UNIVERSIDAD DE COSTA RICA

SISTEMA DE ESTUDIOS DE POSGRADO

D-EXPLORER: HERRAMIENTAS LIVIANAS PARA LA EXPLORACIÓN DE SISTEMAS DISTRIBUIDOS EN AMBIENTES ACADÉMICOS

Tesis sometida a la consideración de la Comisión del Programa de Estudios de Posgrado en Computación e Informática para optar al grado de Magister Scientiae en Computación e Informática

MAN SAI ACÓN CHAN

Ciudad Universitaria Rodrigo Facio, Costa Rica

2009

Al final de este largo camino, quedo muy agradecido con las personas que me guiaron y me ayudaron, en especial con Gabriela, Ronald, Edgar y Francisco, y muy agradecido también porque llegué a buen puerto.

Un todo para una vida y un poquito para cada día.
Así vamos dejando nuestras huellas en esta vida.
Hoy alcancé una meta, hoy es un buen día.
Esperemos que la vida nos depare más alegrías.

"Esta tesis fue aceptada por la Comisión del Programa de Estudios de Posgrado en Computación e Informática de la Universidad de Costa Rica, como requisito parcial para optar al grado de Magister Scientiae en Computación e Informática."

PhD Gabriela Marín Raventós
Decana del Sistema de Estudios de Posgrado

PhD José Ronald Argüello Venegas
Director de Tesis

PhD Gabriela Barrantes Slesarieva
Asesora

MSc Edgar Casasola Murillo
Asesor

Magister Lidia Arévalo Bravo
Directora del Programa de Posgrado
en Computación e Informática

Man Sai Acón Chan
Candidato

Índice

Resumen

En este trabajo de investigación se desarrolló y se validó *D-Explorer*, un conjunto de herramientas que facilitan la implementación de aplicaciones distribuidas en ambientes académicos. Inicialmente, se identificaron y se desarrollaron las funcionalidades y los servicios que deben tener *D-Explorer*. Posteriormente, se verificó el funcionamiento correcto de estas herramientas mediante el diseño y la ejecución de una serie de pruebas concurrentes. Finalmente, estudiantes de la Maestría en Computación e Informática de la UCR usaron *D-Explorer* durante el segundo semestre del 2008 y lograron implementar con éxito varias tareas programadas.

D-Explorer es de código abierto, tiene bajos requerimientos de hardware y de software e incluye tutoriales que explican su uso. Los componentes principales de *D-Explorer* son: el canal de servicios y los servicios de descubrimiento, directorio, notificaciones, sincronización y archivos.

Con el uso de *D-Explorer*, se espera facilitar el desarrollo de aplicaciones distribuidas y lograr que estudiantes, tesiarios o investigadores puedan implementar rápidamente tareas programadas y proyectos cortos que involucren conceptos de sistemas distribuidos o aplicaciones distribuidas con diversos fines como estudiar algoritmos distribuidos, evaluar el rendimiento o la correctitud de nuevos algoritmos.

Lista de tablas

Lista de figuras

1 Introducción

Este trabajo consiste en el desarrollo y la validación de *D-Explorer*, un conjunto de herramientas de desarrollo de aplicaciones distribuidas orientadas a ambientes académicos, y con las cuales se espera llenar una necesidad práctica en el estudio y la investigación de sistemas distribuidos. Con el uso de *D-Explorer*, se espera reducir la complejidad de desarrollar aplicaciones distribuidas y permitir que una persona se enfoque en la solución del problema distribuido y no en manejar la comunicación, la serialización de datos, la sincronización de acceso a recursos remotos y otros elementos que no están relacionados con el problema distribuido pero son necesarios para el programa.

Los servicios y utilidades más importantes de *D-Explorer* son: el canal de servicios, un componente que ofrece servicios de comunicación entre procesos, el servicio de descubrimiento, un mecanismo para localizar los nodos del sistema distribuido, el servicio de directorio, un componente que mantiene y publica la lista de servicios disponibles del sistema distribuido, el servicio de notificaciones, un servicio para enviar mensajes asincrónicamente entre procesos, el servicio de sincronización, un mecanismo para sincronizar la ejecución de procesos y el servicio de archivos, un componente para accesar archivos almacenados en otras computadoras.

En el proceso de validación de *D-Explorer*, primero se diseñaron y se ejecutaron una serie de pruebas concurrentes para asegurar el funcionamiento correcto de los servicios y utilidades mencionados. Posteriormente, estudiantes de la Maestría en Computación e Informática de la UCR usaron estas herramientas para implementar tareas programadas durante el segundo semestre del 2008 con resultados positivos.

Se usó Java[1] como lenguaje de desarrollo, un lenguaje orientado a objetos con facilidades para la programación en redes (UDP[2], TCP[3]), hilos y serialización de objetos, y para el cual existen varias herramientas de desarrollo gratuitas.

D-Explorer es de código abierto, se puede usar en computadoras con procesadores de 800 Mhz, 128 Mb en RAM y ha sido usado en los sistemas operativos Windows y Linux.

El uso de UDP o TCP sobre IP[4] deja abierta la posibilidad de usar *D-Explorer* en aplicaciones distribuidas sobre Internet, pero para el presente proyecto se limitó su alcance a redes de área local (LAN), pues el objetivo principal es demostrar la correctitud y la factibilidad de las herramientas. Los aspectos relacionados con la eficiencia, el rendimiento y la escalabilidad en Internet quedan fuera del alcance de esta investigación. De igual manera se limitó su alcance en materia de seguridad y fallos, *D-Explorer* es susceptible a ataques y fallos en la red, en las computadoras o en los componentes de software.

1.1 Justificación

Con la proliferación de redes de computadoras y de Internet, la existencia de nodos terminales y dispositivos móviles cada vez más poderosos, y de diferentes sistemas operativos como Windows[5], Linux[6], Palm OS[7] o Mac OS[8], se crea un ambiente propicio para el desarrollo de sistemas distribuidos. La factibilidad y la utilidad de estos sistemas, incluso en una escala mundial, han quedado demostradas con la puesta en marcha de proyectos globales como LimeWire[9], un sistema para compartir archivos, o del programa SETI@home[10], un sistema distribuido para el análisis de frecuencias de radio con el objetivo de descubrir comunicaciones entre formas de vida extraterrestres. El desarrollo de estos sistemas presenta retos diferentes para los practicantes de computación. Es necesario desarrollar habilidades en programación de redes, de hilos, de diferentes esquemas de sincronización de recursos, habilidades para captar problemas distribuidos y plantear soluciones distribuidas, que involucren protocolos entre componentes, intercambio de mensajes, cooperación y coordinación de múltiples nodos.

Sistemas distribuidos profesionales han sido desarrollados con éxito usando abstracciones como los *sockets*[11], la invocación remota de procedimientos, los objetos distribuidos y la cola de mensajes.

El *socket* es un punto unívocamente identificado de una red desde el cual se puede enviar bytes a otros *sockets* o recibir bytes de otros *sockets*. Los procesos usan *sockets* para intercambiar mensajes entre ellos.

La invocación remota de procedimientos (RPC[12]) permite que un proceso en un nodo

invoque un procedimiento alojado en otro nodo. Presenta una semántica idéntica al de un llamado a procedimiento (local) y facilita la interacción entre los componentes del sistema distribuido. Cuando un procedimiento remoto es invocado, el ambiente local es suspendido, los parámetros son transferidos, a través de la red, al ambiente remoto en donde se ejecuta el procedimiento remoto. Cuando este finaliza, se transfiere el resultado del procedimiento al ambiente local, en donde se reinicia la ejecución del ambiente local como si retornara de un llamado a un procedimiento local.

Los objetos distribuidos permiten que un proceso en un nodo pueda instanciar objetos alojados en otro nodo, o invocar métodos del objeto remoto instanciado. Dos herramientas que implementan esta abstracción son CORBA[13] y RMI[14] de Java.

La cola de mensajes es una abstracción que presenta un mecanismo de comunicación asíncrona y confiable entre los componentes distribuidos: un conjunto de procesos depositan mensajes en una cola, otro conjunto de procesos leen mensajes de la cola, los procesos no interactúan directamente entre ellos, y las operaciones para depositar y leer mensajes ocurren de manera asíncrona. Algunas herramientas que implementan esta abstracción son WebSphere MQ[15] de IBM y MSMQ[16] de Microsoft.

Las herramientas mencionadas, como el *socket*, RMI, etc, también han sido usados en ambientes académicos. Sin embargo, en ambientes académicos se desea elaborar prototipos de sistemas, pruebas de concepto de algoritmos o proyectos y tareas cortas de cursos semestrales, usualmente acotados por cortos plazos de entrega. En estos casos, estas herramientas presentan varios inconvenientes que se detallan a continuación:

- Carecen de abstracciones o servicios de más alto nivel. Estas herramientas permiten intercambiar mensajes o invocar procedimientos remotos pero no agregan funcionalidad adicional. Sería útil contar con algunos servicios de más alto nivel como semáforos, directorios, notificaciones, archivos distribuidos, etc.
- Es posible que se tenga que configurar el ambiente para usar las herramientas. Por ejemplo, las colas de mensajes se deben instalar y configurar en cada nodo.
- Es posible que se tenga que aprender más de una herramienta. Mientras que RPC y

objetos distribuidos facilitan interacciones en donde los clientes solicitan servicios a los servidores, las colas de mensajes facilitan el intercambio de mensajes de manera asíncrona entre componentes distribuidos. El uso de más de una herramienta en el desarrollo del sistema distribuido también eleva su complejidad.

- Es posible que se tenga que aprender otras herramientas complementarias. En el caso de CORBA y de RPC, es necesario aprender un lenguaje para la definición de interfaces como CORBA IDL[17] u otros.
- Las herramientas descritas contienen muchas funciones que no se necesitan en ambientes académicos. A diferencia de proyectos profesionales o comerciales, en ambientes académicos se ocupa un subconjunto básico de la funcionalidad de las herramientas. Es preferible un API enfocado a la funcionalidad básica que facilite su aprendizaje y permita su uso en un plazo corto de tiempo.

Los inconvenientes descritos indican la necesidad de herramientas de desarrollo de sistemas distribuidos orientadas a ambientes académicos y constituyen la justificación principal de la presente investigación. Estas herramientas beneficiarían a las personas que estudian o investigan sistemas distribuidos pues se espera que las herramientas simplifiquen el desarrollo de aplicaciones distribuidas y permitan que una persona se enfoque en el problema distribuido en sí y no en problemas secundarios como implementar mecanismos de comunicación o de sincronización. Además, como resultado de esta investigación se espera obtener los siguientes beneficios en el ámbito educativo y académico:

1. Exponer conocimientos internos de implementaciones de sistemas distribuidos. A diferencia del uso de algún estándar como CORBA o RPC - en donde la tecnología subyacente es una caja negra y el enfoque es usar esa tecnología para implementar la aplicación - el proceso de desarrollar estas herramientas expone detalles internos de la caja negra, la arquitectura del *middleware*[1] y su interacción con los servicios. Este tipo de conocimientos facilita el entendimiento y permite un mejor uso de tecnologías con una arquitectura similar, pero más complejas.

---

1 Capa de software que encapsula y esconde la complejidad de la comunicación entre nodos y provee una interfaz de programación conveniente para la comunicación.

2. Crear condiciones para un mejor aprendizaje de sistemas distribuidos. Las herramientas permiten proponer tareas programadas y proyectos más elaborados o construir rápidamente prototipos para pruebas de concepto.

En esta sección se presentaron la justificación y los beneficios del presente trabajo de investigación. En las siguientes secciones se presentan el objetivo general y los objetivos específicos del mismo.

1.2 Objetivo general

Identificar, diseñar e implementar un conjunto de herramientas que faciliten el desarrollo de prototipos de sistemas distribuidos, pruebas de concepto de algoritmos distribuidos, y proyectos y tareas en cursos semestrales de sistemas distribuidos en ambientes académicos.

1.3 Objetivos específicos

Los objetivos específicos son los siguientes:

1. Identificar un conjunto de servicios y funcionalidades que faciliten la exploración de sistemas distribuidos.
2. Definir abstracciones y operaciones que provean dichas funcionalidades, teniendo en cuenta que se usarán en ambientes educativos.
3. Diseñar e implementar un componente distribuido que cumpla con las características definidas para cada servicio.
4. Verificar la correctitud de los componentes distribuido mediante un conjunto de pruebas que están definidas en la metodologia.
5. Implementar un prototipo de aplicación distribuida o de algoritmo distribuido para verificar la funcionalidad de las herramientas.
6. Preparar material de apoyo para facilitar el aprendizaje de las herramientas.
7. Verificar la utilidad de las herramientas en ambientes académicos.

1.4 Trabajos relacionados

En el desarrollo de sistemas distribuidos se usan herramientas como *sockets*, colas de mensajes y objetos distribuidos entre otros. Estas herramientas son útiles para el desarrollo de sistemas distribuidos profesionales pero presentan inconvenientes en el ambiente académico.

En la actualidad no se ha encontrado un conjunto de herramientas para ambientes académicos. Un trabajo reciente es DCF[18] (*Distributed Component Framework*), un marco de trabajo sencillo, para la creación y distribución de sistemas basados en componentes. Este trabajo menciona que la madurez y la riqueza del conjunto de características estándares de los marcos de trabajo y *middlewares* comerciales, agrega complejidad innecesaria cuando se trata de explorar por primera vez este tipo de desarrollo de sistemas. El propósito de este marco de trabajo (DCF) es proveer a los novatos un conjunto de características suficientes que les permitan entender la naturaleza de sistemas distribuidos sin el costo y la complejidad asociados a estos sistemas. Sus objetivos son: a) soportar comunicación asíncrona, basada en mensaje e independiente de la localización o de la ubicación, b) usar técnicas de programación sencillas y depender únicamente de *Java Standard Edition*, y c) minimizar la cantidad de código requerido para crear un componente. Las clases principales de DCF son: *Message*, *Behavior* y *Component*. La clase *Message* es la clase base de los mensajes. La clase *Behavior* es la clase base para definir lógica para procesar mensajes. Subclases de *Behavior* deben implementar el método *interpretMessage()* con la lógica para procesar un mensaje y devolver resultados. La clase *Component* sirve de contenedor de *Behavior* y proveer los mecanismos para enviar y recibir mensajes. El marco de trabajo contiene un *middleware* sencillo basado en mensajes usando *sockets* y un servicio de localización para proveer transparencia de localización. En futuras versiones se espera implementar Bluetooth[19], el cual es un mecanismos de transporte[18]. A diferencia de DFC, el cual soporta únicamente un modelo de interacción entre procesos encapsulado en *Behavior*, *D-Explorer* implementará varios mecanismos de comunicación como el conector de servicio, el componente de servicios y el servicio de notificaciones, y ofrecerá otros servicios adicionales como semáforos, directorio,

descubrimiento, archivos remotos, etc.

