la aplicación que lo esperaba.

Todos los fragmentos de mensaje están numerados secuencialmente $(0, \ldots, n)$ e incluyen el número de fragmentos en que se dividió el mensaje original (n). Como los fragmentos se envían en orden $(0, 1, 2, \ldots, n)$, respetando los períodos entre transmisiones de mensajes y los recursos físicos de la red no son sobrecargados, la lógica que controla el tratamiento de mensajes entrantes asume que los mensajes llegan en orden y que, si un fragmento falta, ha sido perdido y no llegará en el futuro. Así, cuando un fragmento llega a la cola, su número de fragmento y número total de fragmentos son examinados para ver si se trata del fragmento que esperábamos (e.g. si el fragmento que llegó inmediatamente antes era el número i de una secuencia de n fragmentos, este debería ser el número $i + 1$ de una secuencia de n fragmentos; suponiendo, claro, que $i + 1 <= n$). Si algún número de la secuencia $0, \ldots, n$ falta, descartaremos todos los fragmentos del mensaje puesto que uno de los fragmentos falta (y no llegará) y por tanto no podremos reensamblar el mensaje.
El Apéndice B.1.2 contiene fragmentos de pseudocódigo que ofrecen una visión más detallada de la lógica que controla la recepción de los fragmentos.

El siguiente ejemplo muestra un caso de error que resulta indetectable para el algoritmo usado: sean M_1 y M_2 dos mensajes de la misma longitud

[1] Todo mensaje enviado en una red ThrottleNet no es un mensaje en sí mismo sino un fragmento de otro mensaje (posiblemente más grande). Sin embargo, las palabras mensaje y fragmento serán empleadas sin distinción para referirnos a estos fragmentos en tránsito.

que van a ser enviados uno tras otro utilizando una determinada conexión lógica. Cada uno de ellos será dividido en n fragmentos ($M_1 = m_{11}, \ldots, m_{1n}$ y $M_2 = m_{21}, \ldots, m_{2n}$). Supongamos que un fallo de conexión ocurre durante la transmisión y que sólo los fragmentos m_{11} a m_{1i} y m_{2i+1} a m_{2n} ($1 < i < n$) llegan. Esta secuencia sería válida desde el punto de vista del algoritmo de control y el error nunca sería detectado (salvo por un comportamiento inapropiado de la aplicación que consuma dicho mensaje).
Sin embargo, dadas la fiabilidad de Ethernet y la ausencia de colisiones en una red ThrottleNet, podemos asumir este caso como muy improbable.

3.3. GlobeThrottle

Gestionar el ancho de banda en una red ThrottleNet requiere tomar decisiones acerca de si cada nueva petición de establecimiento de conexión es factible o no en términos de disponibilidad de ancho de banda. Estas decisiones se pueden efectuar de manera centralizada o alcanzando acuerdos entre todos los nodos de la red[28]. Optaremos por crear un agente gestor de ancho de banda centralizado (GlobeThrottle) al que los nodos de la red dirigirán sus peticiones de permiso para establecer conexiones lógicas hacia los demás nodos.

GlobeThrottle tiene otras tareas aparte de la gestión de ancho de banda: como la primera tarea de cada nodo recién conectado a la red es informar a GlobeThrottle de su existencia, GlobeThrottle tiene una visión completa del estado de la red (i.e. nodos registrados en la red y conexiones activas entre esos nodos). GlobeThrottle usará este conocimiento para sustentar tráfico NRT de los usuarios y para proveer de mecanismos de resolución de direcciones para tráfico RT y NRT (Secciones 3.3.3 y 3.4.1).

3.3.1. Conexión a la red ThrottleNet

Como una red ThrottleNet trabaja directamente sobre Ethernet, en el caso del *driver* ThrottleNet usamos las interfaces que este definía para comunicar con el *driver* Ethernet y así ser capaces de conectar el nodo a la red. Sin embargo, las aplicaciones de espacio de usuario (como GlobeThrottle) no son capaces de definir una interfaz de red. Utilizaremos el *driver* universal TUN/TAP[29][30] para crear una interfaz TAP. Una interfaz TAP es una interfaz de red virtual que simula un dispositivo Ethernet y puede ser acoplada a un puente Linux, exactamente como ya hicimos antes. GlobeThrottle abrirá la interfaz utilizando una llamada **ioctl**[31] y leerá y escribirá de ella

empleando las llamadas al sistema *read* y *write*. En este caso, "leer" significa obtener mensajes de la red, mientras que "escribir" significa enviar mensajes a través de la red.
La Figura 3.3 ilustra cómo se hace esto.

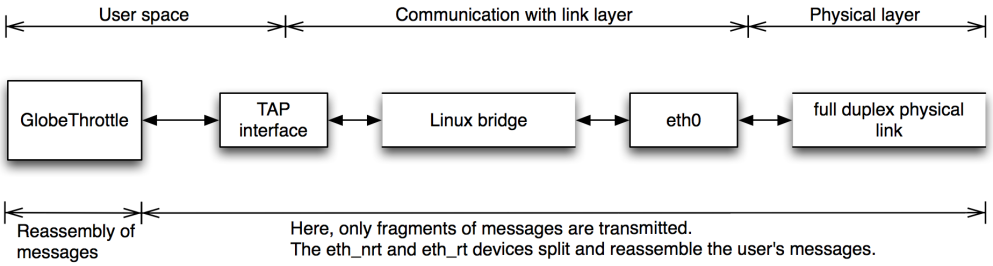

Figura 3.3: Conexión de GlobeThrottle a la red ThrottleNet. Aunque un *driver* ThrottleNet también podría ser acoplado al puente Linux, esto no se muestra en esta imagen por motivos de claridad.

3.3.2. Gestión del ancho de banda

Una conexión lógica en una red ThrottleNet implica a un emisor y a dos o más receptores. Estas conexiones no deben usar en exceso los tres elementos físicos de la red que los sustentan, que son los enlaces físicos de los nodos emisor y receptores al conmutador y los *bufferes* de los puertos de salida del conmutador que dirigen tráfico a los receptores. Es decir, validar una nueva conexión supone comprobar que:

- Hay suficiente ancho de banda disponible para enviar tráfico sobre los enlaces físicos afectados.
- Ninguno de los *bufferes* de salida del conmutador asignados a los nodos receptores se verá sobrecargado en caso de que llegaran, simultáneamente, mensajes de todas las conexiones a las que esos nodos están suscritos como receptores de datos.

Ejemplos:

- Aquellas conexiones que envíen datos con elevada frecuencia serán probablemente rechazadas, puesto que sus requisitos de ancho de banda serán altos incluso si envían mensajes de pequeño tamaño.

- Las conexiones que envíen mensajes grandes a baja frecuencia probablemente pasarán el control de ancho de banda. Sin embargo, si un nodo está suscrito a varias de estas conexiones como receptor de datos y una nueva petición de conexión llega, deberá verificarse que una llegada simultánea de mensajes de todas esas conexiones no sobrecargará el *buffer* de salida del conmutador dedicado al nodo en cuestión.

Cuando un nodo ThrottleNet desea establecer una conexión ha de solicitar permiso a GlobeThrottle. GlobeThrottle mantiene un registro de todas las conexiones lógicas establecidas y efectúa cálculos para determinar si aceptar la nueva conexión conduciría a un uso abusivo de los segmentos físicos de la red mencionados.
Una conexión lógica en una red ThrottleNet únicamente genera tráfico cuando los dos extremos de la comunicación están definidos: en ausencia de emisor, el receptor nunca recibirá datos de la conexión, mientras que si sólo hay emisor, este nunca enviará datos a través de la conexión, pues no habrá nadie esperando a consumirlos (más detalles en la Sección 3.4.1). Como una conexión puede tener más de un sumidero[1], es necesario verificar que ninguno de los segmentos físicos de red utilizados por los nodos que participan en la conexión será utilizado en exceso como consecuencia de la aceptación de dicha conexión.

El proceso seguido por GlobeThrottle para decidir si las conexiones son aceptadas o no está ilustrado en el Apéndice B.2. Si GlobeThrottle estima que aceptar la conexión no hará peligrar la integridad de la red, la conexión será aceptada y GlobeThrottle actualizará su base de datos de conexiones. Cuando esto ocurre, se muestra un informe en pantalla (ver Figura 3.4).

```
Alive nodes: 06:06:00:00:01:02, 06:06:00:00:02:02, 06:06:00:00:03:02
-------------------------------------------------------------------------------------------------------
| ID  | Service source    | Service sinks     | Period (ns) | Size (B)| Service description           |
-------------------------------------------------------------------------------------------------------
|   7 | 06:06:00:00:01:03 |                   |      100000 |      70 | pressure sensor               |
-------------------------------------------------------------------------------------------------------
|  19 | 06:06:00:00:01:03 | 06:06:00:00:03:03 |   800000000 |     700 | heat sensor                   |
|     |                   | 06:06:00:00:02:03 |             |         |                               |
-------------------------------------------------------------------------------------------------------
| 213 |                   | 06:06:00:00:02:03 |        1700 |     400 |                               |
-------------------------------------------------------------------------------------------------------
```

Figura 3.4: Posible informe mostrado por GlobeThrottle, detallando los servicios activos y un listado de los nodos en la red.

[1]Un sumidero es un nodo que consume datos de una conexión mientras que una fuente es un nodo que produce datos y los envía a través de una conexión.

3.3.3. Organizador de tráfico NRT

Además de tráfico de tiempo real, sería deseable permitir a los usuarios utilizar la red con fines que no requieran de una transmisión de datos "urgente" (e.g. establecer una conexión SSH a un servidor, emplear la utilidad *ping* para comprobar si se puede alcanzar a un computador, etc). Todos los nodos ThrottleNet cuentan con una conexión lógica por defecto, la conexión NRT. Esta conexión dispone de ranuras de transmisión de 60 *bytes* de longitud cada milisegundo que se utilizan para enviar el tráfico que no tiene necesidades de tiempo real (NRT) del nodo. Encapsularemos el tráfico de las capas superiores de la pila de protocolos como tráfico NRT y lo transmitiremos a través de esta conexión. Así, podremos utilizar TCP/UDP y/o IP (entre otros) en redes ThrottleNet. La Figura 3.5 muestra un escenario en que un nodo está suscrito a tres conexiones lógicas de tiempo real y envía el tráfico de las capas de Internet, Transporte y Aplicación a través de la conexión NRT.

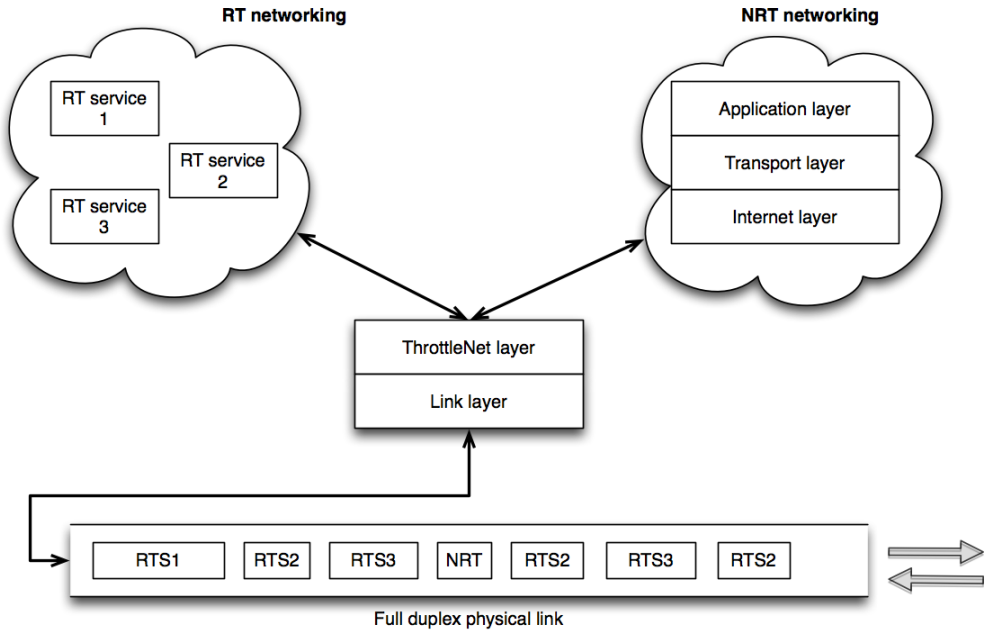

Figura 3.5: ThrottleNet permite tráfico RT y NRT. El tráfico de cada una de las redes se encapsula empleando, o bien la conexión lógica de tiempo real correspondiente, o bien la conexión NRT. Puede apreciarse como todas estas conexiones tienen diferentes tamaños de paquete y períodos de espera entre transmisiones si se examinan los fragmentos enviados a través del enlace físico.

Ahora que las capas superiores de la pila de protocolos pueden trabajar en la red, hemos de controlar de alguna manera los mensajes de multidifusión de ARP para mantener el consumo de ancho de banda de la conexión NRT en un nivel razonable. Lo haremos utilizando un mecanismo de entrega de mensajes en dos pasos: los nodos ThrottleNet no intercambiarán mensajes NRT directamente entre ellos, sino que los enviarán a GlobeThrottle y este los reenviará a sus destinatarios. Ello requerirá encapsular el mensaje original dentro de un mensaje intermedio, que será dividido y sus fragmentos enviados a GlobeThrottle. GlobeThrottle reconstruirá el mensaje intermedio a partir de sus fragmentos y obtendrá de ahí el mensaje original. Entonces, GlobeThrottle fragmentará el mensaje original y enviará los fragmentos al destinatario original de ese mensaje. Este proceso está mostrado en la Figura 3.6.