Después de explicar la justificación y los objetivos de la investigación, en el siguiente capítulo se explicará el marco teórico que sustenta el proyecto. El resto del documento está organizado de la siguiente manera: el capítulo 3 describe las funcionalidades y servicios de *D-Explorer*, en el capítulo 4 se presentan detalles de implementación de las herramientas, en el capítulo 5 se describe el material de apoyo para aprender a usar *D-Explorer*, en los capítulos 6 y 7 se presenta el proceso de validación de las herramientas. En el capítulo 8 se se explican las conclusiones y en el último capítulo se sugieren trabajos futuros a realizar.

2 Marco teórico

En este capítulo se introduce el concepto de sistema distribuido y sus características principales, y luego se explica la arquitectura de sistemas distribuidos. A continuación se explica el concepto de sistema distribuido.

2.1 Sistema distribuido

Un sistema distribuido es un sistema, cuyos componentes están localizados en diferentes computadoras enlazadas por una red. Estos componentes se comunican y coordinan sus acciones únicamente mediante el intercambio de mensajes [20]. La motivación principal para desarrollar sistemas distribuidos es compartir recursos de hardware, como impresoras o discos, o de software, como archivos, bases de datos, objetos, o lógica en forma de código remoto o de código móvil. El código remoto se ejecuta en el nodo remoto, y puede devolver un mensaje con el resultado. El código móvil, por el contrario, se transfiere al nodo local y se ejecuta localmente. En las siguientes secciones se describen las características, los modelos de interacción y los retos de desarrollar sistemas distribuidos.

2.1.1 Características

Un sistema distribuido presenta las siguientes características: es concurrente, no tiene un reloj global y presenta fallos independientes[20].

La concurrencia se refiere a que el sistema distribuido tiene componentes corriendo al mismo tiempo en diferentes computadoras. Esto implica la necesidad de coordinar el acceso a recursos entre los nodos, pero también facilita el crecimiento del sistema, con la incorporación de más recursos como computadoras y componentes de software.

La ausencia de un reloj global se debe a que cada nodo tiene su propio reloj físico, lo cual dificulta el ordenamiento de eventos cronológicos ocurridos en diferentes nodos. Los relojes físicos se pueden sincronizar con algoritmos que intercambian mensajes como Cristian[21] y Berkeley[22], pero siempre queda un margen de error pues estos algoritmos se ven afectados por los retrasos arbitrarios de la red, y por la existencia de otros procesos

que están ejecutándose en la misma computadora.

La presencia de fallos independientes ocurre debido a que el sistema está compuesto por componentes, computadores y enlaces de comunicación entre las computadoras, y estos elementos pueden fallar independientemente de los demás. A diferencia de un sistema centralizado compuesto por un único programa, los otros elementos del sistema distribuido pueden seguir funcionando. Esto introduce una faceta adicional en el desarrollo de sistemas distribuidos, pues se debe considerar las consecuencias de posibles fallos y anticipar acciones que permitan corregir el fallo o mitigar sus efectos. Por ejemplo, si se tienen varios nodos que ofrecen un mismo servicio, se puede mitigar los efectos de la caída de uno de los nodos enrutando las solicitudes de servicio hacia los otros nodos.

Después de explicar las características de un sistema distribuido, en la siguiente sección se presentan los modelos de interacción de un sistema distribuido.

2.1.2 Modelos de interacción

En los sistemas distribuidos hay dos modelos básicos de interacción entre nodos: el modelo cliente-servidor y el modelo punto a punto[20].

En el modelo cliente-servidor, los procesos clientes accesan recursos compartidos interactuando con procesos servidores que administran los recursos, ambos procesos pueden estar en la misma computadora o residir en computadoras separadas. El cliente envía un mensaje con la solicitud al servidor y este le devuelve un mensaje con la respuesta. Este modelo tiene el defecto de que el servidor se convierte en un punto de fallo y es poco escalable. Si el servidor falla, el sistema deja de operar. Si la demanda aumenta significativamente, el servidor deja de atender algunas solicitudes.

En el modelo punto a punto, los procesos cumplen funciones similares, interactuando cooperativamente sin distinción entre clientes, servidores o computadoras. El modelo es más escalable y distribuye mejor la carga de trabajo. Como todos los nodos cumplen funciones similares, es fácil incorporar nuevos nodos para atender el aumento en la demanda o para sustituir nodos caidos. La infraestructura actual en equipos y comunicación favorece este tipo de modelo: los equipos tienen alta capacidad de procesamiento y de

almacenamiento, y es factible contratar canales de banda ancha para conectar estos equipos.

Existen variantes de los modelos cliente-servidor y punto a punto que incorporan la replicación de servidores, clientes con *caches*, clientes o puntos móviles, etc.

Seguidamente se explican los retos que surgen en el desarrollo de sistemas distribuido.

2.1.3 Retos

A diferencia de los sistemas que corren en una computadora, el desarrollo de sistemas distribuidos presenta otros retos y problemas como: heterogeneidad, apertura, seguridad, escalabilidad, manejo de fallos, concurrencia y transparencia[20].

Los problemas de heterogeneidad ocurren por la necesidad de integrar diferentes tipos de hardware y software en un mismo sistema distribuido. A nivel de hardware, el sistema puede integrar diferentes tipos de computadoras interconectadas con equipos de red de diferentes tipos. A nivel de software, el sistema puede tener diferentes tipos de sistemas operativos, alojando componentes desarrollados en diferentes tipos de lenguajes de programación. En la actualidad, la definición y la implementación de ciertos estándares abiertos a nivel de hardware y de software en las diferentes plataformas, ha permitido establecer la comunicación de redes heterogéneas, siendo Internet el caso extremo de diversidad. Sin embargo, en el caso de sistemas distribuidos es necesario una capa adicional denominada *middleware*. Esta capa esconde la heterogeneidad y presenta un modelo computacional uniforme o una abstracción programática para usar los servicios y las aplicaciones distribuidas. El concepto de *middleware* se explica más adelante.

La apertura se refiere a la capacidad del sistema distribuido para ser extendido y para ser implementado fácilmente en diferentes ambientes o de diferentes formas. La extensión del sistema puede darse en forma de replicación de un servicio existente para satisfacer el crecimiento en la demanda, la incorporación de un nuevo servicio o variaciones de un servicio existente, la inclusión de nuevos tipos de recursos.

Algunos mecanismos que promueven la apertura en un sistema distribuido son: el uso de estándares comunes como TCP para transporte o XML[23] para definir el formato de los datos, la definición y la publicación de una interfaz estándar para facilitar el uso de los

servicios o recursos del sistema distribuido, y la definición de una interfaz estándar para el uso de un componente del sistema distribuido. El uso de interfaces desliga los componentes del sistema y facilita la sustitución de algún componente con una nueva implementación.

La seguridad trata con amenazas asociadas a la naturaleza distribuida del sistema y plantea posibles soluciones a estos problemas.

La escalabilidad se refiere a la capacidad del sistema distribuido para mantener su efectividad ante un incremento significativo en el número de usuarios atendidos o en el número de recursos compartidos administrados por el sistema.

En el manejo de fallos se debe tener en cuenta que los fallos en un sistema distribuido son parciales, independientes y pueden ocurrir en cualquier nodo o sección de la red. Un buen manejo de fallos permite mantener un alto grado de disponibilidad del sistema.

Los problemas de concurrencia ocurren cuando el sistema debe atender solicitudes de diferentes clientes al mismo tiempo. El sistema debe evitar que la atención de una solicitud interfiera con los resultados de otras solicitudes y mantener un nivel adecuado de rendimiento. El sistema debe implementar mecanismos de sincronización y usar técnicas de programación concurrente para reducir el tiempo ocioso.

Los retos de transparencia están relacionados con el deseo de ocultar la naturaleza distribuida del sistema, con el objetivo de facilitar su uso por parte de usuarios o aplicaciones clientes. En este trabajo se considerarán las siguientes formas de transparencia:

- De acceso. El sistema debe presentar las mismas operaciones para accesar recursos locales y remotos.
- De localización o ubicación. El cliente no requiere conocer la ubicación física del recurso para usarlo. Se deben usar nombres simbólicos para identificar recursos.
- De concurrencia. Permitir que varios clientes puedan accesar el recurso al mismo tiempo, sin que se interfieran entre ellos. El sistema debe manejar la concurrencia de manera transparente para los clientes.

Esta sección introdujo el concepto de sistema distribuido y explicó sus aspectos más

relevantes. En las siguientes secciones se revisarán diferentes conceptos y elementos usados en la implementación de las herramientas. El primer tema a tratar es la arquitectura de sistemas distribuidos, explicado en la siguiente sección.

2.2 Arquitectura de sistemas distribuidos

Una aplicación sencilla, que funcione en una única computadora, usa el API provisto por el sistema operativo para manejar memoria, accesar archivos, crear procesos e hilos, solicitar la hora y otros servicios básicos. El sistema operativo constituye una capa de software que encapsula y esconde la complejidad subyacente del hardware. En la actualidad los sistemas operativos más populares para PC son Windows, Linux y Mac OS [24]. Estos incorporan capacidades de red, incluyendo los protocolos TCP o UDP, e IP y permiten el acceso de recursos remotos como impresoras y archivos.

En el caso de sistemas distribuidos es necesaria una capa adicional de software que encapsule la comunicación entre los nodos y el intercambio de mensajes, y que opcionalmente resuelva otros problemas, como heterogeneidad de las plataformas de los diferentes nodos o transparencia de localización de los nodos. En el contexto de sistemas distribuidos, esta capa se denomina *middleware*. Ella se encarga de encapsular y esconder la complejidad de la comunicación entre los nodos y provee una interfaz de programación conveniente para la comunicación[20].

Figura 2.1. Arquitectura de sistema distribuido. El sistema distribuido está compuesto por componentes distribuidos alojados en diferentes nodos. Estos componentes intercambian mensajes usando los servicios de comunicación de las capas inferiores. Cuando el mensaje llega a la capa del sistema operativo, este usa algún protocolo de transporte para transmitir el mensaje hacia el sistema operativo de algún otro nodo.

En la figura 2.1 se visualizan los elementos que forman parte de un sistema distribuido. En el nivel superior está la aplicación, compuesta por un conjunto de componentes distribuidos ubicados en diferentes computadoras. En la capa de componentes distribuidos, se implementan servicios que pueden ser genéricos, como un servicio de directorio o un servicio particular de la aplicación. Un ejemplo de un servicio genérico puede ser el servicio DNS[25] (Domain Name System), para resolver nombres legibles a direcciones IP. Un ejemplo de una aplicación particular es LimeWire[9]: un programa para compartir archivos en Internet, en donde un componente particular podría tener la función de solicitar partes de un archivo que desea bajar a otros nodos, y ensamblarlo en la computadora local. Este componente se comunica con componentes similares de otras instancias de LimeWire corriendo en otros nodos. Los componentes distribuidos intercambian mensajes usando la interfaz de programación provista por el *middleware*, el cual es una abstracción construida sobre algún medio básico de transporte como TCP/IP. Algunos ejemplos de *middleware* son RMI (Remote Method Invocation) de Java[14], CORBA (Common Object Request Broker Architecture)[13], RPC (Remote Procedure Call)[12]. En general, el *middleware* esconde la capa de transporte y provee una interfaz de programación más conveniente para el programador de componentes distribuidos. Debajo del *middleware* se ubica la plataforma compuesta por el sistema operativo y el hardware. Diferentes nodos del sistema distribuido pueden tener diferentes plataformas. En la actualidad, es común que el sistema operativo provea los servicios básicos de transporte (TCP o UDP sobre IP), aunque se pueden usar otros mecanismos de transporte externos al sistema operativo como JMS[26] (Java Message Service).

Como se explicó anteriormente, una capa muy importante en la arquitectura de sistemas distribuidos es la capa del *middleware*. En la siguiente sección se explica este concepto.

2.2.1 Middleware

La palabra *middleware* aparece por primera vez en la edición de 1970 de Dictionary of Computers[27] para referirse al software de proveedores de computadoras adaptados para las necesidades particulares de una instalación. En 1972 se redefine este término como el conjunto de programas, ubicados entre el sistema operativo y las aplicaciones finales, que

implementan las modificaciones o mejoras al sistema operativo estándar para satisfacer las necesidades cada vez más complejas de las aplicaciones finales[28]. En el contexto actual de sistemas distribuidos, el *middleware* es una capa de software que encapsula y esconde la complejidad de la comunicación entre nodos y provee una interfaz de programación conveniente para la comunicación[20]. Esta capa asume tres grupos de responsabilidades: serialización de datos, protocolos de solicitud y respuesta, modelos de programación.

La serialización de datos se encarga de la conversión de representaciones de datos en memoria a un formato transportable. Por ejemplo, convertir un objeto de Java a una secuencia de bytes, para enviarlo a otro nodo y su operación inversa, convertir la secuencia de bytes recibida a un objeto. En la actualidad, es común el uso de XML como formato transportable de los datos.

Los protocolos de solicitud-respuesta definen el formato del mensaje, un mecanismo de transporte y una secuencia de intercambio de mensajes que le permita a los componentes distribuidos solicitar servicios y recibir respuestas.

El modelo de programación implementa la interfaz de programación para la aplicación distribuida. Existen tres modelos principales de programación: 1) basado en mensajes, 2) basado en llamados a procedimientos (Remote Procedure Call) y, 3) basado en objetos.

En todos los modelos de programación, siempre se intercambian mensajes en el nivel más bajo. Desde el punto de vista del programador de componentes distribuidos, el primer modelo es el más complejo pues debe encargarse del envío y de la recepción de los mensajes. Los otros dos modelos son bastante más sencillos pues invocan procedimientos o métodos que representan operaciones de más alto nivel.

Con la explicación de *middleware* concluye este capítulo, en donde se explicó el concepto y la arquitectura de sistemas distribuidos. En el siguiente capítulo se describen los servicios de las herramientas.

3 Descripción de los servicios

En este capítulo se identifican las funciones que debe proveer *D-Explorer* y se definen abstracciones, operaciones y componentes para suplir estas funciones. Estas actividades están relacionadas con los primeros dos objetivos específicos: identificar servicios y funcionalidades que faciliten la exploración de sistemas distribuidos, y definir abstracciones y operaciones que provean estas funcionalidades, teniendo en cuenta que se usarán en ambientes académicos. Seguidamente se identifican las funciones de *D-Explorer*.

3.1 Identificar funciones

Se identifican las siguientes funciones a incluir en *D-Explorer*: comunicación, localización, sincronización y almacenamiento centralizado de información.

La comunicación es una funcionalidad fundamental para desarrollar aplicaciones distribuidas. Los componentes distribuidos del sistema necesitan transmitir mensajes entre ellos para intercambiar información y coordinar acciones entre ellos. La comunicación entre los procesos puede ser secuencial o concurrente y sincrónica o asincrónica.

La necesidad de localización surge cuando un proceso del sistema distribuido arranca su ejecución. Este proceso no tiene conocimientos del resto del sistema y necesita localizar los nodos activos y los servicios existentes para interactuar con estos.

La sincronización es una necesidad que se deriva de la naturaleza concurrente del sistema distribuido. En este contexto, se necesitan mecanismos para accesar recursos del sistema sin interferencia de otros procesos, o coordinar actividades entre varios procesos.

Se identifica también la necesidad de un punto central de almacenamiento de archivos para simplificar la logística para configurar el sistema o recopilar resultados del sistema.

Las funciones identificadas contienen las funciones mínimas requeridas para desarrollar aplicaciones distribuidas: permite localizar nodos, intercambiar mensajes entre nodos e implementa varios mecanismos para sincronizar procesos que están corriendo en diferentes nodos. Existen otras funciones que se podrían considerar, como replicación, elecciones,

cachés, etc, pero se descartan por las siguientes razones: las funciones identificadas contienen lo necesario para desarrollar aplicaciones distribuidas, incluyendo las funciones de comunicación, las herramientas son extensibles y permiten incorporar nuevas funciones en el futuro, y muchas de estas funciones son bastantes sencillas y se pueden implementar como parte una tarea programada de algún curso de sistemas distribuidos.

Después de identificar las funciones de las herramientas, a continuación se definen los componentes y los servicios que suplen las funciones de comunicación, localización, sincronización y almacenamiento centralizado de información de *D-Explorer*. En la siguiente sección se describen los componentes y servicios de comunicación.

3.2 Componentes y servicios de comunicación

El *socket* es una herramienta de comunicación que ha sido usado con éxito para desarrollar aplicaciones distribuidas. Esta herramienta se puede usar para la comunicación secuencial, sincrónica o asincrónica, y se puede complementar con hilos para incorporar concurrencia. El *socket* provee todas las funciones de comunicación que una aplicación distribuida podría requerir y *D-Explorer* podría usarlos para las funciones de comunicación pero sería más conveniente ofrecer abstracciones de más alto nivel y APIs más sencillos para la comunicación. En *D-Explorer* se definen, entonces, los siguientes componentes y servicios de comunicación: el canal de servicios, el conector de servicios, el componente de servicios y el servicio de notificación. El canal de servicios administra los *sockets* y provee servicios de comunicación a los otros tres componentes. El conector de servicios se usa en la comunicación sincrónica. El componente de servicios encapsula la comunicación concurrente. El servicio de notificaciones permite el envío asincrónico de mensajes. Seguidamente se explican estos cuatro componentes, empezando con la descripción del canal de servicios en la próxima sección.

3.2.1 Canal de servicios

En sistemas distribuidos, un canal es un componente de software que provee el servicio de intercambio de mensajes entre dos procesos. El canal se construye sobre protocolos de transporte existentes como TCP o UDP, usa los enlaces físicos de la red y puede

proporcionar funcionalidades como: comprimir y encriptar los mensajes, administrar hilos para atender solicitudes concurrentemente, o usar operaciones asincrónicas para mejorar el rendimiento del canal. Si el canal usara protocolos no confiables como UDP, este debería implementar un mecanismo de reenvío y de ordenamiento de los mensajes, para garantizar el arribo de los mensajes en el orden correcto.

En *D-Explorer*, el canal de servicios en un componente de software construido sobre TCP/IP que ofrece servicios de comunicación a otros componentes de *D-Explorer*. El canal de servicios maneja los *sockets* y esconde mucha de la complejidad involucrada en su manejo: establece o acepta conexiones, envía y recibe bytes, opera los *sockets* en modo asincrónico para mejorar el rendimiento, reutiliza conexiones abiertas, maneja hilos para atender solicitudes concurrentemente, serializa y deserializa los mensajes, etc. Los procesos de la aplicación distribuida no usan las funciones de comunicación del canal de servicios directamente sino que estos usan conectores de servicio para intercambiar mensajes, envían notificaciones a través del servicio de notificación o implementan servicios usando el componente de servicio.

En esta sección se explicó el canal de servicios, un componente que ofrece servicios de comunicación a otros componentes de *D-Explorer*. En la siguiente sección se explica el conector de servicios, una abstracción para intercambiar mensajes entre procesos.

3.2.2 Conector de servicio

El conector de servicio es un componente que implementa un mecanismo sincrónico de intercambio de mensajes entre dos procesos. Conceptualmente, el conector de servicio es similar a un *socket*: representa un extremo de la conexión entre dos procesos y tiene operaciones para enviar y recibir mensajes del otro extremo de la conexión.

El canal de servicios administra un conjunto de conectores de servicios. Para identificar un conector de servicio, se asigna un entero positivo único a cada conector. Este número es denominado el número de servicio del conector de servicio o simplemente el número de servicio. Mientras que en los *sockets* se usa el par *<dirección ip,puerto>* para identificar un destino, en los conectores de servicio se usa la tripleta *<dirección ip, # puerto, # servicio>*

para identificar un destino. Los atributos *dirección ip* y *# puerto* identifican el canal destino, y el atributo *# servicio* identifica el conector de servicio en el canal destino.

Para establecer una conexión, un proceso debe pasarle una dirección al canal de servicios. El canal establece la conexión a esa dirección, asocia un conector de servicio al extremo local de la conexión y retorna este conector al proceso. Luego, el proceso puede usar este conector para enviar y recibir mensajes al otro extremo de la conexión.

Una conexión entre dos conectores de servicios tiene tres segmentos: el segmento entre el conector de servicio local y el *socket* local administrado por el canal de servicios local, el segmento entre el *socket* local y el *socket* remoto administrado por el canal de servicios remoto, y el segmento entre el *socket* remoto y el conector de servicio remoto. Un conector de servicio está asociado con un único *socket* pero un *socket* puede estar asociado con varios conectores de servicio. Cuando se envía un mensaje por el conector de servicios, este pasa del conector al *socket* local, y de este al *socket* remoto. Una vez ahí, se usa el *número de servicio destino* del mensaje para determinar el conector de servicio al cual se le debe entregar el mensaje.

El conector de servicio es un mecanismo barato y eficiente de comunicación pues usa las conexiones de *sockets* mínimas necesarias a nivel de programas. La primera conexión entre dos programas incurre en un costo alto pues establece una conexión a nivel de *sockets* pero las siguientes conexiones entre conectores de servicios de estos programas reutilizan la misma conexión de *socket*. El costo de abrir y cerrar subsiguientes conexiones entre estos programas es bajo. Para lograr un efecto similar usando *sockets*, el desarrollador debería implementar algún mecanismo para compartir un grupo de *sockets*.

El conector de servicios implementa las operaciones *enviar(Mensaje)* y *Mensaje recibir()* para enviar y recibir mensajes respectivamente. El atributo *Mensaje* representa el mensaje a transmitir y está encapsulado en una clase de Java. Estas operaciones presentan varias ventajas: simplifica el API del conector pues maneja el mensaje como una unidad lógica indivisible y no como una serie de atributos, y permite que el canal implemente estas operaciones como una operación atómica: el canal convierte el mensaje a una secuencia de bytes, y se encarga de sincronizar el envío y la recepción de estos bytes en una operación

atómica, garantizando de esta forma que los bytes de un mismo mensaje sean enviados y recibidos en un bloque contiguo, y no se traslapen con los bytes de otros mensajes.

Después de explicar el conector de servicio, el mecanismo de intercambio sincrónico de mensajes del canal de servicio, en la siguiente sección se explica el componente de servicios, un mecanismo que facilita la interacción cliente-servidor entre procesos.

3.2.3 Componente de servicio

Un servicio es un proceso que recibe solicitudes, las procesa y devuelve un mensaje de respuesta al solicitante. En la implementación de un servicio, se identifican dos tipos principales de lógica: la lógica de comunicación encargada de recibir solicitudes y enviar respuestas, y la lógica de procesamiento. Mientras que la lógica de comunicación es la misma para todos los servicios, la lógica de procesamiento varía de un servicio a otro.

El componente de servicios facilita la implementación de servicios. Este componente se encarga de las funciones de comunicación para recibir solicitudes e implementa un mecanismo para delegar el procesamiento de la solicitud.

El implementador de un servicio escribe la lógica para procesar solicitudes y registra esta lógica en el componente de servicios. Esta lógica se usa cuando el componente de servicios recibe una solicitud dirigida a este servicio.

El componente de servicios usa los conectores de servicios para recibir solicitudes y enviar respuestas. Como cada conector de servicio está identificado unívocamente por su número de servicio, se usa este número para identificar el servicio. Cuando se registra un servicio, se debe indicar el número de servicio asociado al servicio. Así por ejemplo, un implementador de servicios puede escribir dos lógicas diferentes para procesar mensajes de solicitud, las cuáles se denotan como *lógica A* y *lógica B*, y registrarlas en el componente de servicios. La *lógica A* se registra con el número de servicio *501* y la *lógica B* con el número de servicio *502*. Internamente, el componente de servicios asocia el conector de servicio *501* con la *lógica A* y el conector de servicio *502* con la *lógica B*. Cuando llega un mensaje de solicitud al conector *501*, el componente de servicio usa la *lógica A* para procesar la solicitud. Esta lógica procesa el mensaje y devuelve un mensaje de respuesta.

Luego, el componente de servicio envía esta respuesta a través del conector de servicio 501. Para usar el servicio *501*, un proceso cliente debe crear un conector de servicio, establecer una conexión al número de servicio *501*, enviar una solicitud y esperar la respuesta.

El componente de servicios maneja un conjunto de hilos y procesa cada solicitud en un hilo separado para alcanzar un nivel adecuado de concurrencia y de eficiencia.

Para facilitar el uso de un servicio, se debe implementar un objeto remoto que encapsule cada operación del servicio en un método. El cliente crea una instancia de este objeto e invoca sus métodos para obtener los servicios. La implementación de cada método involucra la construcción de un mensaje específico usando los parámetros del método, el envío del mensaje al servicio, la espera por el mensaje de respuesta del servicio, y la conversión de la respuesta a un resultado del método. La responsabilidad de implementar el objeto remoto recae en el programador que desarrolla el servicio.

En este esquema, el número de servicio es administrado por la aplicación distribuida. Diferentes aplicaciones distribuidas pueden asignar los números de servicio de diferente manera. Este enfoque promueve el crecimiento modular de la aplicación, ligando nuevos servicios a números de servicios no asignados. También permite la estandarización de servicios dentro de una aplicación distribuida. Por ejemplo, una aplicación puede usar el número de servicio *2* para el servicio de directorio, el *3* para el servicio de notificaciones y, en general, reservar un conjunto de números de servicios para los servicios más relevantes. Las ventajas de modularización también se extienden a los componentes. Un componente puede construirse usando un grupo de servicios, cada uno asignado a un número de servicio diferente, pero cooperando estrechamente entre ellos para realizar las funciones del componente o simplemente cumpliendo funciones diferentes dentro del componente. Por ejemplo, un componente puede usar el número de servicio *3* para atender solicitudes normales de clientes, el *4* para intercambio de mensajes de control con otros componentes similares alojados en otros nodos, el *5* para funciones de monitoreo del servicio, etc.

Con los mecanismos propuestos, se simplifica el desarrollo de servicios y el uso de servicios sin sacrificar el rendimiento. El componente de servicios absorbe los detalles de intercambio de mensajes y del manejo de los hilos de atención de solicitudes. El

desarrollador de servicios se enfoca en la lógica del servicio y el cliente usa los servicios invocando métodos remotos. En el caso de *D-Explorer*, los servicios internos de la herramienta tendrán objetos remotos para facilitar el uso del servicio.

En esta sección se explicó el componente de servicios. A continuación se explica el servicio de notificaciones, un servicio construido sobre el componente de servicios que facilita el envío asincrónico de mensajes.

3.2.4 Servicio de notificación

Conceptualmente, un servicio de notificaciones es un mecanismo de intercambio asíncrono de mensajes entre procesos. Los procesos interesados en recibir mensajes se registran con el servicio de notificaciones. Los procesos interesados en enviar mensajes los deposita en el servicio de notificaciones, y este los envía a los procesos interesados.

Con el servicio de notificaciones se puede implementar los eventos remotos y las interacciones asíncronas *productor-consumidor* o *publicador-suscriptor*. Un evento es un mensaje cuya función es informar al receptor sobre la ocurrencia de cierta condición en el sistema distribuido. A modo de ejemplo, este servicio se podría usar para implementar el algoritmo de región crítica de Ricart & Agrawala[29]: las solicitudes de permiso para entrar a la región se envían usando conectores de servicio pero las autorizaciones de permiso para entrar a la región se reciben asincrónicamente a través del servicio de notificaciones. El acoplamiento es bajo entre el proceso que solicita el permiso y los procesos que autorizan. Las autorizaciones se envían al servicio de notificaciones, y este las hace llegar al interesado, en este caso, el proceso que solicita el permiso para entrar a la región crítica.

El servicio de notificación de *D-Explorer* se construye sobre el componente de servicios e implementa dos operaciones:

> *agregarManejador(Nombre, Proceso)*. Este método registra un proceso en el servicio de notificaciones bajo el nombre *Nombre*.
>
> *notificar(Direccion, Nombre, Mensaje)*. El parámetro *Direccion* contiene la dirección de un nodo, el parámetro *Nombre* corresponde al nombre de un proceso. Esta operación envía el *Mensaje* al proceso *Nombre* del nodo *Direccion*.