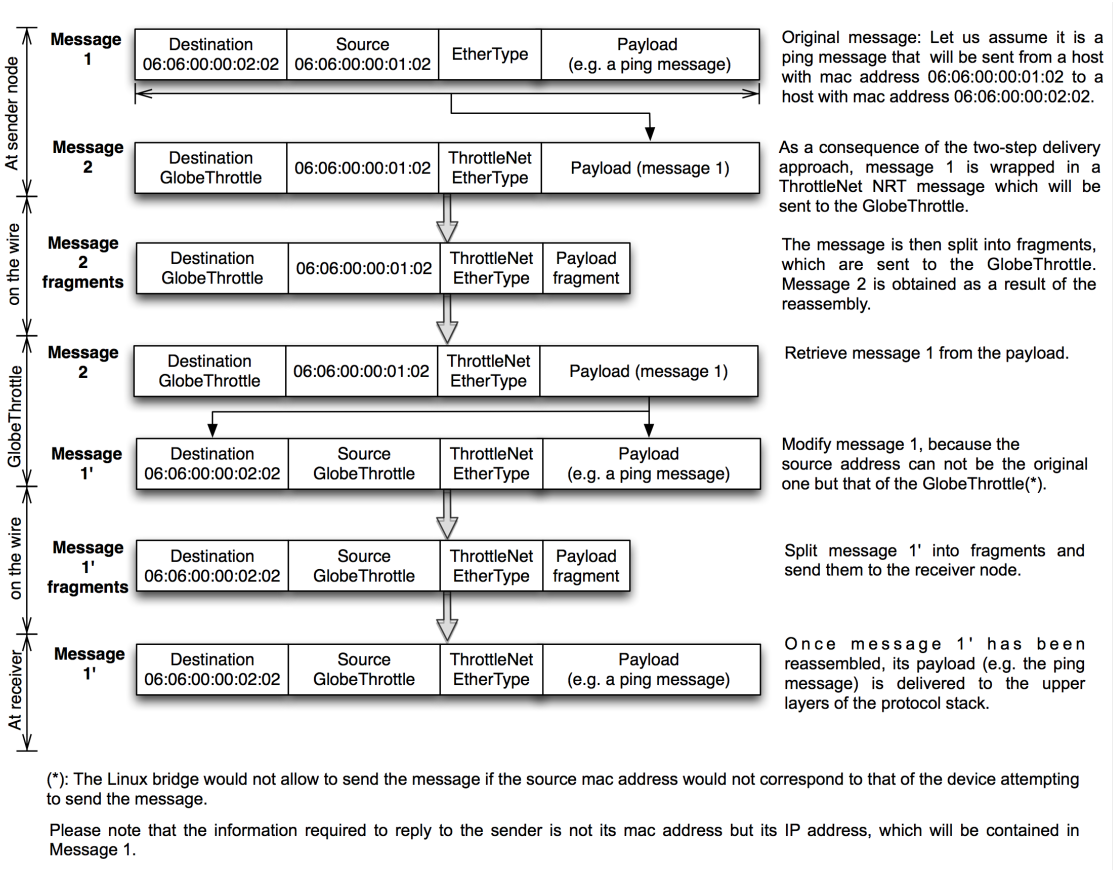

Figura 3.6: El mecanismo de entrega en dos pasos requiere encapsular el mensaje original dentro de otro y remitir este último a GlobeThrottle.

Como GlobeThrottle recibe y envía mensajes NRT de/hacia los nodos ThrottleNet, podemos afirmar que existen dos conexiones lógicas entre cada nodo ThrottleNet y GlobeThrottle: una que envía datos del nodo a GlobeThrottle y otra que funciona en sentido contrario. GlobeThrottle utiliza una cola de entrada para procesar los mensajes que llegan a través de la conexión que remite datos desde el nodo y una cola de salida para enviar mensajes a través de la conexión que transmite datos hacia el nodo.
Cuando un mensaje es reensamblado en una cola de entrada, su dirección de destino es examinada. Si es una dirección *unicast*, el mensaje es puesto en la cola de salida del nodo con dicha dirección. En cambio, si la dirección es multidifusión (*broadcast*) (e.g. ARP) GlobeThrottle envía una copia *unicast* de ese mensaje al resto de nodos en la red (pero no al emisor) y descarta el mensaje de multidifusión. De esta manera los mensajes multidifusión NRT no ocasionan problemas y **ARP puede ser usado para resolver direcciones de tráfico NRT**.

Imaginemos el siguiente escenario: hay una red de área local en la que utilizamos IP sobre ThrottleNet. Hay cuatro computadoras en la red, cada una de las cuales se conecta a la red utilizando el protocolo ThrottleNet. Denominaremos a estos equipos nodos A, B, C y D, respectivamente. Un usuario trata de utilizar el nodo A para enviar una solicitud *ping* al nodo B. El proceso desencadenado es el siguiente:

1. Las capas superiores de la pila de protocolos en el nodo A generan un mensaje ARP para hallar la dirección MAC del nodo B. Como ya se ha explicado, el mensaje ARP se encapsula dentro de otro que va dirigido a GlobeThrottle. Este último mensaje es fragmentado y sus fragmentos colocados en una cola de transmisión a la espera de ser enviados. El proceso es ilustrado en la Figura 3.6 (con la salvedad de que el mensaje enviado no será un mensaje *ping* sino un mensaje ARP).

2. Una vez haya recibido todos los fragmentos, GlobeThrottle los ensamblará para formar el mensaje que actúa como envoltorio y de su interior recuperará el mensaje original. GlobeThrottle verá que su dirección destino es de tipo *broadcast* y entonces creará copias *unicast* del mensaje y las enviará a las colas de salida asociadas a los nodos B, C y D (ver Figura 3.7). El mensaje ARP original será descartado.

3. Las versiones *unicast* del mensaje ARP son fragmentadas y sus fragmentos transmitidos a B, C y D. Cada nodo reensamblará el ARP y lo enviará a las capas superiores de la pila de protocolos, las cuales lo entregarán a la aplicación que corresponda. Las capas superiores del nodo

B verán que la dirección IP contenida en el ARP es la del nodo B y generarán un mensaje ARP de respuesta identificando al nodo B como receptor y dando a conocer su dirección MAC. Este mensaje también será envuelto en otro que será enviado al GlobeThrottle, a causa del mecanismo de entrega en dos pasos.

4. Tanto la respuesta de B a A como los mensajes *ping* que A y B intercambiarán serán de tipo *unicast*. El proceso para transmitir estos mensajes es mostrado en la Figura 3.8.

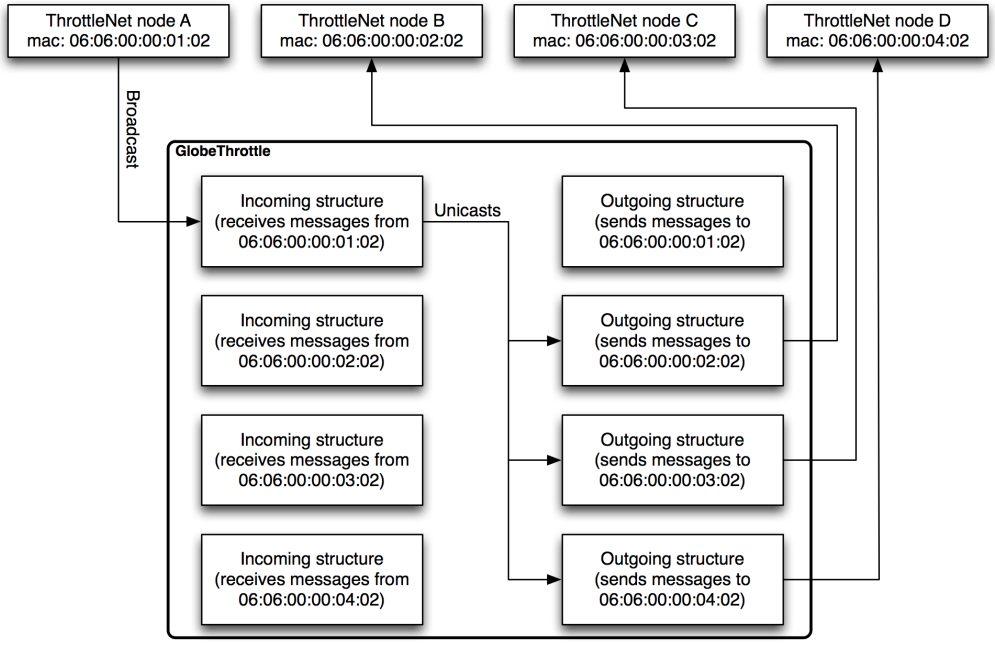

Figura 3.7: Tratamiento de mensajes multidifusión dentro de GlobeThrottle. Estos mensajes suelen ser mensajes de descubrimiento/exploración de ARP.

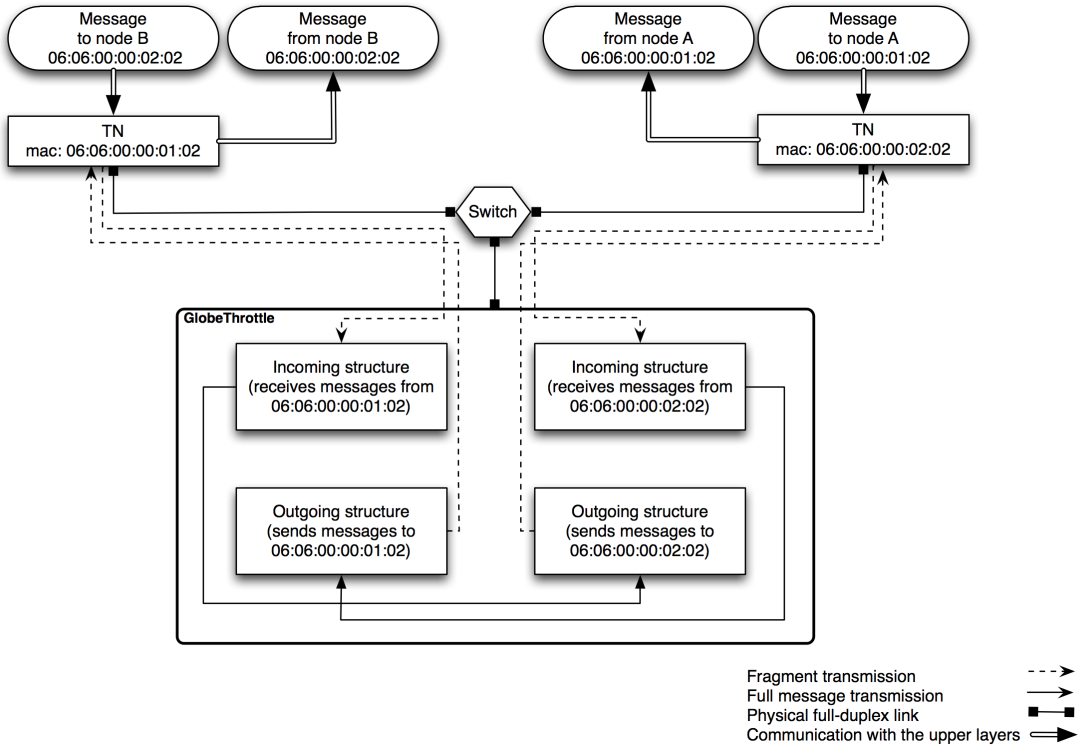

Figura 3.8: Tratamiento de tráfico NRT dentro de GlobeThrottle y comunicación con las capas superiores de la pila de protocolos.

3.4. Tipos de tráfico en una red ThrottleNet

Todos los mensajes transmitidos a través de una red ThrottleNet son tramas Ethernet provistas de cabeceras ThrottleNet (ver Figura 3.9). Los campos de estas cabeceras contienen la siguiente información:

- Direcciones MAC **fuente** y **destino**.
- **Identificador de fragmento** y **número de fragmentos en que ha sido dividido el mensaje original**. Esta información es requerida por la lógica que controla la recepción de mensajes (ver Sección 3.2.2).
- **Longitud** del mensaje.
- Por último, y de especial interés para esta sección, el **tipo de mensaje** que se obtendrá del ensamblaje de los fragmentos.

Los diferentes tipos de mensajes ThrottleNet pueden clasificarse atendiendo a diferentes criterios, de los que el más importante es la necesidad de

respetar los plazos de entrega de los mensajes (i.e. clasificación como tráfico RT o NRT).