El servicio de notificaciones de *D-Explorer* se debe activar en todos los nodos del sistema distribuido. Los procesos interesados en enviar mensajes invocan el método *notificar* pasándole la dirección del nodo y el nombre del proceso destino. Los procesos interesados en recibir mensajes usan la operación *AgregarManejador* para registrarse con el servicio.

Con el servicio de notificaciones concluye la explicación de componentes de comunicación de *D-Explorer*. En la siguiente sección se explican las ventajas de estos componentes.

3.2.5 Ventajas de los componentes de comunicación

Las funciones de comunicación de *D-Explorer* son implementadas por el canal de servicios, el conector de servicio, el componente de servicio y el servicio de notificación. Esta separación permite que cada componente se especialice absorbiendo más funciones y a la vez simplifica su interfaz de programación pues la interfaz no necesita exponer todas estas funciones. Por ejemplo, el canal de servicios se encarga de convertir mensajes a bytes y de transmitir estos bytes usando un *socket*. Al asumir estas funciones, el canal puede ofrecer una interfaz de programación más sencilla basada en mensajes y no en bytes o en datos como enteros, hileras, etc. Otro ejemplo es el componente de servicio, el cual se encarga de la lógica de comunicación y delega la implementación de la lógica de procesamiento en el programador de la aplicación distribuida. Esta separación permite que el componente de servicios, sin complicar su interfaz de programación, pueda implementar una lógica de comunicación compleja que maneja la concurrencia.

D-Explorer presenta un modelo de programación basado en eventos en donde se escriben métodos para procesar eventos. En este caso, el evento es la llegada de un mensaje y el programador escribe un método que recibe un mensaje como parámetro y procesa el mensaje. Este método se registra en el componente de servicios o en el servicio de notificaciones, y cuando el canal de servicios recibe un mensaje, se invoca este método para procesar el mensaje. Este modelo de programación es bastante común. Por ejemplo, en el desarrollo de interfaces gráficas se escriben métodos para manejar eventos como el click del ratón, la modificación de un campo de texto, etc. Además, tiene mucho sentido usar este modelo en el desarrollo de aplicaciones distribuidas pues los mensajes son enviados hacia

otros nodos para que sean procesados. El envío de un mensaje es un medio y no un fin. El destino final de los mensajes son procesos que reciben mensajes, los procesa y opcionalmente, devuelven un mensaje de resultado.

El componente de servicios y el servicio de notificación presenta una misma interfaz de programación para desarrollar aplicaciones con comunicación sincrónica o asincrónica y secuencial o concurrente. El programador implementa métodos para procesar mensajes, y la asincronía y la concurrencia es manejada internamente por estos componentes.

Los componentes de comunicación de *D-Explorer* ofrecen un paradigma modular de desarrollo de servicios similar al paradigma que ofrece el *socket* a nivel de computadora. En este paradigma de desarrollo, los servicios de la aplicación distribuida se pueden desarrollar y mantener de manera independiente, e integrar en el canal de servicios sin producir conflictos de interferencia entre ellos. Este paradigma facilita la incorporación de nuevos servicios en la aplicación y permite que cada aplicación distribuida administre su rango de números de servicios a su conveniencia.

Cada servicio tiene asignado un número de servicio único y es registrado en el canal de servicios, y más allá de este punto común, cada servicio opera aislado de los otros servicios. El conector de servicio representa un diálogo entre un servicio y un cliente. Un servicio puede mantener, al mismo tiempo, más de un diálogo con varios clientes usando diferentes conectores de servicio. Estos diálogos no se interfieren entre ellos.

Después de presentar las principales ventajas de los componentes de comunicación de *D-Explorer*, en la siguiente sección se describen los componentes de localización.

3.3 Componentes de localización

Un proceso del sistema distribuido que inicia su ejecución no tiene conocimientos del resto del sistema. Este proceso necesita localizar los nodos y servicios activos para interactuar con el resto del sistema. Esta necesidad se puede resolver con un servicio de directorio y un servicio de descubrimiento. El servicio de directorio mantiene una base de datos con información sobre los nodos y los servicios del sistema distribuido, y resuelve consultas sobre la localización de estos. El servicio de descubrimiento implementa un mecanismo

para descubrir el directorio. Estos dos servicios se usan de la siguiente manera: el proceso inicial del sistema distribuido debe activar un servicio de descubrimiento y un servicio de directorio. Otros procesos que se activan posteriormente usan el servicio de descubrimiento para localizar el nodo inicial. Dado que en el nodo inicial también está el servicio de directorio, estos otros procesos pueden registrarse en el directorio, registrar sus servicios y consultar sobre la ubicación de otros servicios que requiera. Seguidamente se explican estos dos servicios, empezando con el servicio de directorio.

3.3.1 Servicio de directorio

Un servicio de directorio mantiene una base de datos sobre objetos del sistema distribuido y resuelve consultas sobre dichos objetos. Cada objeto está identificado por un nombre único y tienen asociado una lista de atributos. Conceptualmente, un objeto se puede representar como una tupla de la siguiente forma: $<nombre, atributo_1=valor_1, ...>$. Algunos ejemplos de atributo son: el tipo del objeto, la dirección de red en donde está el objeto, el estado del objeto, etc. Un servicio de directorio provee operaciones de mantenimiento para agregar objetos, eliminar objetos, agregar atributos, eliminar atributos o modificar el valor de un atributo. También provee facilidades de consulta como: buscar el objeto con el nombre *X*, buscar los objetos que contenga el valor *X* en el atributo *Y*, o cuál es el valor del atributo *X* del objeto *Y*. Si el servicio solo implementa la consulta por nombre, el servicio es denominado servicio de nombres o de resolución de nombres, y en este caso, la única consulta posible es buscar el objeto con el nombre *X*. Por su similitud con el directorio telefónico tradicional, al servicio de directorio también se le denomina páginas amarillas, mientras que al servicio de nombre se le denomina páginas blancas[20]. Un ejemplo de servicio de directorio es **X.500** [30]. Este servicio permite búsquedas por páginas blancas y por páginas amarillas y la base de datos con los objetos del directorio pueden ser particionados y alojados en diferentes nodos.

En *D-Explorer* se implementa el servicio de directorio con las siguientes operaciones:

inscribirServicio(Servicio): agrega un servicio en el directorio. El parámetro *Servicio* representa un servicio y tiene atributos como *nombre*, *tipo*, etc.

desinscribirServicio(Servicio): remueve un servicio del directorio.

consultarServicioPorNombre(String nombre): busca un servicio por nombre en el directorio.

consultarServicioPorTipo(String tipo): busca servicios por tipo en el directorio. Puede retornar una lista de servicios del tipo buscado.

entrarGrupo(Nodo, Grupo): agrega un nodo a un grupo.

salirGrupo(Nodo, Grupo): elimina un nodo de un grupo.

consultarGrupo(String grupo): devuelve una lista con los nodos del grupo.

En esta sección se explicó el servicio de directorio. En la siguiente sección se explica el servicio de descubrimiento.

3.3.2 Servicio de descubrimiento

En sistemas distribuidos, un problema fundamental es el descubrimiento del servicio de directorio. Los dos enfoques más comunes para este problema son: usar direcciones conocidas o usar *broadcast* en LAN. El primer caso implica que cada nodo tiene conocimiento de esta dirección, la cual está codificada como una constante dentro del programa. Si el servicio de directorio es trasladado a otra computadora, es necesario modificar esta constante, producir un nuevo ejecutable y distribuirlo en todos los nodos. Se puede usar direcciones simbólicas como "www.directorio.com" para mitigar los efectos de este problema. En este caso no se modifican los programas sino que se configuran los servidores DNS para asignarle una nueva dirección física a la dirección simbólica. En el segundo caso, un nodo difunde un mensaje específico por la red. Cuando el nodo con el servicio de directorios recibe este mensaje, le responde indicándole su dirección. En este caso, el mecanismo no necesita conocer la dirección del servicio pero la difusión de mensajes puede impactar negativamente en el rendimiento de la red.

En *D-Explorer* se usa *broadcast* en LAN para implementar el servicio de descubrimiento. Este servicio solo tiene la operación *descubrir*(), la cual se usa para localizar el nodo inicial y obtener la dirección de este nodo.

Con el servicio de descubrimiento concluyen las funciones de localización. En la siguiente sección se explican las funciones de sincronización.

3.4 Componente de sincronización

Las funciones de sincronización son usadas por los procesos concurrentes del sistema distribuido para coordinar actividades entre ellos o controlar el uso de los recursos compartidos. En *D-Explorer* se implementan el semáforo y la barrera, dos componentes de sincronización que se describen a continuación.

3.4.1 Semáforo

El semáforo es un mecanismo de sincronización que permite implementar regiones críticas. Este mecanismo permite que un proceso realice un conjunto de operaciones sobre algún recurso sin interferencia de otros procesos, y ha sido usado para resolver problemas de sincronización como *Producer-Consumer*, *Dining Philosopher* y *Reader-Writer*. En procesos que se ejecutan en una misma máquina, el sistema operativo ofrece primitivas que garantizan la exclusión mutua. En sistemas distribuidos, el sistema operativo no provee este servicio y es necesario implementar la exclusión mutua mediante intercambio de mensajes.

El problema de región crítica distribuida se plantea de la siguiente manera. En un sistema distribuido asincrónico con N procesos que no comparten memoria, con canales confiables y procesos que no fallan, se debe implementar un mecanismo basado en intercambio de mensajes que satisfaga las siguientes condiciones[20]:

- RC1 (seguridad): en un momento dado puede haber a lo sumo un proceso en la región crítica.
- RC2 (vitalidad[2]): eventualmente la solicitud de entrada o de salida de la región crítica debe ser aprobada.

De manera opcional, el mecanismo puede satisfacer la siguiente condición de equidad:

- RC3 (orden causal): si una solicitud de entrada ocurre antes que otra, la aprobación de la primera debe ocurrir antes que la aprobación de la segunda.

2 Calidad de tener vida o estar vivo.

Para evaluar el rendimiento de la solución al problema, se consideran los siguientes factores:

- Cantidad de mensajes enviados para implementar la región crítica.
- El retraso incurrido por el cliente para entrar o salir de la región crítica.
- El retraso de sincronización medido en número de mensajes incurridos por la salida de un proceso a la región y la entrada a la región de un proceso que está esperando usar el recurso.

Existen varios algoritmos distribuidos que implementan regiones criticas: el algoritmo del servidor central [20], el algoritmo del anillo lógico [20], el algoritmo de Ricart & Agrawala [29] y el algoritmo de Maekawa [31]. En *D-Explorer* se implementa el algoritmo del servidor central pues a diferencia de los otros algoritmos, en este el cliente solo interactúa con el servidor y no con los otros clientes que accesan el recurso compartido, lo que facilita su uso por parte del programador.

En el algoritmo del servidor central se tiene un servidor que controla el acceso a la región crítica y concede permisos para entrar a la región crítica. Los clientes deben solicitar permiso al servidor para entrar a la región crítica. Seguidamente se describe el algoritmo:

1. Cuando un cliente desea entrar a la región crítica:
   - Envía un mensaje de solicitud de entrada al servidor.
   - Espera un mensaje de entrada concedida del servidor, que le indica que puede entrar a la región crítica.
1. Cuando el servidor recibe un mensaje de solicitud de entrada:
   - Si la región crítica está desocupada, le envía un mensaje de solicitud concedida al cliente y pasa al estado de región crítica ocupada.
   - Si la región crítica está ocupada, agrega la solicitud a la cola de solicitudes pendientes.
1. Cuando el cliente que está en la región crítica desea salir:

- Envía un mensaje de salida al servidor.

1. Cuando el servidor recibe un mensaje de salida:
   - Si hay solicitudes pendientes en la cola, envía un mensaje de solicitud concedida al primer cliente en la cola y lo elimina de la cola.
   - Si no hay solicitudes pendientes, pasa al estado de región crítica desocupada.

El algoritmo cumple las condiciones RC1 y RC2 pero no cumple RC3. Los retrasos arbitrarios en la red pueden ocasionar que una solicitud enviada antes no llegue de primero al servidor. Para entrar a la región crítica se ocupan dos mensajes, para salir se ocupa un mensaje. El retraso de sincronización es de dos mensajes. Para completar las operaciones de sincronización, el algoritmo intercambia pocos mensajes. Sin embargo, presenta problemas de escalabilidad. El algoritmo puede fallar si el servidor falla o si el proceso que está en la región crítica falla y no le envía el mensaje de salida al servidor.

En *D-Explorer*, el servicio de sincronización implementa el semáforo usando el algoritmo del servidor central y provee las siguientes operaciones para manejar semáforos:

int abrirSemaforo(String nombre, int n): crea o abre un semáforo *n*-ario. Retorna un identificador único asociado a este semáforo. Este identificador es usado en las operaciones *cerrarSemaforo*, *adquirirSemaforo* y *liberarSemaforo*.

int abrirSemaforo(String nombre): crea o abre un semáforo binario.

cerrarSemaforo(int semaforo): cierra el semáforo.

adquirirSemaforo(int semaforo): adquiere el semáforo. Esta operación se bloquea hasta adquirir el semáforo.

liberarSemaforo(int semaforo): libera el semáforo.

Después de explicar el semáforo, un mecanismo de sincronización que implementa las regiones críticas, seguidamente se describe el concepto y la implementación de la barrera.

3.4.2 Barrera

La barrera[32] es un mecanismo de sincronización, conocido también como *Rendezvous* o

punto de reunión. Este mecanismo obliga a los procesos a esperar en la barrera hasta que todos los procesos hayan alcanzado este punto. Se usa para detectar la finalización de un grupo de tareas concurrentes realizadas por diferentes procesos. Dado un proceso cuyo inicio depende de la finalización de un conjunto de tareas previas, se puede usar una barrera para sincronizar la finalización de las tareas con el inicio del proceso.