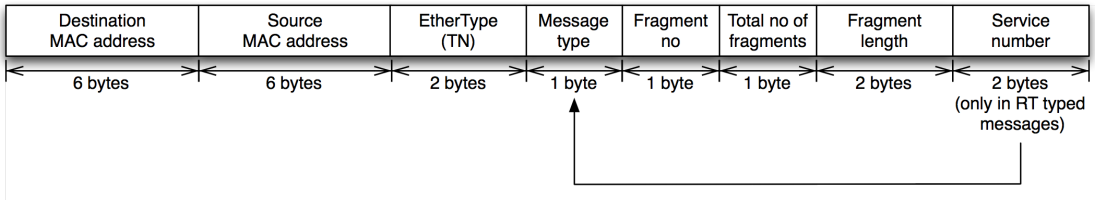

Figura 3.9: Cabeceras utilizadas por los mensajes enviados en una red ThrottleNet. El campo identificador de servicio se utiliza para identificar la conexión lógica utilizada (si el mensaje no se envía a través de la conexión NRT).

3.4.1. Tráfico RT

Por definición, el tráfico de tiempo real es aquel que requiere que sus mensajes se entreguen a tiempo. Este tráfico fluye a través de conexiones lógicas que pueden ser configuradas para asegurar la entrega a tiempo de los mensajes. Los parametros que admiten ser configurados son:

- El mínimo período de espera entre dos transmisiones de mensajes consecutivas. Puede tomar cualquier valor entre 1 microsegundo y un segundo.
- El máximo tamaño de los mensajes enviados a través de la conexión. Este parámetro determina el tamaño de los fragmentos en que serán divididos los mensajes para poder ser transmitidos. Toma valores entre 64 y 1500 *bytes*, que son las longitudes mínimas y máximas permitidas para tramas Ethernet.

Los nodos se suscriben a las conexiones lógicas como fuentes o sumideros de datos. GlobeThrottle mantiene informadas a las fuentes de cada servicio acerca de quienes son los sumideros del servicio (así es como se **resuelven direcciones para tráfico RT**). Por tanto, cuando una fuente quiere enviar un mensaje a través de una conexión, crea una copia *unicast* del mensaje para cada sumidero conocido y envía las copias. Es interesante observar que:

1. Las fuentes de conexiones lógicas para las que no hay sumideros registrados siempre descartan los mensajes que se les pide que envíen, por lo que no consumen ancho de banda.

2. El ancho de banda consumido por una fuente es directamente proporcional al número de sumideros del servicio, pues todos los mensajes *unicast* a los sumideros se envían en paralelo. Esto debe ser considerado cuando se evalúan los riesgos de aceptar una petición de establecimiento de conexión (ver Sección 3.3.2).

3.4.2. Tráfico NRT

El tráfico NRT engloba a aquellos mensajes que no han de cumplir con plazos de entrega, pero que se desea sean entregados lo más rápido posible. El tráfico NRT puede ser generado por el usuario o por la propia red ThrottleNet para efectuar tareas de mantenimiento. Este último se usa para conocer el estado de la red (e.g. nodos que hay en la red) y el usuario desconoce su existencia. Independientemente de su origen, todos los mensajes NRT se envían a través de la conexión NRT. Los diferentes tipos de mensajes son:

- **Mensajes multidifusión para encontrar a GlobeThrottle**. Los nodos ThrottleNet necesitan poder comunicarse con GlobeThrottle, pues es este quien proporciona los mecanismos de resolución de direcciones en una red ThrottleNet. Si un nodo no puede comunicarse con GlobeThrottle, tampoco podrá enviar mensajes de ningún tipo, pues no será capaz de hallar las direcciones físicas de los destinatarios de los mensajes. Por esto, cada vez que un nodo se conecta a la red ThrottleNet, su primera tarea es enviar estos mensajes de manera periódica hasta obtener respuesta. Cuando GlobeThrottle recibe un mensaje de este tipo de un nodo que aún no consta como registrado en la red, GlobeThrottle crea las estructuras necesarias para sustentar la comunicación del nodo en la red (i.e. procesar sus mensajes NRT, atender peticiones de establecimiento de conexiones, etc) y comienza a enviar *keepalives* al nodo en cuestión, para hacerle saber que se ha conectado con éxito a la red. Los nodos recuperan la dirección de GlobeThrottle de los *keepalives* y pueden entonces empezar a operar normalmente (i.e. enviar sus mensajes).

- Mensajes para **solicitar el establecimiento o desvinculación de una conexión lógica**. Los primeros sirven para pedir permiso a GlobeThrottle para crear una nueva conexión lógica de tiempo real, mientras que los segundos indican a GlobeThrottle que un nodo no desea enviar/recibir más datos a través de una determinada conexión y que, por tanto, el ancho de banda que dicha conexión utilizaba vuelve a estar disponible.

- ***Keepalives* de los nodos ThrottleNet** hacia GlobeThrottle. Gracias a estos mensajes, GlobeThrottle está siempre al corriente de las conexiones lógicas a las que cada nodo está suscrito. Además, si GlobeThrottle dejase de recibir estos mensajes, podría inferir que el nodo ThrottleNet que los emitía ya no está conectado a la red (debido a un problema de conexión o similar). En ese caso, GlobeThrottle procedería a eliminar tanto el nodo como sus suscripciones a servicios de su base de datos. Naturalmente, tampoco enviaría más *keepalives* hacia ese nodo.

- ***Keepalives* de GlobeThrottle** hacia los nodos ThrottleNet. Estos mensajes satisfacen las siguientes necesidades:

  - Dar a conocer a los nodos ThrottleNet la dirección de GlobeThrottle, permitiendo así localizarlo.
  - **Proporcionar mecanismos de resolución de direcciones para tráfico de tiempo real**. Esto es así porque los *keepalives* que GlobeThrottle envía a cada nodo contienen, para cada conexión lógica de la que el nodo es una fuente, las direcciones MAC de los sumideros de la conexión.
  - Permitir a los nodos detectar si GlobeThrottle está conectado a la red o no. Si los nodos dejan de recibir *keepalives* de GlobeThrottle, deducirán que GlobeThrottle ha dejado de funcionar, no puede acceder a la red o ha cambiado de dirección MAC. Entonces, los nodos volverán a empezar a enviar periódicamente mensajes para encontrar a GlobeThrottle de nuevo.

- **Mensajes encapsulando tráfico de las capas superiores de la pila de protocolos como tráfico NRT**, para permitir así a los usuarios emplear programas o utilidades de dichas capas (e.g. *ping*).

Ambos tipos de *keepalives* sirven otro propósito: como se los envía con la suficiente frecuencia, evitan que el conmutador olvide a qué nodo ThrottleNet conduce cada uno de sus puertos de salida.

3.5. Tratamiento de ARP en situaciones de alta carga NRT

Imaginemos un escenario en que varios nodos ThrottleNet se envían mensajes *ping* (encapsulados usando la conexión NRT) entre ellos de forma continuada a lo largo del tiempo. Como Linux borra de forma periódica sus tablas de rutas, cada uno de los nodos habrá de enviar esporádicamente mensajes ARP para determinar las direcciones físicas de los otros nodos. Imaginemos ahora que *ping* está trabajando en modo inundación (*flood*) para simular alta carga de tráfico NRT. Cuando quiera que los ARPs se generen, se colocarán al final de la cola NRT (que será muy larga, como consecuencia de la velocidad a la que *ping* genera mensajes en modo inundación). Como los ARPs no serán enviados cuando deberían (desde el punto de vista del sistema operativo) sino tras todos los mensajes NRT que los preceden, se producirán pérdidas de mensajes debidas a que no será posible entregar aquellos mensajes NRT enviados tras haber borrado las tablas de rutas.

Este problema ha sido resuelto mediante una ligera modificación de la lógica que controla las colas de mensajes salientes: cuando un mensaje va a ser puesto en una cola de salida, se comprueba su EtherType[32]. Si este coincide con el de ARP, el mensaje se coloca en la primera posición de la cola de mensajes en vez de en la última, que sería lo habitual.

Capítulo 4

Gestión del proyecto

Esta sección describe cómo se ha gestionado el proyecto en cuanto a las metodologías utilizadas y al tiempo empleado para cada tarea. Por último, se hacen comentarios respecto del tamaño del *software* implementado.

4.1. Metodologías de desarrollo y gestión del tiempo

Los dos sistemas desarrollados (simulador e implementación de ThrottleNet) presentan ciclos de vida diferentes a causa de sus distintos niveles de complejidad y del planteamiento sugerido por cada supervisor. Se presentan a continuación las metodologías empleadas y se hacen referencias a las diferentes fases de desarrollo de cada sistema. Estas fases se muestran en la Figura 4.1.

4.1.1. El simulador

La primera parte del PFC está dedicada a la implementación de una herramienta de simulación que facilite el aprendizaje de LabComm. Las fases del proceso de desarrollo de este simulador se corresponden con las de un **ciclo de vida en cascada clásico**:

1. Inicialmente, se pidió al proyectando que efectuase un estudio de LabComm y redactase un pequeño tutorial a fin de familiarizarse con sus posibilidades. El objetivo de esto no era adquirir una gran pericia en el manejo de LabComm sino permitir al proyectando experimentar las cuestiones que se le pueden plantear a un ingeniero en su primera aproximación a LabComm. Esto tendría dos beneficios directos: en primer

lugar, el proyectando estaría más preparado para afrontar la tarea de facilitar a otros el aprendizaje de LabComm y, además, el tutorial desarrollado podría servir a nuevos usuarios como una toma de contacto. El tutorial está disponible en el Apéndice D.

2. Se discutió con los supervisores cuál sería la mejor manera de apoyar el proceso de aprendizaje de LabComm. Se coincidió en que un simulador sería la forma más apropiada de proporcionar ese apoyo y se delinearon los requisitos funcionales que dicho simulador debería cumplir (ver Sección 1.1).

3. El proyectando efectuó un estudio de las posibles herramientas que podrían ser empleadas para desarrollar el simulador: applets, servlets, Java Web Start[33], etc. Decidido esto, se procedió a diseñar por completo el sistema y su interfaz gráfica, atendiendo a su sencillez de uso y considerando todos aquellos aspectos que se preveía podían causar problemas durante la implementación: comunicación de cliente y servidor, ejecución de simulaciones, carga de código, etc.

4. Una vez aprobado el diseño, se procedió con la implementación. Se efectuaron pruebas unitarias y de integración sobre cada módulo añadido al sistema y, una vez terminada la implementación, se probó el sistema al completo.

5. El simulador fue implantado en una de las máquinas virtuales de la Universidad de Lund a fin de ofertar su uso al público en general y, en particular, a aquellos departamentos de diferentes universidades con los que existían acuerdos de colaboración y líneas de investigación comunes. La aplicación fue probada por miembros del Departamento de Robótica de la Universidad de Leuven (Bélgica) y del Departamento de Electrónica e Información del Politécnico di Milano (Italia) para asegurar su buen funcionamiento.

6. Finalmente, se elaboró una pequeña guía de ilustrando errores comunes y se puso a disposición de los usuarios. Esta guía puede consultarse en el Apéndice A.3.

La aplicación está disponible a través de la url: `http://vm15.cs.lth.se:2323/labCommDemo/Client.html`.

La Figura 4.2 muestra la relación de tiempos para cada una de las principales tareas de desarrollo.

4.1.2. La implementación de ThrottleNet

El proceso de desarrollo de ThrottleNet ha estado determinado por su elevada complejidad como sistema y por las complicaciones derivadas de su implementación como módulo de Linux. Se ha considerado apropiado desarrollar el sistema en una serie de iteraciones (nos referiremos a ellas como Fases), cada una de las cuales analiza un problema determinado y diseña e implementa una solución para el mismo. Estas iteraciones siguen una aproximación *bottom-up*: la primera iteración se ocupa de la implementación como módulo del sistema operativo y el resto de iteraciones dan solución a problemas de comunicación de complejidad creciente, hasta llegar a la solución final.

Como se puede ver a continuación (y en la Figura 4.1), las fases del proceso de desarrollo son muy similares a las del **modelo incremental**:

1. Inicialmente, se proporcionó una descripción del sistema al completo y de sus componentes y esquema de funcionamiento en detalle. Se discutieron alternativas y se motivó la implementación como aplicación de espacio de núcleo. De este proceso se obtuvieron los requisitos funcionales de ThrottleNet.

2. Se efectuó un estudio del trabajo previo y del material formativo facilitado para adquirir los conocimientos necesarios para afrontar el desarrollo de ThrottleNet. Entonces, se hizo un diseño a grandes rasgos del sistema.

3. **Fase I**: **Implementación de un driver** que defina una interfaz virtual de red (eth_bridge) y **estudio de su interacción con el sistema operativo**.

4. **Fase II**: **Averiguar cómo transmitir y recibir datos utilizando el driver Ethernet de Linux**. Se definió la interfaz de tráfico NRT (eth_nrt) y se la utilizó para enviar mensajes *ping* entre dos máquinas que utilizaban ThrottleNet. Esta fase planteó muchas complicaciones y requirió del análisis del código de red de Linux para poder ser resuelta.

5. **Fase III**: Introducción de los **mecanismos de racionamiento de tráfico**.

6. **Fase IV**: Introducción de **GlobeThrottle** en la red. Se diseñó GlobeThrottle y se implantó el mecanismo de entrega de mensajes en dos pasos.

7. **Fase V**: Introducción de **tráfico de tiempo real** y de **mantenimiento de la red**. Se añadió la interfaz eth_rt y se enriquecieron los *keepalives* enviados para contener información de la red.

8. **Fase VI**: Introducción de las **técnicas de gestión de ancho de banda**.

9. **Fase VII**: **Pruebas del sistema y de carga**.

La Figura 4.3 muestra la relación de tiempos para cada una de las principales tareas de desarrollo.

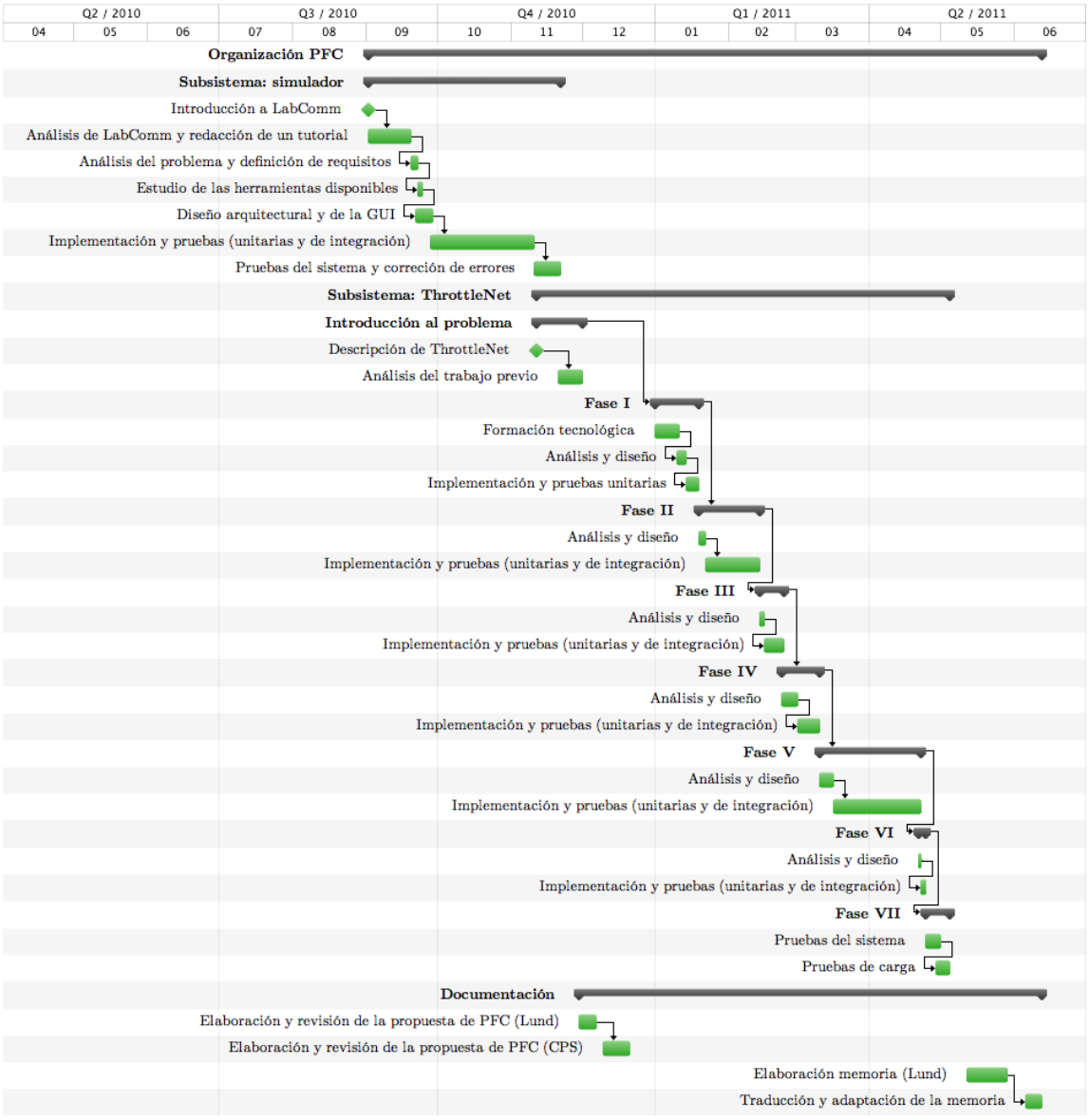

Figura 4.1: Diagrama de Gantt ilustrando las principales fases del proyecto y el tiempo dedicado a cada una.

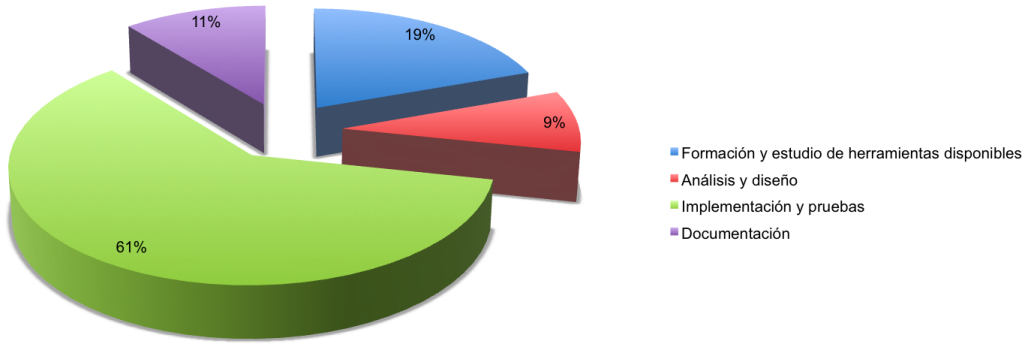

Figura 4.2: Relación de tiempos para cada una de las principales tareas del desarrollo del simulador.

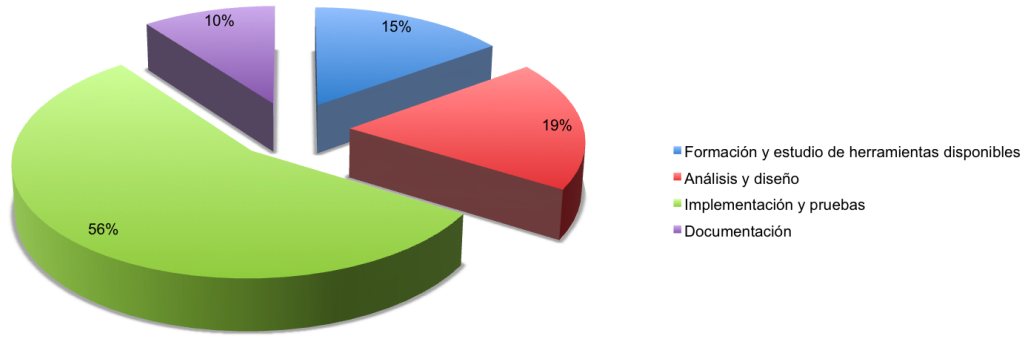

Figura 4.3: Relación de tiempos para cada una de las principales tareas del desarrollo de ThrottleNet.

4.2. Tamaño del proyecto

Esgrimir cifas como el número de líneas de código de que constan las implementaciones del simulador y de ThrottleNet (13260 y 8691 líneas de código[1], respectivamente) carece de sentido, pues estas no ofrecen una clara indicación de los esfuerzos realizados. Esto es especialmente cierto en el caso de la implementación de ThrottleNet, por los siguientes motivos:

- La aproximación incremental empleada hace difícil prever desde el principio cómo se va a organizar todo el código. Cada paso de una iteración a otra ha implicado grandes reestructuraciones de la aplicación que no se ven reflejadas en la métrica de líneas de código. Por ejemplo, la introducción de GlobeThrottle ha supuesto:

  - Cambiar parte de la lógica de envío y recepción de mensajes para poder implementar la entrega en dos pasos (ver Sección 3.3.3).
  - Reescribir todas las funciones de tratamiento de mensajes (fragmentación, reensamblado, planificación de envíos, etc.) de manera que puedan funcionar de tanto en espacio de núcleo como en espacio de usuario.

- No es tan importante el número de líneas de código como su calidad: el diseño no tiene por qué ser extenso, pero ha de ser adecuado para cooperar con el código de gestión de redes (*networking*) del núcleo, para permitir utilizar puentes Linux, etc. Así, el análisis y formación tecnológica suponen una parte muy importante de los esfuerzos realizados (ver Figura 4.3), pero no se ven reflejados en las líneas de código escritas.

- Por último, desarrollar ThrottleNet en C como una aplicación de espacio de núcleo ha permitido lograr una gran eficiencia a costa de un mayor tiempo de implementación y depurado, que tampoco se ve reflejado por la cantidad de líneas de código escritas.

Así, lo más apropiado no es medir el *software* implementado en cuanto a su longitud en líneas de código sino en cuanto a su complejidad y a la funcionalidad que brinda. Pese a no emplear ninguna métrica de punto función, es fácil apreciar la cantidad de funcionalidad implementada si se considera el amplio número de posibilidades de aplicación que el *software* ofrece; recordemos que proporciona una infraestructura de comunicación en tiempo real

[1]Se ha empleado el contador de líneas de código disponible en `http://code.google.com/p/loc-calculator/`.

para la comunicación de casi cualquier sistema distribuido a términos muy asequibles (en cuanto al *hardware* y *software* que requiere para funcionar).

En cualquier caso, para conocer el tamaño del proyecto en sí y la cantidad de esfuerzos que ha supuesto, también se han de considerar factores como la carga de formación necesaria para poder acometerlo, dificultades de implementación y depurado, etc.

Capítulo 5

Conclusiones

Esta última sección cierra la memoria presentando las contribuciones de este proyecto, evaluando el cumplimiento de los requisitos funcionales planteados al inicio del mismo (ver Sección 1.1), detallando las líneas de trabajo futuro que deja abiertas y con un resumen de la experiencia del proyectando.

5.1. Contribuciones

Como ya se dijo en la Sección 1, la Universidad de Lund proyecta ofrecer una solución a la comunicación en tiempo real utilizando LabComm y ThrottleNet. En el momento de iniciar este proyecto, la Universidad de Lund ya ofrecía una implementación de LabComm para su uso, pero esta no era ampliamente aceptada dado que no transmitía con claridad las posibilidades de aplicación que ofrecía a sus potenciales usuarios. Se había esbozado también una especificación del protocolo ThrottleNet, pero todavía no se disponía de una implementación. Este proyecto trata de contribuir a la solución planteada por la Universidad de Lund proporcionando medios para facilitar el aprendizaje del manejo de LabComm y desarrollando una implementación de ThrottleNet a partir de la especificación proporcionada.

Dado que la mejor manera de familiarizarse con una tecnología es experimentar con ella, este proyecto ha optado por **desarrollar un simulador que permita experimentar con LabComm, para así comprender qué puede hacer y cómo ha de usarse**. Este simulador se ha concebido como una **herramienta educativa** y su **simplicidad de uso** ha sido uno de los principales objetivos de diseño. Como las aplicaciones que requieren de un gran esfuerzo de configuración para poder ser usadas tienden a disuadir a los usuarios de emplearlas, esta herramienta de simulación ha sido diseñada

como una aplicación web para liberar al usuario de cualquier esfuerzo de configuración.

La aplicación presenta un escenario en que un productor y un consumidor intercambian datos empleando LabComm. El usuario puede modificar el código fuente de la simulación (i.e. el productor, el consumidor y el protocolo que define los datos transmitidos durante la simulación) a su gusto, siendo así capaz de alcanzar un elevado grado de personalización para cada simulación. Esta aplicación puede ser accedida a través de: `http://vm15.cs.lth.se:2323/labCommDemo/Client.html`.

Además, se ha desarrollado una implementación de ThrottleNet. Esta permite hacer uso de **redes de comunicación en tiempo real** que son **fácilmente desplegables** en una variedad de entornos gracias a que sus **requisitos de funcionamiento** son **muy asequibles**. En concreto, estas redes requieren del uso de un conmutador y enlaces *full-duplex*, utilizados para conectar a los nodos utilizando una topología de estrella (ver Sección 3). La implementación proporcionada (como *driver* Linux) permite una sencilla instalación en casi cualquier distribución Linux y, con ligeros cambios, en sistemas operativos del tipo UNIX, Mac OS, Windows, etc (ver Sección 5.3.3).

5.2. Cumplimiento de objetivos

Se examinan ahora los requisitos funcionales planteados para el simulador y la implementación de ThrottleNet y se explica si han sido cumplidos o no, cómo y qué problemas se han encontrado en el camino.

5.2.1. El simulador

- **Req. 1.1: cumplido**, pues la aplicación funciona como se deseaba: el usuario puede seguir un proceso iterativo de modificación y ejecución de las simulaciones hasta alcanzar los resultados deseados. Como ya se ha explicado, para conseguir esto hubo que dar solución al problema de recargar las clases dinámicamente utilizando un *applet* no firmado (ver Sección 2.6.2).

- **Req. 1.2: cumplido.** La implementación como aplicación web libera al usuario de las tareas de configurar internamente el simulador (e.g. configurar los compiladores, cargar las clases, etc). La Sección 1.2.1 explica esto con mayor detalle.

5.2.2. La implementación de ThrottleNet

- **Req. 2: cumplido**, pues ThrottleNet tan sólo requiere de enlaces *full-duplex* y de un conmutador y puede ser desplegado en prácticamente cualquier distribución Linux (ver Secciones 1 y 1.2.2).

- **Req. 2.1: cumplido**, porque se ha determinado que los nodos ThrottleNet no se comuniquen directamente a través de los enlaces físicos sino que deban solicitar permiso para establecer conexiones lógicas para poder comunicarse entre ellos (ver Sección 3).