En *D-Explorer* se implementa la barrera usando un servidor central. La barrera mantiene el número total de procesos y el número de procesos que han terminado su ejecución. También mantiene una cola con los procesos bloqueados. Seguidamente se describe el algoritmo de servidor central para implementar barreras:

1. Inicialización:
   - TOTAL se inicializa con el número total de procesos.
   - FINALIZADOS se inicializa con cero.
1. Cuando el proceso ***i*** termina
   - Envía un mensaje al servidor informando que ha terminado.
   - Espera hasta que reciba el mensaje de respuesta del servidor.
1. Cuando el servidor recibe un mensaje de finalización del proceso ***i***:
   - Incrementa FINALIZADOS
   - Si FINALIZADOS es menor que TOTAL

     Agrega el proceso ***i*** a la cola de procesos bloqueados

     sino

     Envía un mensaje de respuesta al proceso ***i***.

     Envía un mensaje de respuesta a cada proceso de la cola de bloqueados

El servicio de sincronización de *D-Explorer* contiene la operación *barrera(String nombre, int n)*, la cual se usa para crear una barrera que sincroniza un grupo de *n* procesos. El parámetro *nombre* permite crear barreras con diferentes nombres, y usar estas para coordinar diferentes grupos de procesos sin que un grupo interfiera con otros grupos.

Después de revisar varios mecanismos de sincronización de procesos, en la siguiente

sección se describen las funciones de almacenamiento de datos.

3.5 Componente para almacenamiento de datos

Las funciones de almacenamiento de datos se identifican por la necesidad de proveer un mecanismo sencillo para almacenar archivos de parámetros o de resultados. Los archivos de parámetros almacenan la configuración de un sistema distribuido y modificando este archivo se modifica el comportamiento del sistema. Los archivos de resultado guardan los resultados de los diferentes procesos en un punto centralizado. Esto facilita la logística para configurar y para revisar resultados de una aplicación distribuida. En *D-Explorer*, estas funciones son implementadas por el servicio de archivos, el cual se explica a continuación.

3.5.1 Servicio de archivos

Un sistema local de archivos[20] es un módulo del sistema operativo que administra los medios físicos de almacenamiento y provee una interfaz de programación de alto nivel para accesar estos recursos. El modelo predominante de interacción y de organización del sistema de archivos es el paradigma de directorios y archivos. Los directorios y los archivos están identificados por nombres. Hay un directorio raíz y los directorios restantes están organizados jerárquicamente a partir del directorio raíz. Un directorio es una lista de otros directorios y de archivos. Un archivo es una secuencia de bytes cuyo significado es irrelevante para el sistema de archivos. El sistema de archivos tiene operaciones sobre directorios y archivos como la creación de directorios o archivos, la navegación por la estructura jerárquica de directorios, la lectura de una parte del archivo, etc.

Un sistema distribuido de archivos[20] es una capa de software que se construye sobre los sistemas locales de archivos de los diferentes nodos. Esta capa tiene la función principal de compartir archivos del sistema local con clientes remotos. Un ejemplo de sistema distribuido de archivos es Sprite Network File System [33].

El servicio de archivos de *D-Explorer* implementa un sistema distribuido de archivos. Este servicio comparte archivos del sistema local de archivos a través de varios objetos remotos que encapsulan flujos de entrada y de salida de datos y de texto. Los flujos de entrada se

usan para leer archivos remotos. Los flujos de salida se usan para grabar archivos remotos. Los flujos de datos permiten accesar datos simples de diferentes tipos (*int*, *double*, *String*, etc). Los flujos de texto permiten accesar archivos de texto únicamente. Un proceso cliente usa estos flujos para realizar operaciones sobre el sistema local de archivos del servidor. Las operaciones solicitadas son transmitidas al servicio de archivos. Este servicio realiza las operaciones solicitadas en el sistema local de archivos. Los resultados son devueltos, por la misma vía, al proceso cliente.

El flujo de entrada de datos tiene operaciones para leer datos primitivos del archivo: *int getInt()*, *String getString()*, *bool getBool()*, etc.

El flujo de salida de datos tiene operaciones para grabar datos primitivos al archivo: *putInt(int)*, *putString(String)*, *putBool(bool)*, etc.

El flujo de entrada de texto tiene la operación *String readLine()* para leer una línea de texto del archivo.

El flujo de salida de datos tiene la operación *writeLine(String)* para grabar una línea de texto al archivo.

El servicio de archivos debe garantizar un nivel mínimo de concurrencia. Para ello, en *D-Explorer* se usa el esquema *Multiple Reader-Single Writer*[32] (MRSW), un esquema para sincronizar el uso de un recurso compartido. En este esquema existe un recurso compartido y dos clases de procesos: *lectores* (*readers*) y *escritores* (*writers*). Un *lector* es un proceso que desea leer el recurso compartido pero no desea modificarlo. Un *escritor* es un proceso que desea modificar el recurso compartido. El esquema MRSW permite que varios *lectores* lean el recurso compartido al mismo tiempo pero solo un *escritor* puede modificar el recurso en un momento dado. Cuando un *lector* intenta leer el recurso, el *lector* puede continuar si no hay procesos usando el recurso o solo hay *lectores* usando el recurso, pero el *lector* queda bloqueado si hay un *escritor* modificando el recurso. Cuando un *escritor* intenta modificar el recurso: el *escritor* puede continuar si no hay procesos usando el recurso pero queda bloqueado si algún proceso está usando el recurso, sea este proceso un *lector* o un *escritor*.

El servicio de archivos implementa el esquema MRSW sobre los flujos de entrada y de salida. Un flujo de entrada es un *lector* y un flujo de salida es un *escritor*. El servicio permite que varios procesos abran el mismo archivo remoto con flujos de entrada ó que un único proceso abra el archivo con un flujo de salida.

Con el servicio de archivos concluye el capítulo sobre funciones, componentes y servicios de *D-Explorer*, en donde cabe destacar el conector de servicio, una simplificación del *socket* a partir del cual se construyen los componentes y servicios más importantes de las herramientas. En el siguiente capítulo se explica la implementación de estos servicios.

4 Implementación de los servicios

En este capítulo se explican los aspectos más relevantes de la implementación de los componentes y servicios de *D-Explorer*, los cuales fueron identificados en el capítulo 3 y se dividen en tres capas: utilitarios, comunicación y servicios internos. En la figura 4.1 se muestran estas capas.

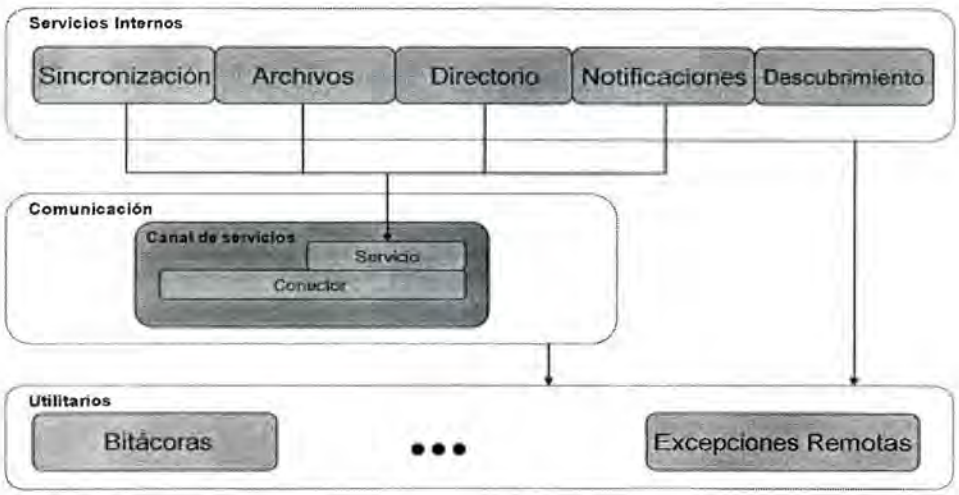

Figura 4.1. Organización de las herramientas. Las componentes y servicios se muestran en rectángulos de tono gris. Las capas se muestran en rectángulos de fondo blanco y contienen grupos de componentes y servicios. Las flechas indican dependencia entre componentes o capas

La capa de utilitarios contiene funcionalidades generales como las excepciones remotas y las bitácoras, y sirven de apoyo a las otras capas.

La capa de comunicación contiene el canal de servicios, un componente que encapsula las funciones de intercambio de mensajes. Los dos componentes principales del canal de servicios son: el conector de servicio, una simplificación del concepto de *socket*, y el componente de servicios, el cual encapsula las funciones de comunicación de un servicio.

La capa de servicios internos se apoya en las dos capas anteriormente descritas y provee servicios de más alto nivel. Esta capa incluye los siguientes servicios: descubrimiento, directorio, notificaciones, sincronización y archivos.

D-Explorer está compuesta por un conjunto de clases y de interfaces programados en Java y está organizada jerárquicamente en *packages* de Java. El *package* raíz es *dexplorer*. En el

package dexplorer está la mayoría de los clases de la capa de utilitarios. El canal de servicios está en el *package dexplorer.canal*. El servicio de descubrimiento está en el *package dexplorer.descubrimiento*. Los demás servicios están dentro del *package dexplorer.servicio*, cada uno en su propio *package*. Por ejemplo, el servicio de sincronización está en el *package dexplorer.servicio.sincronizacion*.

En el resto del capítulo se explican detalles de implementación del canal de servicios y los servicios de directorio, descubrimiento, notificación, sincronización y archivos.

4.1 Canal de servicios

El canal de servicios está compuesto por un conjunto de componentes desacoplados, los cuales, a partir de protocolos de transporte existentes, construyen un servicio para el intercambio de mensajes entre los componentes del sistema distribuido. El canal soporta conectores de servicio para el intercambio de mensajes y manejadores de servicios para facilitar la implementación de servicios. En la figura 4.2 se muestra la arquitectura del canal de servicios y su relación con el cliente y el protocolo de transporte.

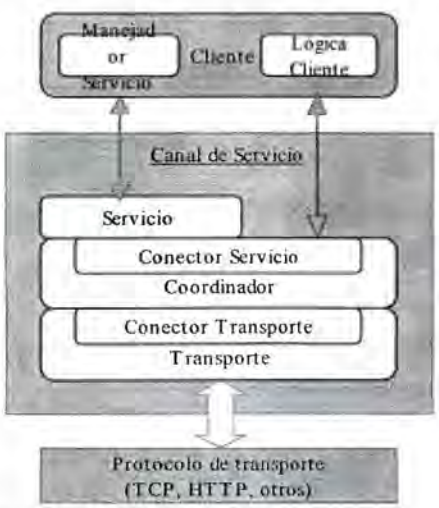

Figura 4.2. Arquitectura del canal de servicios.

El canal contiene los siguientes componentes: transporte, conector de transporte, coordinador, conector de servicio y servicio. En el módulo del cliente están el código del manejador de servicio y la lógica del cliente que usa conectores de servicios. El cliente se

refiere al cliente del canal, el cual puede ser un servicio o un usuario del servicio. El protocolo de transporte provee el medio de transporte, el canal usa este protocolo.

El componente de transporte y el conector de transporte encapsulan un protocolo de transporte y presentan una interfaz de programación independiente del protocolo. Mientras que el componente de transporte implementa operaciones para configurar el protocolo de transporte, establecer conexiones y cerrar conexiones, el conector de transporte implementa operaciones para enviar o recibir mensajes. Pueden existir diferentes implementaciones de estos componentes, cada uno usando un protocolo de transporte particular (TCP, UDP, HTTP) pero implementando la misma interfaz de interacción al coordinador. Un conector de transporte está asociado a una única conexión.

El conector de servicio encapsula un conector lógico para el cliente del canal, su función es proveer la abstracción para enviar y recibir mensajes entre componentes distribuidos. Para enrutar correctamente los mensajes, se mantienen un número de servicio origen y un número de servicio destino en cada mensaje.

Los conectores de servicio usan conectores de transporte para enviar y recibir mensajes. Este enfoque permite usar diferentes implementaciones del componente de transporte y del conector de transporte en el canal de servicios sin necesidad de modificar el código del cliente que usa el canal.

El coordinador actúa como intermediario entre los componentes de transporte y de servicio. Asume las siguientes responsabilidades:

- Enrutamiento de mensajes: el coordinador recibe mensajes del componente de transporte y usa el número de servicio destino del mensaje para enrutarlo al conector de servicio apropiado o al componente de servicio, en el caso de solicitudes cuyo destino es un servicio.
- Registro de servicios: mantiene la lista de los servicios registrados en el canal. Cada servicio tiene asociado un número de servicio, un conector de servicio y un manejador del servicio. El manejador es un método implementado por el programador de servicios para procesar solicitudes.

- Administración de solicitudes e hilos de servicio: el coordinador usa el patrón productor-consumidor para procesar mensajes entrantes destinados a servicios. El coordinador administra un conjunto de hilos y ejecuta cada manejador de servicio en un hilo separado. Cuando el coordinador recibe una solicitud, usa el número de servicio destino del mensaje para determinar el manejador del mensaje, y despacha un hilo para ejecutar este manejador.
- Administración de conexiones de transporte: el coordinador mantiene una lista con las conexiones de transporte establecidas para promover la reutilización de conexiones existentes. Diferentes conectores de servicio pueden compartir la misma conexión de transporte.
- Administración de conectores de servicios: mantiene la lista de conectores de servicios. Hay conectores asignados a servicios y hay conectores creados por la lógica del cliente.

El componente de servicios implementa la clase base de los manejadores de servicios y el hilo encargado de invocar el manejador de servicio.

Aparte de los componentes explicados anteriormente, hay tres elementos adicionales que forman parte del funcionamiento del canal de servicios: la dirección de transporte, el mensaje y el mecanismo de serialización y deserialización de mensajes. La dirección de transporte encapsula direcciones del protocolo de transporte, y es usado por el mecanismo de reutilización de conexiones físicas para identificar cada conexión. El mensaje encapsula la información a enviar o recibir.

En general, el canal de servicios funciona de la siguiente manera:

- Un servidor crea una instancia del canal y registra uno o más servicios.
- Un cliente crea una instancia del canal y le solicita que se conecte a un servicio particular del servidor. El canal establece la conexión y le devuelve un conector de servicio al cliente. El cliente usa el conector de servicio para enviar una solicitud al servicio y espera la respuesta a través del mismo conector. El servicio recibe la solicitud a través del conector de servicio asignado al servicio, procesa la solicitud y

envía la respuesta a la dirección origen de la solicitud.