- **Req. 2.2: cumplido**, gracias a que la implementación fragmenta los mensajes salientes y los envía respetando los períodos de espera entre transmisiones. A la recepción de los fragmentos, estos se ensamblan para formar el mensaje original (ver Sección 3.2).

- **Req. 2.3: cumplido**, porque cada nodo ThrottleNet cuenta con una conexión NRT por defecto y utiliza GlobeThrottle para resolver las direcciones de sus mensajes (ver Sección 3.3.3).

- **Req. 2.4: cumplido**, pues los nodos ThrottleNet pueden establecer conexiones lógicas de tiempo real a medida. GlobeThrottle proporciona resolución de direcciones para tráfico RT (ver Sección 3.4.1).

- **Req. 2.5: cumplido**, gracias a los *keepalives* que GlobeThrottle y el resto de nodos intercambian. Estos mensajes contienen toda la información necesaria para comunicar con GlobeThrottle, obtener direcciones de los destinatarios de tráfico RT, determinar qué agentes de la red continúan conectados, etc. Todos estos mensajes se envían a través del canal NRT, puesto que no tienen requisitos de entrega de tiempo real (ver Sección 3.4.2).

- **Req. 2.6: cumplido**. Este informe es producido y mostrado por GlobeThrottle de forma periódica y también cada vez que se establecen/eliminan conexiones lógicas en la red. La Figura 3.4 muestra uno de estos informes.

5.3. Trabajo futuro

Aquí se describen posibles mejoras al simulador y a la implementación de ThrottleNet realizados. Además, con base en los resultados obtenidos de este proyecto, se presentan los pasos a seguir para finalmente obtener la solución integrada proyectada.

5.3.1. Posibles mejoras para el simulador

Aunque LabComm permite generar las rutinas de serialización en varios lenguajes, esta primera versión del simulador tan sólo permite ejecutar simulaciones en lenguaje Java. **Añadir la funcionalidad necesaria para soportar más lenguajes** queda como tarea pendiente para futuras versiones de esta herramienta de simulación.

5.3.2. Posibles mejoras para ThrottleNet

Esta implementación de ThrottleNet emplea un GlobeThrottle para las tareas de gestión de ancho de banda y resolución de direcciones. Una desventaja de esta aproximación es que **convierte a GlobeThrottle en un punto singular de fallo de la red ThrottleNet**[28]: si falla o no funciona correctamente, no se podrá transmitir tráfico RT ni NRT en la red.
Sin embargo, GlobeThrottle permite encapsular tráfico NRT en la red, lo cual posibilita utilizar protocolos de las capas de Internet y Transporte sobre ThrottleNet. Sería deseable encontrar una solución que incrementase la fiabilidad de la red (i.e. la tolerancia frente a fallos de GlobeThrottle) al tiempo que mantenemos la capacidad de soportar tráfico NRT. Esto podría hacerse utilizando dos o más GlobeThrottles: uno activo y uno o varios redundantes, que se mantienen actualizados respecto del estado de la red analizando los *keepalives* enviados por el GlobeThrottle activo. El GlobeThrottle activo llevaría a cabo las tareas propias de GlobeThrottle y, en caso de fallo (que podríamos detectar en caso de dejar de recibir los *keepalives* de GlobeThrottle), uno de los redundantes lo sustituiría. Sin embargo, esta aproximación sólo será efectiva si GlobeThrottle no se comporta incorrectamente antes de fallar por completo, ya que de lo contrario los GlobeTrottle redundantes habrán analizado *keepalives* con información falsa y tendrán por tanto una idea errónea del estado real de la red.

5.3.3. Pasos hacia la solución integrada

Finalmente, enumeraremos las ampliaciones necesarias para alcanzar la solución integrada deseada:

1. **Proporcionar una implementación de ThrottleNet para aquellos otros sistemas operativos que podamos querer utilizar**. En un laboratorio de robótica no es raro encontrar sistemas como Xenomai y VxWorks. Portar ThrottleNet a éstos sería sencillo dada la actual implementación como driver. Para otros sistemas, como Windows, UNIX, etc, podríamos utilizar una implementación en espacio de usuario similar a la utilizada con GlobeThrottle (ver Sección 3.3).

2. **Crear una capa de abstracción que facilite el trabajo con las interfaces de red ofrecidas al usuario (eth_rt y eth_nrt)**. Como medida provisional, se ha proporcionado un programa para que el usuario pueda solicitar la creación y eliminación de conexiones lógicas, además de enviar y recibir datos a través de estas. El código de esta aplicación está contenido en el Apéndice B.3.

3. **Asociar a las conexiones lógicas de tiempo real una definición de los tipos de datos que serán enviados a través de ellas** (el protocolo LabComm), consiguiendo así la previamente mencionada abstracción de "servicio".

5.4. Experiencia personal

Este proyecto me ha resultado una experiencia muy grata, en cuanto a que me ha permitido poner en práctica los conocimientos adquiridos durante la carrera, experimentar con nuevas formas de desarrollar *software* y, sobre todo, aprender muchas cosas nuevas. Además, me ha brindado la posibilidad de trabajar con gente realmente capaz y brillante, y de aprender de su manera de afrontar los problemas técnicos, por desesperantes que puedan llegar a ser.

Apéndice A

La herramienta de simulación

A.1. Transferencia de ficheros mediante ObjectStreams

Esta sección contiene el código empleado para transferir ficheros de código fuente entre el cliente y el servidor del simulador.

```
/**
 * Class FileTransfer.
 * Provides methods for transferring text files.
 */
package common;

import java.io.*;
import java.util.Arrays;


public class FileTransfer {

  private static final int BUFFER_SIZE = 256;

  /**
   * Returns a String containing all text in the requested file. This
   * file is read from the socket which connects to the server.
   */
  public String receiveFile(ObjectInputStream in)
  throws IOException, ClassNotFoundException, Exception {
```

```
  Object o;
  byte [] buffer;

  Integer bytesRead = 0;
  ByteArrayOutputStream baos = new ByteArrayOutputStream(BUFFER_SIZE);
  // Start receiving the file
  do {
    o = in.readObject();
    if (!(o instanceof Integer)) {
      throw new Exception("Size of next msg expected.");
    }
    bytesRead = (Integer) o;

    o = in.readObject();
    if (!(o instanceof byte [])) {
      throw new Exception("Array of bytes expected.");
    }
    buffer = (byte []) o;
    baos.write(buffer, 0, bytesRead);
  } while (bytesRead == BUFFER_SIZE);
  String myFile = baos.toString();

  return myFile;
}


public void sendFile(ObjectOutputStream out, String modifiedFile) {

  try {
    ByteArrayInputStream bais = new ByteArrayInputStream(
        modifiedFile.getBytes());
    byte[] buffer = new byte[BUFFER_SIZE];
    Integer bytesRead = 0;

    while ((bytesRead = bais.read(buffer)) > 0) {
      out.writeObject(bytesRead);
      out.writeObject(Arrays.copyOf(buffer, buffer.length));
    }
  }
  catch (IOException e) {
    e.printStackTrace();
```

```
      System.exit(1);
    }
  }


  public void sendFile(File file, ObjectOutputStream out) {

    FileInputStream fis = null;
    try {
      fis = new FileInputStream(file);
      byte [] buffer = new byte[BUFFER_SIZE];
      Integer bytesRead = 0;

      while ((bytesRead = fis.read(buffer)) > 0) {
        out.writeObject(bytesRead);
        out.writeObject(Arrays.copyOf(buffer, buffer.length));
      }
    }
    catch (IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
    finally {
      try {fis.close();}
      catch (IOException e) { }
    }
  }
}
```

A.2. Comunicación productor-consumidor

Esta sección contiene el código empleado para comunicar al productor y al consumidor con independencia de la máquina en que se ejecutan (cliente o servidor).

CommunicationSupport interface

```
/**
 * Interface to allow Producers and Consumers to communicate
 * regardless of the host they are running at (client or server).
 */

package clientServerSupport;

import java.net.*;


public interface CommunicationSupport {

  /**
   * Used to connect a producer and a consumer.
   * Implementation depends on the implementing class being either
   * a server or a client.
   * It may block if the port is on use!
   */
  public void connect();


  /**
   * Used to obtain the socket used to communicate.
   */
  public Socket getSocket();
}
```

ClientSupport class

```
/**
 * Allows a producer or a consumer to communicate to
 * each other regardless of  their situation, either at client
 * side or at server side (holding the ServerSocket).
```

```
 *
 * Once communication has been established through the
 * use of connect(), the getter method for the socket
 * should be called.
 */

package clientServerSupport;

import java.io.IOException;
import java.net.*;


public class ClientSupport implements CommunicationSupport {


  private int port;
  private String host;
  private Socket socket;


  public ClientSupport(String host, int port) {

    this.host = host;
    this.port = port;
    socket = null;
  }


  /**
   * Connect to a server using host and port attributes.
   */
  public void connect() {

    boolean connected = false;
    while (!connected) {
      try {
        socket = new Socket(host, port);
        if (socket != null)
          connected = true;
      }
      catch (UnknownHostException uhe) {
```

```
        uhe.printStackTrace();
        System.exit(1);
      }
      catch (IOException ioe) {
        /* Server is not listenning for connections yet. Keep on trying. */
      }
    }
  }


  /**
   * Return the socket used to communicate.
   */
  public Socket getSocket() {
    return socket;
  }
}
```

ServerSupport class

```
/**
 * Allows a producer or a consumer to communicate to each
 * other regardless of their situation, either at client side or
 * at server side (holding the ServerSocket).
 *
 * Once communication has been established through the
 * use of connect(), the getter method for the socket should
 * be called.
 */
package clientServerSupport;

import java.net.*;
import java.io.*;

public class ServerSupport implements CommunicationSupport {

  private int port;
  private ServerSocket sSocket;
  private Socket cSocket;
```

```
  public ServerSupport(int port) {

    super();
    this.port = port;
    this.sSocket = null;
    this.cSocket = null;
  }


  /**
   * Create a server socket and await a connection from a client.
   */
  public void connect() {

    try {
      sSocket = new ServerSocket(port);
      try {
        cSocket = sSocket.accept();
      }
      catch (IOException ioe) {}
      finally {
        sSocket.close();
      }
    }
    // Server could not create a ServerSocket.
    catch (IOException e) {
      System.err.println(e);
      e.printStackTrace();
      System.exit(1);
    }
  }


  /**
   * Return the socket used in the connection.
   */
  public Socket getSocket() {

    return cSocket;
  }
}
```

A.3. Common mistakes when using the simulator

There is a series of possible common mistakes which may be made when trying to add new samples to encode and decode in an existing system. Since these mistakes are located at either the producer or the consumer source code, this quick guide has been divided into two such sections.

A.3.1. Producer code

- Try to encode something with a non-existent encoder: **Compilation fault**.
  Example: theEncoder.encodeIt("foo bar");

- Try to encode something with an unappropriate encoder. Different possibilities follow...

  1. Use an encoding routine whose parameter type does not match the type of the sample you try to encode: **Compilation fault**.
     Example: *theEncoder.handleABoolean(5);*
  2. Use an encoding routine whose parameter type does not match the type of the sample you try to encode, but automatic conversion is allowed: **Permitted** and **should** work fine.
     Example: *theEncoder.handleADouble(5);*
     Example: *theEncoder.handleADouble((double) 5);*
  3. Try to encode a vector sample using an encoding routine which accepts as a parameter a vector of the same type but has a different number of components. If the size of the vector to be encoded is

     - **bigger** than that expected, there will be no compile or runtime failures but (**attention!**) the latter components of the vector will not be sent.
     - **smaller** than that expected, an exception will be thrown while traversing the vector. In the providen implementation of the Producer this exception is caught and properly treated.

- Have encoder routines with the same name.

- If they are declared to encode the same type of sample, that is, accept the same type of parameter, then there will be a **compilation fault**, due to the fact that there are duplicated symbols.
  Example:
  public void encodeADouble(double value) {...}
  public void encodeADouble(double v) {...}
- If they are declared to encode samples of different types, that is, accept different types of parameters then there is no problem at all; the method which has not been declared in the Handler of the class which encoder routine we want to use will be ignored.

A.3.2. Consumer code

First, there is no possibility to decode a sample using a decoder routine which is not suitable for such a sample. The reason is that the sample messages are self-descriptive and thus know which routine they need to use to be decoded[1]. Thus, either the decoder routine has not been properly registered to the decoder or there are duplicated routines or any other kind of error which may cause compilation faults.

- For the case in which there are decoder routines with the same name, the same described at the Producer section applies.
- Should there not be a routine registered, the causes for that may be any of the following:

  1. Cause **A**: The decoder is not declared to implement the handling routine for the class which has samples communicated through the channel. That is, the *implements* clause is missing.
     Example: *class MyDecoder implements aBoolean.Handler* is missing.
  2. Cause **B**: The decoder does not register the decoder routine.
     Example: The statement *ABoolean.register(dChannel, this);* is missing.
  3. Cause **C** : The decoder routine is missing, that is, there is no implementation for it.

  And, depending on the combination of causes A, B and C the result may be one of those shown in table A.1 (*P* will stand for "The sentences which would avoid cause X from happening are present.[a]nd *M* for "the statements that would avoid cause X from happening are missing").

[1] The routine which implements the Handler declared at that class.