Las clases del canal están dentro del *package dexplorer.canal*. En este *package* están el coordinador, la dirección de transporte y los mensajes. En *dexplorer.canal.serialización* está el mecanismo de serialización y deserialización, en *dexplorer.canal.servicio* está el componente de servicios, en *dexplorer.canal.transporte* está el componente de transporte y en *dexplorer.canal.transporte.nio* está la implementación del componente de transporte usando el protocolo TCP/IP en modo asincrónico.

En las siguientes secciones se explican los diferentes elementos del canal de servicios. Primero se describen los elementos básicos: la dirección de transporte, el mensaje, el conector de servicios y la interface *ICanal*. Luego se describen los componentes principales: el mecanismo de serialización y deserialización, los componentes de transporte y de servicios y el coordinador.

4.1.1 Dirección de transporte

La clase *DireccionTransporte* es la clase base que representa una dirección de transporte. Para la implementación actual del componente de transporte del canal usando TCP/IP, se deriva la subclase *NioDireccion* con los atributos *host* y *puerto*. El atributo *host* es una hilera y puede contener una dirección IP (192.168.0.1) o un nombre simbólico (www.apache.org).

Otras implementaciones del componente de transporte deben derivar su propia subclase de *DireccionTransporte* y definir los atributos apropiados. Por ejemplo, si se usara una implementación de JMS[26] como mecanismo de transporte, se ocuparía un atributo con el nombre de la cola de mensajes.

4.1.2 Mensaje

La clase *Mensaje* representa los mensajes enviados y recibidos por el canal de servicios. Esta clase es la clase base de todos los mensajes y contiene atributos para identificar el emisor y el receptor: dirección de transporte origen, número de servicio origen y número de servicio destino. Otros servicios como archivos o semáforos tienen sus clases particulares

de mensajes pero todos descienden de *Mensaje*. Esta clase implementa la interfaz *java.io.Serializable* y se puede usar el mecanismo de serialización de Java para convertir instancias de *Mensaje* a una secuencia de bytes y viceversa.

4.1.3 Conector de servicio

La clase *ConectorServicio* encapsula el conector de servicios. Esta clase tiene operaciones para enviar mensajes, recibir mensajes y cerrar la conexión. Una instancia de esta clase representa una conexión hacia otro proceso y tiene asociado un número de servicio. El canal enruta hacia esta instancia todos los mensajes entrantes cuyo número de servicio destino coincide con el número de servicio de esta instancia.

La operación *enviar(Mensaje)* utiliza el componente de transporte del canal para enviar un mensaje al extremo remoto de la conexión. El mensaje es depositado en una cola de mensajes salientes administrada por el componente de transporte. Este componente se encarga de enviar el mensaje asincrónicamente al destino especificado en el mensaje.

La operación *recibir* espera un mensaje del extremo remoto de la conexión. Para implementar esta operación, el conector de servicio maneja una cola de mensajes entrantes. Los mensajes recibidos por el componente de transporte son depositados asincrónicamente en esta cola. La operación *recibir* extrae el primer mensaje de la cola y lo retorna al cliente. Si no hay mensajes en la cola, esta operación queda bloqueada hasta que arribe un nuevo mensaje.

4.1.4 La interface *ICanal*

La interface *ICanal* define los métodos para usar el canal de servicios. Esta interface incluye los siguientes métodos.

DireccionTransporte getDireccionCanal(): retorna la dirección del canal.

aceptarConexiones(int maximoHilos): prepara el canal para recibir mensajes, activa el componente de transporte para aceptar conexiones y prepara el conjunto de hilos para atender solicitudes.

apagar: desactiva el componente de transporte y cierra las conexiones de transporte

activas.

registrarServicio(IServicio servicio): registra un servicio en el canal.

ConectorServicio conectar(DireccionTransporte destino): establece una conexión a la dirección *destino* y devuelve un conector de servicio con la conexión establecida.

La operación *aceptarConexiones* se invoca en la fase de inicialización del programa. La operación *apagar* se invoca en la fase de finalización del programa. Después de la fase de inicialización, un programa servidor puede invocar el método *registrarServicio* para registrar los servicios del nodo, y un programa cliente puede invocar el método *conectar* para obtener un conector de servicio y usar este conector para enviar y recibir mensajes.

Con la inteface *ICanal* termina la descripción de los elementos básicos del canal. En las siguientes secciones se explican los componentes principales del canal: serialización y deserialización, transporte, servicios y coordinador.

4.1.5 Serialización y deserialización

Se usa el mecanismo de serialización de Java para convertir instancias de mensajes a secuencias de bytes y viceversa. Para usar este mecanismo, la clase a serializar y deserializar debe implementar la interface *java.io.Serializable*.

```
ByteArrayOutputStream bas = new ByteArrayOutputStream();
ObjectOutputStream os = new ObjectOutputStream(bas);
os.writeObject(mensaje);
bytes = bas.toByteArray();
```

Figura 4.3. Serialización de un mensaje.

En la figura 4.3 se muestra el código en Java para serializar un mensaje: se encadena un flujo de salida de bytes (*ByteArrayOutputStream*) con un flujo de salida de objetos (*ObjectOutputStream*), el método *writeObject* convierte el mensaje a una secuencia de bytes y con el método *toByteArray* se obtienen los bytes del mensaje serializado.

```
ByteArrayInputStream bas = new ByteArrayInputStream(bytes);
ObjectInputStream os = new ObjectInputStream(bas);
objeto = (Serializable)os.readObject();
```

Figura 4.4. Deserialización de un mensaje.

En la figura 4.4 se muestra el código en Java para deserializar un mensaje: se encadena un flujo de entrada de bytes (*ByteArrayInputStream*) con un flujo de entrada de objetos (*ObjectInputStream*), el constructor de *ByteArrayOutputStream* recibe los bytes del mensaje serializado como parámetro y el método *ObjectInputStream.readObject* convierte la secuencia de bytes a un objeto.

La clase *dexplorer.canal.serializacion.Serializador* provee métodos estáticos para serializar y deserializar mensajes:

static public byte[] serializar(Mensaje mensaje): convierte un mensaje a una secuencia de bytes.

static public Mensaje deserializar(DataBuffer buffer): convierte una secuencia de bytes a un mensaje.

El mecanismo de serialización y deserialización convierte mensajes a secuencias de bytes y viceversa. En la siguiente sección se explica el componente de transporte, un componente usado para enviar y recibir secuencias de bytes entre canales.

4.1.6 Transporte

El componente de transporte del canal interactúa con el coordinador del canal a través de las siguientes interfaces: *ITransporte*, *IConectorTransporte* e *ICanalTransporte*. El componente de transporte implementa las primeras dos interfaces y el coordinador del canal implementa la interface *ICanalTransporte*.

1. *ITransporte* define las operaciones para administrar el componente de transporte. El coordinador invoca estas operaciones.

   arrancar(): contiene el código de inicialización del componente de transporte y es invocado por el coordinador durante la fase de inicialización del canal.

   aceptarConexiones(DireccionTransporte direccion): el coordinador invoca este método para indicarle al componente de transporte que acepte conexiones.

   apagar(): contiene el código de finalización del componente de transporte y es invocado por el coordinador durante la fase de finalización del canal.

IConectorTransporte conectar(DireccionTransporte direccionTransporte): el coordinador invoca este método para establecer una conexión a la dirección especificada en el parámetro *direccionTransporte*.

2. *IConectorTransporte*: define las operaciones de un conector de transporte.

   DireccionTransporte getDireccionRemota(): retorna la dirección del nodo que está en el extremo opuesto de la conexión.

   cerrar(): cierra la conexión.

   enviar(Mensaje mensaje): envía el mensaje al nodo remoto.

   La interfaz no incluye una operación para recibir mensajes. El componente de transporte opera de manera asincrónica: cuando recibe un mensaje, invoca el método *ICanalTransporte.procesarMensaje* para pasarle el mensaje al canal.

3. *ICanalTransporte*: define los métodos usados por el componente de transporte para notificar eventos de interés al coordinador.

   registrarConectorTransporte(IConectorTransporte conectorTransporte): este método es invocado por el componente de transporte para indicarle al canal de la aceptación de una nueva conexión. El parámetro *conectorTransporte* es el conector local asociado a esta conexión.

   procesarMensaje(IConectorTransporte conectorTransporte, Mensaje mensaje): el componente de transporte invoca este método para notificarle al coordinador sobre la llegada de un mensaje.

El componente de transporte se implementa con las clases de *java.nio.**. Estas clases soportan el protocolo TCP/IP en modo asincrónico. Este componente está en el *package dexplorer.canal.transporte.nio* y contiene las siguientes clases: *NioDireccion*, *NioConector*, *NioHiloTransporte* y *NioTransporte*.

La clase *NioDireccion* desciende de *DireccionTransporte* y representa la dirección de transporte. Tiene dos atributos: *host* es la dirección IP y *puerto* es el número de puerto.

La clase *NioConector* encapsula un *socket* e implementa *IConectorTransporte*. Maneja una

cola de bytes salientes y otra de bytes entrantes. Estas colas contienen los bytes de los mensajes serializados. Antes de cada mensaje serializado se agrega un número entero de 32 bits con la cantidad de bytes del mensaje serializado. Este número se usa para determinar si hay un mensaje completo en la cola de bytes entrantes.

La cola de bytes entrantes de *NioConector* es actualizada asincrónicamente por el hilo *NioHiloTransporte*. Este hilo administra un conjunto de *sockets*. Cuando se detecta que algún *socket* tiene bytes entrantes, el hilo invoca el método *NioConector.leer* para procesar estos bytes.

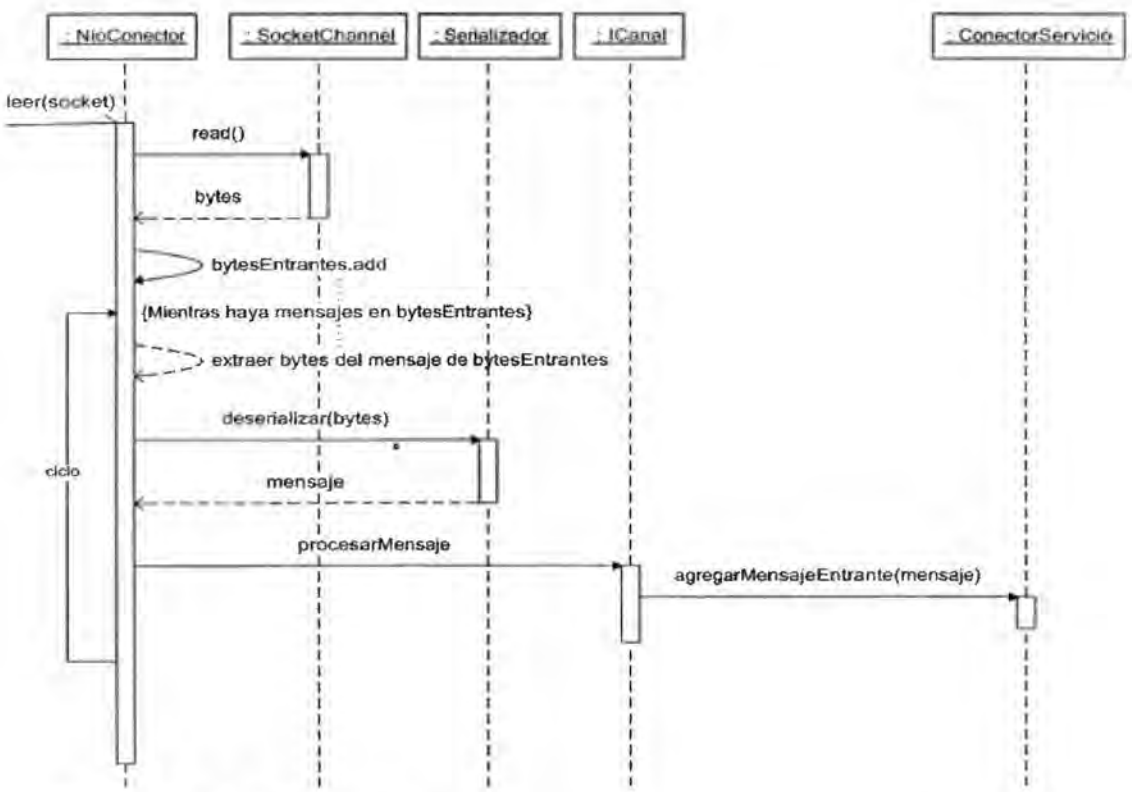

Figura 4.5. Recibir un mensaje.

En la figura 4.5 se muestra la invocación del método *NioConector.leer*. Este método recibe el *socket* como parámetro, lee los bytes del *socket*, lo agrega a la cola de bytes entrantes, y entra en un ciclo para extraer y procesar mensajes de la cola de bytes entrantes. Para determinar si hay un mensaje completo en la cola, se usa el hecho de que cada mensaje serializado va precedido de su largo. Si hay un mensaje completo, se extraen los bytes del

mensaje de la cola, se llama el método *Mensaje.deserializar* para convertir los bytes a un mensaje, y se invoca *ICanalTransporte.procesarMensaje* para procesar este mensaje. En este método se usa el número de servicio del mensaje para determinar el conector de servicio destino y se agrega el mensaje a la cola de mensajes entrantes del conector de servicio con la operación *agregarMensajeEntrante*. Para obtener este mensaje, el proceso cliente debe invocar el método *ConectorServicio.recibir*.

La operación *NioConector.enviar* actualiza la cola de bytes salientes.

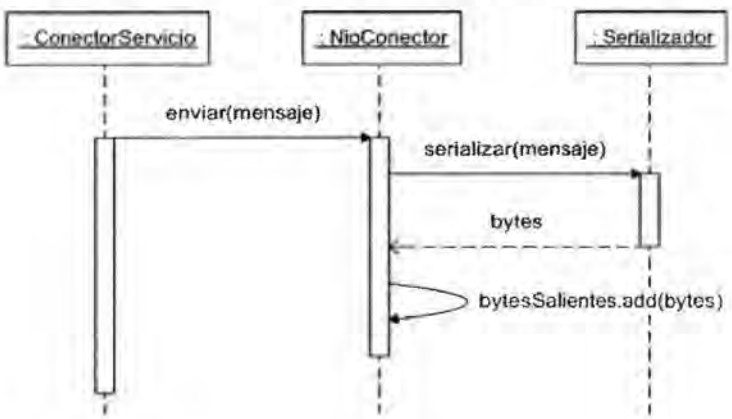

Figura 4.6. Enviar un mensaje.