A	B	C	Consequence
M	M	M	IOException: No dispatcher for ABoolean
M	M	P	IOException: No dispatcher for ABoolean
M	P	M	Compilation fault: Can not find method handleABoolean
M	P	P	Compilation fault: Can not find method ABoolean.register()
P	M	M	Compilation fault: Method handleABoolean is not overriden
P	M	P	IOException: No dispatcher for ABoolean
P	P	M	Compilation fault: Can not find method handleABoolean
P	P	P	OK

Cuadro A.1: Different outcomes depending on proper registration of the decoder routines.

- Finally, if the decoder would not be attached to a communication channel, an exception with the message "size of next message expected"would be thrown.
  Example:*dChannel = new LabCommDecoderChannel();*
  instead of
  dChannel = new LabCommDecoderChannel();.

Apéndice B

La implementación de ThrottleNet

B.1. Racionamiento de tráfico

B.1.1. Procesamiento de tráfico saliente

```
/* Transfer a message using the ThrottleNet protocol.
 * @dst_addr:   receiver of the message
 * @msg_type:   throttleNet type of the message
 * @service_id: id of the logical connection
 * @skb:        Linux networking structure. Contains the message
 * @outgoing:   structure for outgoing traffic. Its lock/semaphore
 *              has been taken before calling this function.
 */
tn_tx(dst_addr, msg_type, service_id, skb, outgoing) {

  // Allow no traffic until GlobeThrottle has been found.
  if (not_known(globeThrottle_address)) {
    drop_message(skb);
    return;
  }
  /* If device is being used, enqueue the message to be
   * sent later on.
   */
  if (being_used(outgoing)) {
    enqueue_msg(outgoing, dst_addr, skb, msg_type, service_id);
    return;
  }
```

```
  // Else split it into fragments and schedule their transmission
  else {
    split_msg(dst_addr, skb.msg, msg_type, service_id, outgoing);
    mark_used(outgoing);
    start_tx_timer(outgoing);
  }
}


/* This function is triggered whenever a transmission timer is
 * activated at an outgoing structure.
 * It performs a delayed transmission of a fragment and schedules
 * transmission of the remaining fragments for that message.
 * Should there be no more fragments to be sent, it will split a
 * message pending to be sent into fragments and schedule
 * their transmission.
 * If nothing is to be sent, marks the outgoing structure as not
 * being used.
 * @timer: timer which triggered this function.
 */
send_delayed_fragment(timer) {
  var remaining = 0, successful, scheduled = false;
  var skb, dst_addr, msg_type, service_id;

  // Get outgoing with some offset calculations and lock it
  outgoing = container_of(timer);
  take_lock(outgoing);

  // If there are fragments to send, send one
  if (get_fragments_to_send(outgoing) > 0) {
    remaining = send_enqueued_fragment(outgoing);
  }

  /* If there are still fragments to send, restart the
   * transmission timer.
   */
  if (remaining > 0) {
    start_tx_timer(outgoing);
    scheduled = true;
  }
  else {
```

```
    /* No more fragments to send. If there are enqueued messages
     * split them into fragments and schedule their transmission.
     */
    <skb, dst_addr, msg_type, service_id> = get_awaiting_msg(outgoing);
    if (skb != NULL) {
      split_msg(dst_addr, skb->msg, msg_type, service_id, outgoing);
      start_tx_timer(outgoing);
      scheduled = true;
  }
  // If there are no fragments to send...
  if (!scheduled) {
    mark_unused(outgoing);
  }
  return_lock(outgoing);
}
```

B.1.2. Procesamiento de tráfico entrante

```
/* Process an incoming fragment under the assumption that all
 * fragments arrive in order.
 * @msg: a newly arrived fragment
 * @incoming: structure in which the incoming fragments will
 *            be stored until reassembling takes place.
 * @handle_packet: function to call when the packet has
 *                 been reassembled.
 *                 Different types of messages may require
 *                 different handling
 * All appropriate locks must be held before calling this
 * function.
 */
void handle_incoming_msg(msg, incoming, handle_message) {
  var latest_frag_no, latest_total_frags;
  var enqueue_it = false, flush_assembly_queue = false;

  /* Get the fragment's fragment number and total number
   * of fragments in which the message is fragmented.
   */
  latest_frag_no = get_fragment_number(msg);
  lastest_total_frags = get_total_frags(msg);

  /* Possible cases...*/
```

```
  // 1st fragment arrives, assembly queue is empty
  if (empty_assembly_queue(incoming) &&
      is_first_fragment(msg)) {
    enqueue_it = true;
  }
  // 1st fragment arrives, assembly queue not empty...
  else if (!empty_assembly_queue(incoming) &&
           is_first_fragment(msg)) {
    enqueue_it = true;
    flush_assembly_queue = true;
  }
  // The fragment which we expected arrives...
  else if (!empty_assembly_queue(incoming)                        &&
           latest_frag_no     == expected_frag_no(incoming)       &&
           latest_total_frags == expected_total_frags(incoming)) {
    enqueue_it = true;
  }
  // We don't want the packet.
  else {
    flush_assembly_queue = true;
  }
  // Enqueue the fragment
  if (enqueue_it) {
    enqueue_fragment(incoming, msg);

    // Have we received all fragments?
    if (latest_frag_no == expected_total_frags(incoming)) {
      assemble_message(incoming);
      handle_message(incoming);
      clean_assembly_buffer(incoming);
      // Flush the assembly queue.
      flush_assembly_queue = true;
    }
  }
  // Flush the assembly queue, if needed
  if (flush_assembly_queue) {
    discard_packets_to_assemble(incoming);
  }
}
```

B.2. Gestión de ancho de banda

Toma de decisiones para la aceptación de establecimiento de conexiones.

```
/* Calculate no of bytes that this service will send/receive
 * using the link in one second.
 * @freq: inter-fragment gap for the connection (nsecs).
 * @size: max bytes per fragment sent using this connection
 */
double bw_link_per_service(freq, size) {
  var no_messages, bw;

  no_messages = 1E9 / freq;
  bw = no_messages * size;
  return bw;
}


/* Calculate no of bytes that this node sends/receives over
 * the link because of its connections to other ThrottleNet
 * nodes.
 * @mac: mac address of the node
 */
double bw_link_per_node(mac) {
  double bw = 0.0, service_bw;
  var source_for_list, sink_for_list;    // Lists of services

  // Get all services for which the node is source/sink
  source_for_list = get_services_for_which_this_node_is_a_source();
  sink_for_list = get_services_for_which_this_node_is_a_sink();

  /* We will send unicasts to all receivers of connections for which
   * we are sources. The required bandwidth will depend on the
   * connection specifics (size and frequency) and on the number of
   * nodes subscribed to that connection.
   */
  foreach (service in sink_for_list) {
    service_bw = bw_link_per_service(service.freq, service.size);
    no_sinks = service.no_sinks;
```

```
    bw += no_sinks * service_bw;
  }
  /* We will receive messages from all connections to which we are
   * subscribed as receivers if there is a source defined for that
   * connection.
   */
  foreach (service in source_for_list) {
    service_bw = bw_link_per_service(service.freq, service.size);
    bw += service.source_is_defined() ? service_bw : 0;
  }

  // Do not forget to account bandwidth reserved for NRT
  bw += NRT_RESERVED_BW;
  return bw;
}


/* Calculate how many bytes will be stored in the output
 * buffers of the switch in case of simultaneous reception
 * of messages from connections of which this node is a
 * receiver.
 */
double bw_buff_per_node(mac) {
  double bw = 0.0;
  var sink_for_list;

  sink_for_list = get_services_for_which_this_node_is_a_sink();

  foreach (service in sink_for_list) {
    // Just consider it if the service source is defined
    if (service.source_is_defined) {
      bw += service.size;
    }
  }
  return bw;
}


/* Check if this service requirements can be satisfied given the
 * current state of the network. Particularly, this function
 * performs checks to avoid overloading the links between the nodes
```

```
 * and the output buffers of the switch (in case of simultaneous
 * reception of messages from several services)
 * @mac:    mac address of the node subscribing the service.
 * @id:     service id.
 * @freq:   service frequency.
 * @size:   service max packet size.
 * @is_src: is the node registering as a source? (boolean)
 * @used_link_bw: already allocated link bandwidth for this
 *                node.
 * @used_buff_bw: already allocated space at the switch output
 *                buffer.
 *
 * Returns true if registering the service is feasible in the
 * current situation and false otherwise.
 */
boolean bw_check_viability(mac, id, freq, size, is_src,
                           used_link_bw, used_buff_bw) {
  boolean abort = false;
  var service, service_sinks, service_source;
  var extra_link_bw = 0, extra_buff_bw = 0, new_link_load,
      sink_link_bw, sink_buff_bw;

  /* If the service already exists in GlobeThrottle's database,
   * check how accepting this request affects the nodes related
   * to the service.
   */
  if ((service = find_service(database, id)) != NULL) {
    new_link_load = bw_link_per_service(freq, size);
    /* Check: 1) Is there any node which has no available bandwidth
     *           to meet this service requirements?
     *        2) Can we (along with the other sources of services
     *           to which the node is subscribed) overload the output
     *           buffer for any of the sinks of this service in case
     *           all the service messages would arrive simultaneously?
     */
    if (is_src) {
      service_sinks = get_sinks_for_this_service(service);
      foreach (sink in service_sinks) {
        sink_link_bw = bw_link_per_node(sink.mac);
        sink_buff_bw = bw_buff_per_node(sink.mac);
        // Check 1)
```

```
        if (sink_link_bw + new_link_load > LINK_BW) {
          abort = true;
        }
        // Check 2)
        if (sink_buff_bw + size > BUFFER_BW) {
          abort = true;
        }
      }
      // We have to send messages from this service to each sink
      extra_link_bw = service.no_sinks * new_link_load;
    }
    else {
      if (service.source_is_defined) {
        source_link_bw = bw_link_per_node(service.source.mac);
        // Check 1, check 2 makes no sense here
        if (source_link_bw + new_link_load > LINK_BW) {
          abort = true;
        }
        extra_link_bw = new_link_load;
        extra_buff_bw = size;
      }
    }
  }
  if (abort)
    return false;
  // Can we afford accepting the new service?
  if (used_link_bw + extra_link_bw > LINK_BW) {
    return false;
  }
  if (used_buff_bw + extra_buff_bw > BUFFER_BW) {
    return false;
  }
  return true;
}
```

B.3. Interacción con la interfaz RT vía ioctl

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <net/if.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>            // CLOCK_MONOTONIC
#include "tn_headers.h"
#include "services.h"
#include "tn_timeouts.h"
#include "ioctl_commands.h"

/************************/
#include <netpacket/packet.h>
#include <linux/if_ether.h>
/************************/


/* Create a connection to send ioctl commands to the ThrottleNet device.
 * Returns zero on failure, non-zero otherwise.
 */
int init(void);

/* Init a service.
 * Returns non-zero on success.
 */
int start_service(__u16 s_id, __u32 s_freq, __u16 s_data, char *s_desc,
                  int desc_len, int s_is_src);


/* Stop a service */
int stop_service(__u16 s_id);
```

```
/* Write a message using a service */
void write_msg(__u16 service_id, __u8 *msg, unsigned int msg_len,
               int eager);


/* Read a message from a service */
void read_msg(__u16 service_id, int eager);



/************************ Global variables ************************/
int fd;                           // fd to write on the TN device
char name[8] = "eth_rt";          // TN device name

/****************** Options and suboptions parsing ******************/
enum {
  ID_IDX = 0,
  FREQ_IDX,
  SIZE_IDX,
  DESC_IDX,
  IS_SRC_IDX,
  MSG_IDX,
  EAGER_IDX
};

char *const token[] = {
  [ID_IDX]     = "id",
  [FREQ_IDX]   = "freq",
  [SIZE_IDX]   = "size",
  [DESC_IDX]   = "desc",
  [IS_SRC_IDX] = "is_src",
  [MSG_IDX]    = "msg",
  [EAGER_IDX]  = "eager",
  NULL
};

/*********************************************************************/

int main(int argc, char **argv) {
  int error_found = 0, eager = 0;
  char opt, *subopts, *value, *msg = NULL, *desc, mode = '\0';
```

```
int cid = -1, cfreq = -1, csize = -1, is_src = 0;
__u16 id, size;
__u32 freq;
extern char *optarg;


// Parse initialization arguments
while ((opt = getopt(argc, argv, "c:d:w:r:h")) != -1) {
  if (mode == '\0')
    mode = opt;
  else {
    error_found++;
    break;
  }
  switch (opt) {
    // Create a new service
    case 'c':
      subopts = optarg;
      while (*subopts && !error_found) {
        switch (getsubopt(&subopts, token, &value)) {
          case ID_IDX:
            if (value != NULL)
              cid = atoi(value);
            else
              error_found++;
            break;

          case FREQ_IDX:
            if (value != NULL)
              cfreq = atoi(value);
            else
              error_found++;
            break;

          case SIZE_IDX:
            if (value != NULL)
              csize = atoi(value);
            else
              error_found++;
            break;
```

```
      case DESC_IDX:
        if (value != NULL)
          desc = value;
        else
          error_found++;
        break;

      case IS_SRC_IDX:
        is_src++;
        break;

      default:
        error_found++;
        break;
    }
  }
  break;