En la figura 4.6 se muestra la implementación de la operación *NioConector.enviar*. El conector de servicio usa un objeto *IConectorTransporte* para enviar un mensaje. La clase *NioConector* implementa *IConectorTransporte*. El método *enviar* convierte el mensaje a bytes y lo deposita en la cola de bytes salientes. Los bytes depositados en esta cola son enviados asincrónicamente por el hilo de transporte.

La clase *NioHiloTransporte* implementa un hilo que administra una lista de *sockets*. Cada *socket* tiene asociado una instancia de *NioConector*. Cuando se detecta que un *socket* está listo para enviar, se usa este *socket* para enviar los bytes de la cola de bytes salientes del *NioConector* asociado a este *socket*. Cuando se detecta que un *socket* tiene bytes entrantes, se llama el método *NioConector.leer* del *NioConector* asociado a este *socket*.

La clase *NioTransporte* es la fachada del componente e implementa la interface *ITransporte*. La operación *conectar* retorna una instancia de *NioConector*.

El componente de transporte usa las clases de *java.nio.**, las cuales soportan el protocolo TCP/IP en modo asincrónico. Las clases más relevantes son:

- *SocketChannel*: implementa un *socket* de conexión. Una conexión está compuesta de un *socket* local y un *socket* remoto.
- *ServerSocketChannel*: implementa el *socket* de escucha, el cual se liga a un puerto y procesa solicitudes de clientes para establecer nuevas conexiones. Para cada solicitud se crea una nueva instancia de *SocketChannel* y se establece una conexión entre este *socket* y el *socket* del cliente.
- *Selector*: representa una cola de eventos de *sockets*. Un evento está compuesto por un tipo de operación y un *socket*. Las operaciones más relevantes son: leer, grabar y aceptar conexión. Los *ServerSocketChannel* y *SocketChannel* se registran al *Selector* y este detecta cuando ocurren eventos en cada *socket*.

En las siguientes figuras se explican el uso de las clases de *java.nio* en las clases *NioTransporte* y *NioHiloTransporte*.

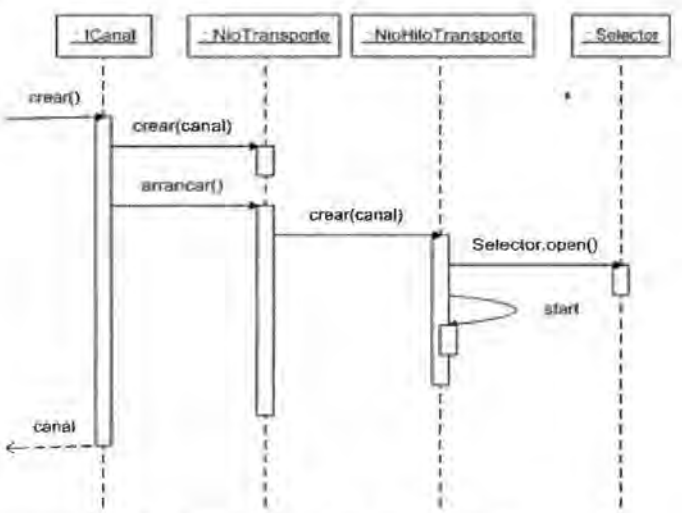

Figura 4.7. Implementación del método *NioTransporte.arrancar*.

En la figura 4.7 se muestra el funcionamiento de la operación *NioTransporte.arrancar*. El canal crea una instancia de *NioTransporte* e invoca el método *arrancar*. En este método se crea una instancia de *NioHiloTransporte*. En el constructor de esta clase se invoca

Selector.open para obtener un selector y se arranca el hilo encargado de obtener y procesar los eventos de los *sockets* ligados al selector.

```
If (selector.selectNow()) {
  Set<SelectionKey> eventos = selector.selectedKeys();
  // procesar eventos
}
```

Figura 4.8. Obtener eventos del *Selector*.

En la figura 4.8 se presenta el código en Java para obtener eventos del *Selector*. Para ello se usan dos métodos de la clase *Selector: selectNow* devuelve el número de eventos disponibles y *selectedKeys* devuelve una lista de tipo *SelectionKey* con los eventos.

La clase *SelectionKey* tiene métodos para manejar el evento: *channel* retorna el *socket* del evento, *isAcceptable* devuelve **true** si el evento corresponde a un cliente intentando conectarse, *isReadable* devuelve **true** si el *socket* del evento tiene bytes entrantes, *isWritable* devuelve **true** si el *socket* del evento está listo para enviar datos.

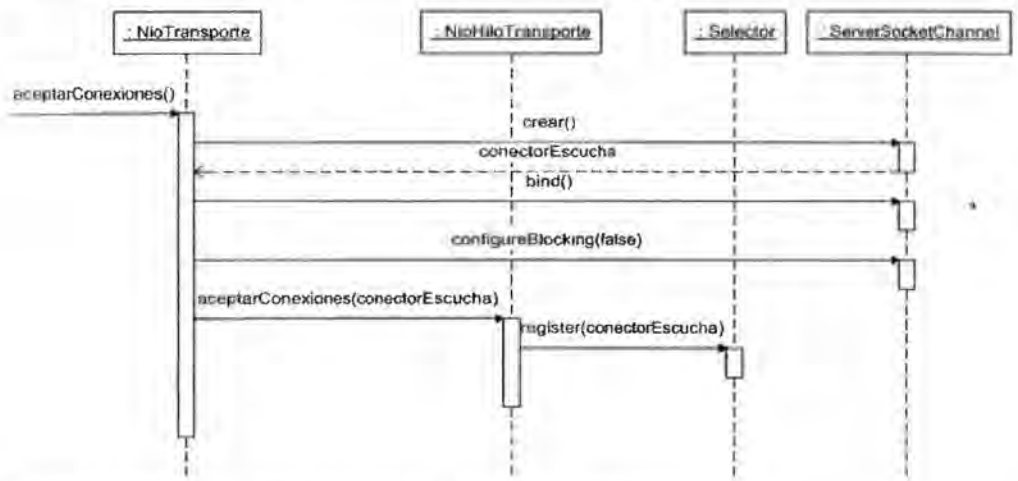

Figura 4.9. Implementación del método *ITransporte.aceptarConexiones*.

La figura 4.9 muestra el funcionamiento de la operación *aceptarConexiones*. Este método crea y administra un conector de escucha, el cual se liga al puerto de escucha y se configura para que opere asincrónicamente. Luego, el conector se pasa al hilo de transporte para que lo registre con el selector del hilo. A partir de este momento, si un cliente intenta conectarse al componente de transporte, el selector del hilo de transporte devuelve un evento de tipo *SelectionKey*. En este evento, si se invoca el método *isAcceptable* se obtiene **true**.

```
If (evento.isAcceptable()) {
  ServerSocketChannel conectorEscucha = (ServerSocketChannel)evento.channel();
  SocketChannel cliente = conectorEscucha.accept();
  NioConector conectorTransporte = crearRegistrarConector(cliente);
  canal.inscribirConectorTransporte(conectorTransporte);
}
```

Figura 4.10. Procesar una solicitud de conexión de un cliente.

La figura 4.10 muestra el código en Java para aceptar una conexión. El método *evento.channel* devuelve el conector de escucha. El método *conectorEscucha.accept* acepta la solicitud de conexión y devuelve el *socket* del cliente. El método *crearRegistrarConector* crea una nueva instancia de *NioConector* para encapsular esta conexión, asocia esta instancia de *NioConector* con el *socket* y registra el *socket* del cliente con el selector del hilo. A partir de este momento, el selector también devuelve eventos *isReadable* e *isWritable* de este *socket*. Finalmente, se invoca el método *canal.inscribirConector* para informarle al canal de que hay una nueva conexión.

Para procesar eventos de tipo *isReadable e isWritable*, primero se obtiene el *socket* del evento con el método *channel*, luego se obtiene el conector de transporte asociado al *socket*. Finalmente, si es un evento *isReadable*, se invoca el método *NioConector.leer* (ver figura 4.5), y si es un evento *isWritable*, se usa el *socket* del evento para enviar los bytes de la cola de bytes salientes del conector de transporte.

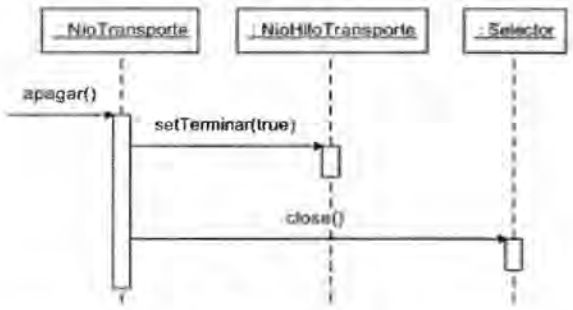

Figura 4.11. Implementación del método *ITransporte.apagar*.

En la figura 4.11 se muestra el funcionamiento del método *apagar*. Este método finaliza el hilo de transporte y cierra el selector. El hilo de transporte ejecuta un ciclo *while* para procesar eventos mientras el atributo *terminar* sea **false**. Para salir de este ciclo, se asigna **true** al atributo *terminar*.

Se explicó la implementación del componente de transporte usando el protocolo de transporte TCP/IP. En la siguiente sección se explica el componente de servicios.

4.1.7 Servicio

La clase abstracta *Servicio* es la clase base de los servicios del canal. De esta clase se derivan subclases que implementan servicios concretos como el servicio de archivos o el servicio de directorio. Las subclases deben implementar el siguiente método abstracto:

abstract public *Mensaje responder(Mensaje mensaje)* ***throws*** *Exception;*

La subclase debe implementar la lógica del manejador del servicio dentro de este método: debe procesar la solicitud contenida en el parámetro *mensaje* y retornar la respuesta en una instancia de alguna subclase de *Mensaje*.

La clase *Servicio* implementa la interfaz *IServicio*. Esta interfaz es usada por el coordinador del canal para interactuar con el servicio y contiene los siguientes métodos:

Runnable crearHiloServicio(ConectorServicio conectorServicio, Mensaje mensaje): el coordinador invoca este método para obtener el hilo encargado de procesar el mensaje. El coordinador controla la cantidad de hilos de servicio corriendo en un mismo momento. Si ya se alcanzó el número máximo permitido, el hilo recién creado pasa a una cola de pendientes.

setCanal(ICanal canal): el coordinador invoca este método cuando un servicio se registra al canal. De esta manera liga el servicio al canal.

getIdentificador(): el coordinador invoca este método para obtener el identificador o número del servicio.

getNombre(): el coordinador invoca este método para obtener el nombre del servicio.

getTipo(): el coordinador invoca este método para obtener el tipo del servicio.

Con el uso de la clase *Servicio*, el programador de servicios implementa la lógica del servicio en el método *responder* y no tiene que preocuparse de la comunicación. El programador recibe la solicitud como un parámetro del método y devuelve la respuesta

como resultado del método. El componente de servicios se encarga de enviar la respuesta al cliente que envió la solicitud.

En la siguiente sección se describe el coordinador, el componente encargado de integrar los diferentes componentes del canal.

4.1.8 Coordinador

El coordinador está implementado en la clase abstracta *Coordinador*. Las subclases concretas de esta clase tienen la responsabilidad de instanciar el componente de transporte (*ITransporte*). La subclase concreta *CoordinadorNio* instancia un componente de transporte que usa el protocolo TCP/IP.

La clase *Coordinador* implementa la interfaz *ICanal*. Esta interfaz define los métodos para el cliente del canal: *aceptarConexiones*, *registrarServicio*, *apagar*, etc.

El coordinador implementa sus funciones principales de la siguiente manera:

- Registro de servicios: los servicios registrados se mantienen en un vector de servicios. Las entradas de este vector son de tipo *IServicio*.
- Administración de conectores de servicios: los conectores de servicio se mantienen en un vector de tipo *ConectorServicio*. Hay conectores asignados a servicios y hay conectores creados por la lógica del cliente. Los primeros tienen una entrada no nula en la correspondiente entrada del vector de servicios (*IServicio*), los últimos tienen una entrada nula.
- Enrutamiento de mensajes: los mensajes entrantes son procesados en el método *ICanalTransporte.procesarMensaje*. El número de servicio destino del mensaje se usa como índice a los vectores de servicios y de conectores de servicio. Para determinar si el mensaje corresponde a un servicio, se revisa el vector de servicios usando el número de servicio destino como índice. Si el mensaje es para un servicio, se despacha usando el mecanismo de hilos de servicio; si el mensaje es para un conector de servicio, se deposita en la cola de mensajes entrantes del conector de servicio.

- Administración de solicitudes e hilos de servicio: Java provee clases para administrar hilos de trabajo. La clase *ExecutorService* implementa un administrador de hilos y permite definir una cota superior de hilos activos. El canal mantiene una única instancia de *ExecutorService*. En la lógica de *procesarMensaje*, cuando se recibe una nueva solicitud, se invoca *IServicio.crearHiloServicio* para obtener el hilo que va a procesar la solicitud y se pasa este hilo a la instancia de *ExecutorService* para que lo despache cuando lo considere conveniente.
- Administración de conexiones de transporte: las conexiones (TCP) activas se mantienen en un *TreeMap<DireccionTransporte,IConectorTransporte>*. Esta clase es un árbol binario ordenado por *DireccionTransporte*, la dirección de transporte. Permite localizar rápidamente un nodo usando la dirección de transporte de la conexión. Cuando se solicita una conexión a una dirección de transporte, si esta dirección está en el árbol, se reutiliza el conector de transporte, en caso contrario, se establece una nueva conexión y se inserta en el árbol.
- Administración de conectores de servicios: mantiene la lista de conectores de servicios. Hay conectores asignados a servicios y hay conectores creados por la lógica del cliente.

Con la explicación del coordinador concluye la sección del canal de servicios. En las siguientes secciones se explican la implementación de los servicios internos: directorio, descubrimiento, notificación, sincronización y archivos.

4.2 Servicio de directorio

El servicio de directorio mantiene una lista de servicios del sistema distribuido y una lista de grupos de nodos. En la lista de servicios, cada entrada contiene el nombre, el tipo y la dirección del servicio. El nombre es usado como código. No se permiten nombres repetidos en la lista. La lista de grupos mantiene el nombre del grupo y una lista con los integrantes y sus direcciones. Un nodo puede pertenecer a más de un grupo.