// Delete an existing service
case 'd':
  subopts = optarg;
  while (*subopts && !error_found) {
    switch (getsubopt(&subopts, token, &value)) {
      case ID_IDX:
        if (value != NULL)
          cid = atoi(value);
        else
          error_found++;
        break;

      default:
        error_found = 1;
        break;
    }
  }
  break;

// Write a real-time message
case 'w':
  subopts = optarg;
  while (*subopts && !error_found) {
```

```
    switch (getsubopt(&subopts, token, &value)) {
      case ID_IDX:
        if (value != NULL)
          cid = atoi(value);
        else
          error_found++;
        break;

      case MSG_IDX:
        if (value != NULL)
          msg = value;
        else
          error_found++;
        break;

      case EAGER_IDX:
        eager++;
        break;

      default:
        error_found++;
        break;
    }
  }
  break;

// Read a real-time message
case 'r':
  subopts = optarg;
  while (*subopts && !error_found) {
    switch (getsubopt(&subopts, token, &value)) {
      case ID_IDX:
        if (value != NULL)
          cid = atoi(value);
        else
          error_found++;
        break;

      case EAGER_IDX:
        eager++;
        break;
```

```
          default:
            error_found = 1;
            break;
        }
      }
      break;

    // Help
    case 'h':
      printf("Provide a command and all necessary arguments.\n");
      printf("Commands:\n");
      printf("\tInstall a service   (i). Requires id, freq, size and "
             "description\n");
      printf("\tDelete a service    (d). Requires id.\n");
      printf("\tRead from a service (r). Requires id.\n");
      printf("\tWrite to a service  (w). Requires id and msg.\n");
      printf("Possible arguments (comma separated):\n");
      printf("\tService id           (id=)\n");
      printf("\tService frequency    (freq=)\n");
      printf("\tService message size (size=)\n");
      printf("\tService description  (desc=)\n");
      printf("\tMessage to write     (msg=)\n");
      break;

    // Wrong argument
    default:
      fprintf(stderr, "Unvalid argument %s\n", optarg);
      break;
  }
}

if (error_found || argc == 1) {
  fprintf(stderr, "\nType ioctler -h for help\n");
  return -1;
}

switch (mode) {
  case 'c':
    if (cid < 0 || cfreq < 0 || csize < 0 || desc == NULL) {
      fprintf(stderr, "Invalid arguments\n");
```

```
      return -1;
    }
    else {
      id = (__u16) cid;
      freq = (__u32) cfreq;
      size = (__u16) csize;
    }
    break;

  case 'd':
    if (cid < 0) {
      fprintf(stderr, "Invalid arguments\n");
      return -1;
    }
    else {
      id = (__u16) cid;
    }
    break;

  case 'w':
    if (cid < 0 || msg == NULL) {
      fprintf(stderr, "Invalid arguments\n");
      return -1;
    }
    else {
      id = (__u16) cid;
    }
    break;

  case 'r':
    if (cid < 0) {
      fprintf(stderr, "Invalid arguments\n");
      return -1;
    }
    else {
      id = (__u16) cid;
    }
    break;
}

// Set up everything and link to the device
```

```
  if (!init())
    exit(1);

  switch (mode) {
    case 'c':
      printf("start service (id = %u, freq = %u, size = %u,"
             "desc = %s, desc_len= %d, is_src = %d)\n",
             id, freq, size, desc, strlen(desc), is_src);
      start_service(id, freq, size, desc, strlen(desc), is_src);
      break;

    case 'd':
      printf("stop service (id = %u)\n", id);
      stop_service(id);
      break;

    case 'w':
      printf("write msg (id = %u, msg = %s, len = %d)\n",
               id, msg, strlen(msg));
      write_msg(id, msg, strlen(msg), eager);
      break;

    case 'r':
      printf("read msg (id = %u, eager = %s)\n",
              id, eager ? "true" : "false");
      read_msg(id, eager);
      break;
  }
  close(fd);
}


int init(void) {

  int result;
  struct ifreq ifr;
  struct sockaddr_ll sa;


  fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
  if (fd < 0) {
```

```
    perror("socket");
    goto socket_failed;
  }

  strncpy(ifr.ifr_name, name, IFNAMSIZ);
  if (ioctl(fd, SIOCGIFINDEX, &ifr) < 0) {
    perror("ioctl");
    goto get_index_failed;
  }

  sa.sll_family = AF_PACKET;
  sa.sll_ifindex = ifr.ifr_ifindex;
  sa.sll_protocol = htons(ETH_P_ALL);
  if (bind(fd, (struct sockaddr *) &sa, sizeof(sa)) < 0) {
    perror("bind");
    goto bind_failed;
  }

  result = 1;
  goto out;


bind_failed:
get_index_failed:
  close(fd);
socket_failed:
  result = 0;
out:
  return result;
}


int start_service(__u16 s_id, __u32 s_freq, __u16 s_data, char *s_desc,
                  int desc_len, int s_is_src) {

  struct ifreq ifr;
  struct ioctl_service_desc *my_desc;
  my_desc = (struct ioctl_service_desc *)
              malloc(sizeof (struct ioctl_service_desc));
  if (my_desc == NULL) {
    perror("service registration");
```

```
    return 0;
  }

  // Fill in the service descripton.
  my_desc->id = s_id;
  my_desc->is_src = s_is_src;
  my_desc->freq = s_freq;
  my_desc->size = s_data;
  my_desc->desc_len = desc_len;
  printf("max_desc_len = %d\n", MAX_DESC_LEN);
  memcpy(my_desc->desc, s_desc, (desc_len < MAX_DESC_LEN) ?
            desc_len : MAX_DESC_LEN);

  // Pass it to the device
  strncpy(ifr.ifr_name, name, IFNAMSIZ);
  ifr.ifr_data = (void *) my_desc;
  if (ioctl(fd, SERVICE_REGISTRATION, &ifr)) {
    perror("ioctl");
    return 0;
  }
  else printf("Successful registration of service %u!\n", my_desc->id);
  free(my_desc);

  return 1;
}


int stop_service(__u16 s_id) {

  struct ifreq ifr;
  struct ioctl_service_desc *my_desc;


  my_desc = (struct ioctl_service_desc *)
               malloc(sizeof (struct ioctl_service_desc));
  if (my_desc == NULL) {
    perror("service registration");
    return 0;
  }

  // Fill in the service descripton.
```

```
  memset(my_desc, 0, sizeof(struct ioctl_service_desc));
  my_desc->id = s_id;

  // Pass it to the device
  strncpy(ifr.ifr_name, name, IFNAMSIZ);
  ifr.ifr_data = (void *) my_desc;
  if (ioctl(fd, SERVICE_DELETION, &ifr)) {
    perror("ioctl");
    return 0;
  }
  else printf("Successful removal of service %u!\n", my_desc->id);
  free(my_desc);

  return 1;
}


void write_msg(__u16 service_id, __u8 *msg, unsigned int msg_len,
               int eager) {

  int copied, i, messages = 10;
  struct ifreq ifr;
  struct ioctl_rt_msg *irm;


  irm = (struct ioctl_rt_msg *) malloc(sizeof (struct ioctl_rt_msg));
  if (irm == NULL) {
    perror("write message");
    return;
  }

  // Prepare a write request
  do {
    memset(irm->msg, 0, ETH_MTU);
    if (eager) {
      for (i = 0; i < 10; i++) {
        irm->msg[i] = i;
      }
      irm->msg_len = 10;
      messages--;
    }
```

```
    else {
      memcpy(irm->msg, msg, msg_len);
      irm->msg_len = msg_len;
    }

    irm->id = service_id;

    // Pass it to the device
    strncpy(ifr.ifr_name, name, IFNAMSIZ);
    ifr.ifr_data = (void *) irm;
    ioctl(fd, RT_WRITE, &ifr);
  } while (eager && messages > 0);
  free(irm);
}


void read_msg(__u16 service_id, int eager) {

  int i;
  struct ifreq ifr;
  struct ioctl_rt_msg *irm;
  struct timespec tp, tp2;


  irm = (struct ioctl_rt_msg *) malloc(sizeof (struct ioctl_rt_msg));
  if (irm == NULL) {
    perror("read msg");
    return;
  }

  // Prepare a read petition
  memset(irm->msg, 0, ETH_MTU);
  irm->msg_len = 0;
  irm->id = service_id;

  // Pass it to the device
  strncpy(ifr.ifr_name, name, IFNAMSIZ);
  ifr.ifr_data = (void *) irm;
  i = 0;
  while (eager) {
    clock_gettime(CLOCK_MONOTONIC, &tp);
```

```
    ioctl(fd, RT_READ, &ifr);
    if (irm->successful) {
      clock_gettime(CLOCK_MONOTONIC, &tp2);
      printf("-------------------------\n");
      printf("[%lf]Received a message from service %u\n",
              ((double) tp2.tv_nsec) - ((double) tp.tv_nsec), irm->id);
      for (i = 0; i < irm->msg_len; i++)
        printf("%u", irm->msg[i]);
      printf(".\n-------------------------");
      i++;
      eager = (i == 10) ? 0 : 1;
    }
  }
  free(irm);
}
```

Apéndice C

El sistema LabComm

C.1. Componentes del sistema LabComm

El sistema LabComm se compone de tres elementos:

- Un **lenguaje de declaración de datos**, que permite declarar tipos de datos simples y compuestos de la misma manera en que lo hace un lenguaje de programación (ver Sección C.2). Este lenguaje se emplea para especificar los tipos de datos que se van a intercambiar durante la comunicación. A la especificación de los tipos de datos que se van a transmitir se la denomina "protocolo".
- Un **protocolo binario**, que especifica cómo han de traducirse los tipos de datos simples a secuencias de bytes y cómo recuperar los datos originales a partir de estas secuencias de bytes (serialización). Puesto que los tipos de datos compuestos se obtienen a partir de la agregación de tipos de datos simples, esta especificación también permite serializar estructuras de datos compuestas.
- Un **compilador** que genera rutinas de serialización en diversos lenguajes de programación[1] a partir de un protocolo. Estas rutinas realizan la conversión de los tipos de datos (tal y como están representados en esos lenguajes de programación) a secuencias de bytes y viceversa.

C.2. El lenguaje LabComm

Esta sección detalla los tipos de datos que ofrece el lenguaje LabComm para especificar un protocolo de comunicación.

[1] Java, C, C# y Python están soportados

Primitive types

```
  sample boolean a_boolean;
  sample byte a_byte;
  sample short a_short;
  sample int an_int;
  sample long a_long;
  sample float a_float;
  sample double a_double;
  sample string a_string;
```

Arrays

```
  sample int fixed_array[3];
  sample int variable_array[_];
  sample int fixed_array_of_array[3][4];
  sample int fixed_rectangular_array[3, 4];
  sample int variable_array_of_array[_][_];
  sample int variable_rectangular_array[_, _];
```

Structures

```
  sample struct {
    int an_int_field;
    double a_double_field;
  } a_struct;
```

User defined types

```
  typedef struct {
    int field_1;
    byte field_2;
  } user_type[_];
  sample user_type a_user_type_instance;
  sample user_type another_user_type_instance;
```

C.3. Ejemplos de protocolos de comunicación

Este apéndice contiene dos ejemplos de protocolos LabComm.

C.3.1. Ejemplo 1

```
// Parameters
  sample boolean a_boolean;
  sample byte a_byte;
  sample short a_short;
  sample int an_int;
  sample long a_long;
  sample double a_double;
  sample string a_string;
  sample int an_int_array[3];
  sample double a_double_array[5];
  typedef struct {
    int an_int_field;
    double a_double_field;
  } user_type;
  sample user_type a_user_type_instance;
```

C.3.2. Ejemplo 2

```
// Parameters
  sample double f_switch;
  sample double sensor_frame[7];

// Logs
  sample double log_velRef[6];
  sample double force_nocomp[6];
  sample double log_f_switch;
  sample double velref[6];
  sample double u[6];
  sample double r[24];
  sample double overload;
  sample double trigger;
```

```
sample double y[36];
sample double state;
sample double force[6];
sample double q_dot[6];
```

Apéndice D

A LabComm tutorial

D.1. Introduction

LabComm is a system devised to automatically implement the necessary mechanisms to support strongly-typed message-based communication between applications. These mechanisms consist of message encoding/decoding routines (from now on, marshalling routines) for generating and parsing typed messages. As the messages are typed, the receiving applications can always interpret the data that they contain correctly.

The user has to feed LabComm with a specification of the data types which are to be transmitted. This specification (to which we will refer as a *protocol*) will be used by LabComm to generate marshalling routines for such data types in several programming languages. Therefore, LabComm allows to effortlessly communicate applications written in different languages whilst achieving message content abstraction and strongly-typed communication in a user transparent way.

The LabComm system consists of three elements:

- A **language for defining simple and structured data types** (see Section C.2). This language will be used to specify the types of data that are to be exchanged during the communication (the protocol).

- A **binary protocol**, which specifies how should the simple data types be marshalled and demarshalled. Since structured data types are obtained by putting simple ones together, this marshalling specification also applies to them.

- A **compiler** that generates marshalling routines for the data specified in the protocol in a variety of languages[1]. These routines allow to transform the data (as it was represented within the programming language) into a series of bytes and viceversa.

A brief description of all the elements involved in LabComm-based data transmission (see Figure D.1) follows:

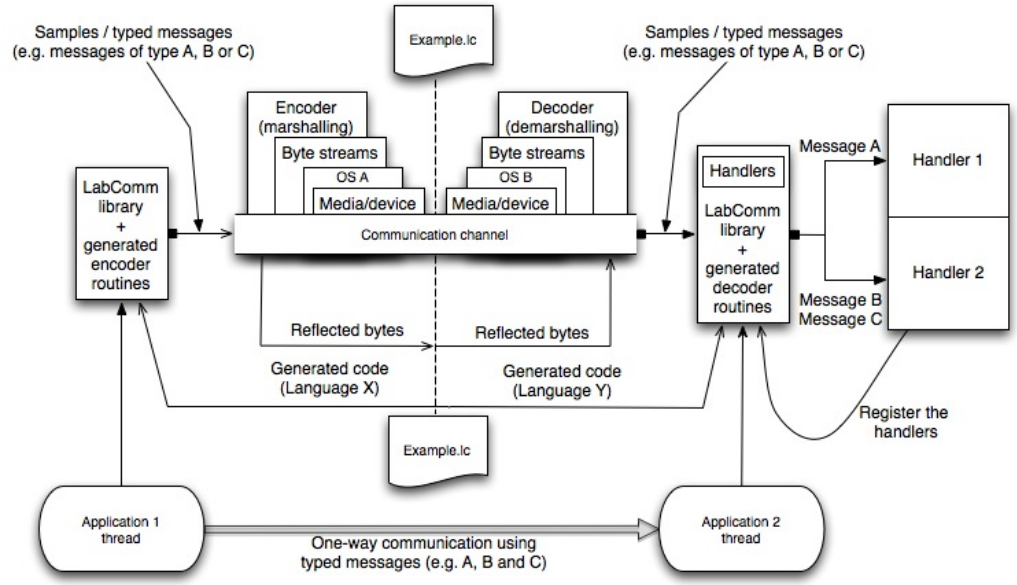

Figura D.1: Communication over a data stream using LabComm.

- The lowest part of Figure D.1 offers an overview of the communication which is taking place: two **application threads**, acting as a producer and a consumer, are communicating over a channel using different types of messages (i.e. A, B and C).

- Even if the producer and the consumer are written in different programming languages (**languages X and Y**, in this example), they will be able to communicate flawlessly. The reason is that their communication is based on sequences of bytes that they can generate and parse because they share the LabComm protocol which defines the exchanged data. This protocol is contained in the **example.lc** file, with which the LabComm compiler is fed to generate the **marshalling routines**.

[1]Code generation for Java, C, C# and Python is supported.

- The **marshalling routines** are used at each end of the communication channel. Our producer and consumer communicate using typed messages (samples), which are marshalled prior to their transmission through the **communication channel** and demarshalled upon reception. Thus the **encoder routines** will transform the data as it was represented in **language X** into bytes, which will travel through the channel. Upon arrival to the receiving end of the channel, the **decoder routines** will demarshall the bytes received into messages which can be understood by the receiving application, written in **language Y**.

- Finally, all received data has to be handled in a proper way. This is done by using **handlers** for each type of sample. These handlers are to be written by the programmer and registered to the LabComm library routines *before* the transmission takes place, so that the handling of the received messages is done in an automatic way.

- The **LabComm libraries** provide support for the generated routines and the application written by the programmer. More on this later.

- And finally, communication should be held via a **communication channel**, which will be any kind of data stream used for one-way communication (namely a file, a socket, a pipe, etc).

D.2. Step one: Generate the marshalling routines

Important: There will be references to the code used in this tutorial. The code is available at `http://vm15.cs.lth.se:2323/labCommDemo/Client.html`.

In order to generate the encoder/decoder routines, we will input a protocol definition to the LabComm compiler.

To generate such routines in Java, you should first create a directory to place the routines and then run the LabComm compiler addressing that directory as the output for the files.

```
mkdir javaRoutines
java -jar ./LabComm.jar --java=javaRoutines example.lc
```

This will generate the following classes[1] in the directory javaRoutines: **a.java**, **b.java**, **SimpleSample.java** and **SimpleType.java**.

The files **a.java**, **b.java** and **SimpleSample.java** have practically the same content:

- An interface specification for their handlers.
- Methods to register a *LabCommDecoder* (to decode that type of data).
- Methods to register a *LabCommEncoder* (to encode that type of data).

As classes **a** and **b** are related to primitive data types (namely booleans and bytes) they do not need any other routines to encode or decode their data than those already provided in the LabComm libraries (remember the previously defined binary protocol). Nevertheless, the file "example.lc" defines a data type which is an aggregate type (a struct). Since there are no routines already implemented to encode/decode structs, the LabComm compiler will create the class **SimpleType** to provide with the proper encoder/decoder routines. However, the programmer will never work directly with this class, but with the **SimpleSample** class.

You can generate all these classes feeding the protocol provided in Appendix D.4 to the LabComm system, available at `http://vm15.cs.lth.se:2323/labCommDemo/labcomm2.zip`.

D.3. Step two: Use the marshalling routines

If you are planning to work with Eclipse, it might be a good idea to separate the LabComm libraries and generated routines from your application classes. You could do this by creating two different packages, one for your own application classes and the other one for the automatically created routines. Then, you can import these routines from your application.

The generated routines will have a number of import clauses, for example:

```
import se.lth.control.labcomm.LabCommDecoder;
import se.lth.control.labcomm.LabCommDispatcher;
import se.lth.control.labcomm.LabCommEncoder;
```

[1] Please note that this is what would happen if you used the protocol definition contained in Appendix D.4. This is the file we assume to be using from now on.

```
import se.lth.control.labcomm.LabCommHandler;
import se.lth.control.labcomm.LabCommSample;
```

In order to help Eclipse find these packages, you should tell it to import the library folder as a source folder. You can do this by navigating through the folder which contains all LabComm material, entering the "lib" folder and choosing the "java" folder inside. Check Figure D.2 to see how the examples in this tutorial have been structured.

There are several elements surrounded by different coloured boxes in Figure D.2. Elements in the blue box are part of the LabComm library. The red and green boxes surround not only the applications written by the programmer himself but also the routines generated by LabComm. Red box elements belong to the first example and green elements to the second one.

We will now describe how elements of data type *a* (in Example 1) are encoded and decoded by the Sender and Receiver applications respectively. These two programs will communicate using a File as a stream.
Please note that the code implementing the agents named below is available at `http://vm15.cs.lth.se:2323/labCommDemo/Client.html`.

The **Sender** will first create a FileOutputStream directed to a File. Then, it will associate it to an object of the class *LabCommEncoderChannel*, which provides an implementation for the interface *LabCommEncoder*. The next step is to register this channel to **any** classes which we want to encode (in this case class *a*) and invoke the static method *encode* in those classes whenever we want to encode one of their objects.

The **Receiver** will first create a FileInputStream to read from a File. Then, it will associate it to an object of the class *LabCommDecoderChannel*, which provides an implementation for the interface *LabCommDecoder*. The next step is to register this channel to **any** classes which we want to decode (in this case class *a*) together with a specific handler for each class. Those handlers have to implement the interface specified in each class that we want to decode. Finally, whenever we want to decode the data on the stream handled by the **LabCommDecoderChannel** we will simply call the *run* method in the class **LabCommDecoderChannel** and it will decode any data in the stream. Since we already registered to this channel all handlers needed to decode the data which is in the stream, this process is performed automatically.

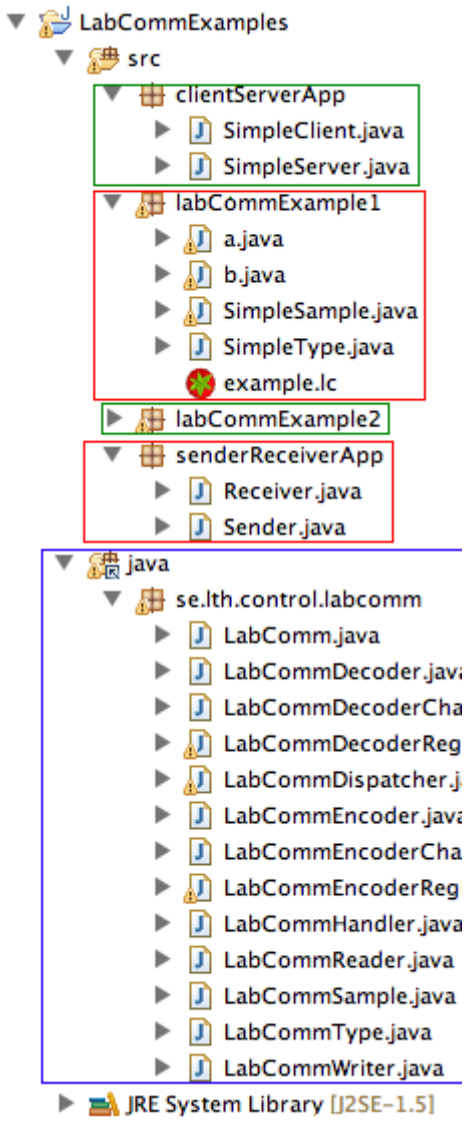

Figura D.2: Organization of the packages in the tutorial.

Another example has been written to illustrate how LabComm ought to be used in a client-server architecture (i.e. the stream would now be a socket). For that example, the encoder/decoder routines would be generated in the same way we did before: create a directory to host the files, run the LabComm compiler on the protocol file and writing an application which uses the routines as explained before.
The applications for this new example have been written in a different way; wrapping all registrations of the classes to encode/decode (and their hand-

lers) to the communication channel into inner classes, namely **MyEncoder** and **MyDecoder**.

Final observation: whenever the method *run* of the **LabCommDecoderChannel** is called, a careful exception handling is required. The reason is that this method will read from a stream until the end of that stream is reached, which will cause a *EOFException* (End Of File Exception) to be thrown. A catch clause which does nothing is required for this exception, since it will always be thrown at the end of a succesful communication.

D.4. Protocol definition for Example 1

```
typedef struct {
    int x;
    int y;
    string name;
    float data[2][_,_];
} SimpleType;

sample SimpleType SimpleSample;
sample boolean a;
sample byte b;
```

Bibliografía

[1] Lund Institute of Technology, Lund University, *A LabComm wiki*. http://torvalds.cs.lth.se/moin/LabComm.

[2] A. Blomdell, K.-E. Arzen, and A. Martinsson, *ThrottleNet: Hard real-time communication over switched Ethernet*.

[3] "Serialización." Wikipedia. http://es.wikipedia.org/wiki/Serializaci%C3%B3n.

[4] "Java applets." Sun Developer Network. http://java.sun.com/applets/.

[5] "The java tutorials: Applets." http://download.oracle.com/javase/tutorial/deployment/applet/.

[6] "What applets can and cannot do." http://download.oracle.com/javase/tutorial/deployment/applet/security.html.

[7] "The java tutorials: Creating a gui using swing." http://download.oracle.com/javase/tutorial/uiswing/.

[8] "Rtnet - hard real-time networking for real-time linux." RTnet Development Team. www.rtnet.org/.

[9] "Xenomai - real-time framework for linux." http://www.xenomai.org/index.php/Main_Page.

[10] "Real-time application interface for linux." https://www.rtai.org/.

[11] "Flow control." Wikipedia, the free Encyclopedia. http://en.wikipedia.org/wiki/Flow_control.

[12] "Vxworks: real-time operating system for embedded systems." http://en.wikipedia.org/wiki/VxWorks.

[13] A. Rubini, J. Corbet, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, 2005.

[14] "Monitors (synchronization)." http://en.wikipedia.org/wiki/Monitor_%28synchronization%29.

[15] "The java tutorials: Objectstreams." http://download.oracle.com/javase/tutorial/essential/io/objectstreams.html.

[16] "Serializable (java 2 platform se v1.4.2)." Oracle. http://download.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html.

[17] T. Greanier, *Java Serialization*. Sun Developer Network. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[18] "The java tutorials: Arrays." http://download.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

[19] "The instanceof keyword (java)." http://www.java2s.com/Tutorial/Java/0060__Operators/TheinstanceofKeyword.htm.

[20] "Localizador uniforme de recursos (url)." Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Localizador_uniforme_de_recursos.

[21] "Java classloaders." Oracle. http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html.

[22] J. Jenkov, "Java reflection: Dynamic class loading and reloading." http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html.

[23] "Java reflection." Sun Developer Network. http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[24] "Address resolution protocol (arp)." Wikipedia, the free Encyclopedia. http://en.wikipedia.org/wiki/Address_Resolution_Protocol.

[25] The Linux Foundation, *Linux bridges*. www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[26] "Semaphores (synchronization)." Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Semaphore_%28programming%29.

[27] "Spinlocks (synchronization)." Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Spinlock.

[28] G. Mathiason and M. Amirijoo, "Real-time communication through a distributed resource reservation approach," Master's thesis, University of Skoevde, 2004.

[29] "Universal tun/tap driver." http://en.wikipedia.org/wiki/TUN/TAP.

[30] "Universal tun/tap driver: Faq." http://vtun.sourceforge.net/tun/faq.html.

[31] "Ioctl: Input output control." Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Ioctl.

[32] "Ethertype." Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/EtherType.

[33] "Java web start." Oracle. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136112.html.