El servicio implementa las siguientes operaciones:

inscribirServicio(String nombre, String tipo, DireccionTransporte direccion): inscribe

el servicio *nombre* en el directorio. Si ya existe un servicio con el mismo nombre devuelve una excepción.

desinscribirServicio(String nombre): desinscribe el servicio *nombre* del directorio. Si este servicio no existe, devuelve una excepción.

modificarServicio(String nombre, String tipo, DireccionTransporte direccion): modifica el servicio *nombre* en el directorio. Si este servicio no existe, devuelve una excepción. Esta operación permite modificar el tipo o la dirección del servicio.

consultarServicioPorNombre(String nombre): busca el servicio *nombre* y retorna su tipo y su dirección. Si el servicio no existe, indica que no encontró un servicio con ese nombre.

ConsultarServicioPorTipo(String tipo): devuelve una lista con los servicios del tipo consultado. Puede devolver una lista vacía. La lista contiene el nombre, el tipo y la dirección de cada servicio.

int entrarGrupo(String grupo, DireccionTransporte direccion): agrega un nodo al grupo. Este método devuelve el identificador asignado al nodo, el cual es un entero consecutivo único que identifica este nodo dentro del grupo.

salirGrupo(String grupo, int identificador): elimina el nodo *identificador* del grupo. El parámetro *identificador* debe contener el valor retornado por la operación *entrarGrupo*.

ConsultarGrupo(String grupo): devuelve una lista con las direcciones de los nodos del grupo.

Las clases de este servicio están en el *package dexplorer.servicio.directorio*. Las clases principales de este servicio son:

Las clases *InformacionServicio* y *ListaServicios* implementan la lista de servicios.

Las clases *InformacionNodo, ListaNodos* implementan la lista de nodos.

Las clases *InformacionGrupo y ListaGrupos* implementan la lista de grupos.

La clase *ServicioDirectorio* implementa el servicio de directorio. Esta clase es una subclase de *Servicio* y sobrescribe el método *Servicio.responder(Mensaje)*. En la lógica

de este método, se analiza el mensaje de solicitud y se invoca el método respectivo de *ListaNodos*, *ListaServicios* o *ListaGrupos* para realizar la operación solicitada. Para registrar el servicio de directorio en un canal, se crea una instancia de *ServicioDirectorio* y se llama el método *ICanal.registrarServicio(servicioDirectorio)*.

La clase *ClienteDirectorio* implementa el objeto remoto con las operaciones del servicio de directorio. Para crear una instancia de *ClienteDirectorio* se ocupa conocer la dirección del servicio de directorio. Para obtener esta dirección, se puede usar el mecanismo de descubrimiento que se describe en la siguiente sección.

Después de explicar el servicio de directorio, en la siguiente sección se explica el servicio de descubrimiento.

4.3 Servicio de descubrimiento

El servicio de descubrimiento se usa para localizar el servicio de directorio en una LAN y está compuesto por dos interfaces: *IServidorDescubrimiento* e *IClienteDescubrimiento*.

La interfaz *IServidorDescubrimiento* define las operaciones para administrar el servidor de descubrimiento:

arrancar(): arranca el servidor de descubrimiento.

apagar(): apaga el servidor de descubrimiento.

La interfaz *IClienteDescubrimiento* contiene el método *DireccionTransporte descubrir()*. Este método localiza el servidor de descubrimiento y retorna la dirección del servicio de directorio.

Las clases de este servicio están en el *package dexplorer.descubrimiento*.

Las clases *ServidorBroadcast* y *ClienteBroadcast* implementan las interfaces anteriores usando el mecanismo de *broadcast* UDP como mecanismo de descubrimiento. En el nodo en donde está el servicio de directorio, se activa una instancia de *ServidorBroadcast* para que escuche *broadcasts* UDP en un puerto determinado. Para descubrir este nodo, un cliente crea una instancia de *ClienteBroadcast* e invoca el método *descubrir()*. Este método emite un *broadcast*. Cuando la instancia de *ServidorBroadcast* recibe este mensaje, le

responde con la dirección del servicio de directorio. Con la dirección del servicio de directorio, el cliente puede localizar otros servicios registrados en el directorio. Cada instancia de *ServidorBroadcast* tiene asociado un nombre de aplicación. El mensaje de *broadcast* emitido por el método *ClienteBroadcast.descubrir* también contiene un nombre de aplicación. La instancia de *ServidorBroadcast* solo responde los mensajes que contienen su nombre de aplicación. Esto permite tener varios servidores de descubrimiento activos en una LAN, pero cada servidor debe tener un nombre de aplicación diferente.

Para utilizar este mecanismo, en el servidor se debe crear un canal y registrar el servicio de directorio en el canal. Luego se debe crear una instancia de *ServidorBroadcast* pasándole la dirección por la cual está escuchando el canal, y llamar el método *arrancar()* para empezar a escuchar *broadcasts*. En el cliente se debe crear una instancia de *ClienteBroadcast* e invocar el método *descubrir()*, el cual devuelve la dirección del servicio de directorio. Con esta dirección se crea una instancia de *ClienteDirectorio* para consultar por otros servicios.

A continuación se describen detalles de implementación del servicio de notificación.

4.4 Servicio de notificación

El servicio de notificación implementa un mecanismo de notificaciones asincrónicas. Este servicio se registra en cada nodo del sistema distribuido. Los procesos locales interesados en recibir notificaciones deben registrar un manejador de notificaciones con este servicio. Los manejadores tienen asociado un nombre. Un proceso que envía una notificación debe indicar la dirección del canal destino y el nombre del manejador que va a procesar la notificación. Cuando el canal destino recibe la notificación, se pasa la notificación al servicio de notificaciones. Este servicio usa el nombre del manejador para buscar el manejador respectivo e invoca este manejador pasándole la notificación.

Las clases de este servicio están en el *package dexplorer.servicio.notificacion*. Los elementos más importantes de este servicio son la interfaz *IManejadorNotificacion*, la clase *ServicioNotificacion* y la clase *ModuloNotificacion*.

La interfaz *IManejadorNotificacion* contiene las siguientes operaciones:

String getNombre(): este método retorna el nombre del manejador.

void procesarNotificacion(Mensaje mensaje): este método es invocado por el servicio de notificaciones cuando recibe una notificación. El parámetro *mensaje* contiene la notificación.

La clase *ServicioNotificacion* implementa el servicio de notificación y contiene el método *registrarManejador(String nombre, IManejadorNotificacion manejador)*. Este método registra un manejador en el servicio de notificación.

La clase *ModuloNotificacion* maneja el servicio de notificaciones y provee los siguientes métodos para enviar notificaciones y administrar manejadores de notificaciones:

void agregarManejador(IManejadorNotificacion manejador): agrega un manejador de notificaciones en el módulo de notificaciones. El parámetro *manejador* es un objeto que implementa la interface *IManejadorNotificacion*. Esta interface define los métodos *void procesarNotificacion(Mensaje mensaje)* y *String getNombre()*. El método *getNombre* retorna el nombre del manejador. Cuando el módulo de notificaciones recibe una notificación con este nombre, se invoca el método *procesarNotificacion* pasándole como parámetro la notificación recibida. Este método debe procesar la notificación.

void eliminarManejador(IManejadorNotificacion manejador): elimina un manejador de notificaciones del módulo de notificaciones.

DestinoNotificacion getDestino(IManejadorNotificacion manejador): retorna la dirección de notificación correspondiente al parámetro *manejador*. Esta dirección está compuesta por la dirección del canal de servicios y el nombre del manejador (*IManejadorNotificacion.getNombre*). Por convención, el cliente es responsable de proveer la dirección de notificación en donde espera recibir las notificaciones. Cuando el proceso cliente solicita un servicio asíncrono, en el mensaje de solicitud se debe agregar la dirección de notificación para que el servicio pueda enviar la notificación de respuesta al destino correcto.

void notificar(DestinoNotificacion destino, Mensaje mensaje): envía un mensaje de notificación al destino indicado. El parámetro *destinoNotificacion* contiene la dirección hacia donde enviar el mensaje. El parámetro mensaje es la notificación a enviar.

El mecanismo de notificaciones funciona de la siguiente manera:

1. El cliente registra el servicio de notificaciones en su canal.
2. El cliente registra un manejador de notificaciones en el servicio de notificación.
3. El cliente solicita un servicio a algún servicio remoto. Como consecuencia de esta solicitud, el servicio envía una notificación al servicio de notificación del cliente.
4. El servicio de notificación del cliente recibe la notificación e invoca el manejador registrado en el paso 2.

En esta sección se explicó la implementación del servicio de notificación. Seguidamente se explica la implementación del servicio de sincronización.

4.5 Servicio de sincronización

El servicio de sincronización implementa dos mecanismos de sincronización: semáforos y barreras. Los semáforos se usan para implementar regiones críticas y las barreras son puntos de contención en donde los procesos de un grupo deben esperar hasta que todos los procesos hayan alcanzado este punto. Ambos mecanismos utilizan el enfoque del servidor centralizado pero manejan múltiples semáforos y barreras, cada uno identificado por un nombre único. Los semáforos son n-arios, y en particular pueden ser binarios. Un único servicio de sincronización puede atender las necesidades de todo el sistema distribuido.

El servicio está compuesto por las siguientes operaciones:

int abrirSemaforo(String nombre, int cantidad): crea o abre un semáforo n-ario. Retorna un identificador único asociado a este semáforo. Este identificador es usado en las operaciones *cerrarSemaforo*, *adquirirSemaforo* y *liberarSemaforo*.

int abrirSemaforo(String nombre): crea o abre un semáforo binario.

cerrarSemaforo(int semaforo): cierra el semáforo.

adquirirSemaforo(int semaforo): solicita adquirir el semáforo. Esta operación bloquea el proceso actual si el semáforo fue adquirido por otro proceso.

liberarSemaforo(int semaforo): libera el semáforo. Si hay procesos pendientes

esperando por el semáforo, le asigna el semáforo al primero de la lista.

barrera(String nombre, int cantidad): el parámetro *nombre* contiene el nombre de la barrera y *cantidad* contiene el número de procesos que deben llegar a esta barrera para liberar todos los procesos. Los primeros *cantidad-1* procesos que invoquen este método quedan bloqueados. Finalmente, cuando el proceso número *cantidad* invoque este método, se desbloquean todos los procesos y pueden continuar con la ejecución de su lógica después de la barrera.

Un semáforo tiene un nombre, un identificador, el número de procesos que han abierto el semáforo, el valor del semáforo, el valor máximo y una lista de solicitudes pendientes. El valor del semáforo fluctúa entre cero y el máximo. Si el valor está en cero, significa que el semáforo ha sido adquirido por algún proceso.

El servicio de sincronización mantiene una lista de semáforos indexados por el identificador, y un consecutivo de identificadores de semáforos. Cuando se abre un semáforo por primera vez, se incrementa este consecutivo y se asigna este número al semáforo. Para determinar si el semáforo ya fue abierto, el servicio hace una búsqueda en la lista de semáforos usando el nombre del semáforo.

Una barrera tiene un nombre, un identificador el valor de la barrera y el valor máximo. El valor máximo es la cantidad de procesos a esperar. El valor de la barrera se inicializa en cero y se incrementa con cada invocación del método *barrera*. Si el valor de la barrera es menor que el máximo, se bloquea el proceso que invocó el método *barrera*. Si alcanza el máximo, el proceso puede continuar y además se desbloquean los otros procesos.

Las clases de este servicio están en el *package dexplorer.servicio.sincronizacion*. Las clases principales de este servicio son:

Semaforo y *Semaforos*: implementa la lógica de un semáforo y la lista de semáforos respectivamente.

Barrera y *Barreras*: implementan la lógica de una barrera y la lista de barreras respectivamente.

ServicioSincronización: es una subclase de *Servicio*. Mantiene instancias de *Barreras* y de

Semaforos, y actúa como coordinador: analiza el mensaje entrante e invoca el método respectivo de *Semaforos* o de *Barreras* para que procese el mensaje.

ClienteSemaforo implementa el objeto remoto para usar las operaciones del servicio de sincronización.

En esta sección se explicó el servicio de sincronización. En la siguiente sección se presentan detalles de implementación del servicio de archivos.

4.6 Servicio de archivos

El servicio de archivos implementa operaciones sencillas para accesar archivos remotos de parámetros y de resultados. El servicio no maneja directorios sino que todos los archivos se guardan en un mismo directorio. Implementa el mecanismo MRSW (*Multiple Readers-Single Writer*), en donde un mismo archivo puede ser abierto por varios lectores ó por un único escritor.

El servicio está compuesto por las siguientes operaciones:

int abrirLectura(String nombre): abre un archivo para lectura. Si algún otro proceso abrió este mismo archivo para escritura, se bloquea el proceso actual. Devuelve el identificador único del archivo.

int abrirEscritura(String nombre): crea o abre un archivo para escritura. Si algún otro proceso abrió este archivo, sea para escritura o para lectura, se bloquea el proceso actual. Devuelve el identificador único del archivo.

cerrarBloque(int archivo): cierra el archivo. Si hay procesos bloqueados, desbloquea el primer proceso de la lista. Si el primer proceso solicitó abrir el archivo para lectura, también desbloquea otros procesos que desean abrir el archivo para lectura.

byte[] leerBloque(int archivo): lee el contenido del archivo y lo retorna en un vector de bytes.

grabarBloque(int archivo, byte[] bloque): graba el *bloque* en el archivo.

Un archivo tiene un nombre, un identificador, y un campo que indica si fue abierto para

lectura o para escritura.

El servicio mantiene una lista de archivos indexados por el identificador, y un consecutivo de identificadores de archivos. Cuando se abre un archivo por primera vez, se incrementa este consecutivo y se asigna este número al archivo. Para determinar si el archivo ya fue abierto, el servicio hace una búsqueda secuencial en la lista de archivos usando el nombre.

Las clases de este servicio están en el *package dexplorer.servicio.archivo*. Las clases principales de este servicio son:

Archivo y Archivos: implementa la lógica de un archivo y la lista de archivos respectivamente.

ServicioArchivo: es una subclase de *Servicio*. Mantiene una instancia de *Archivos* y actúa como coordinador: analiza el mensaje entrante e invoca el método respectivo de *Archivos* para que procese el mensaje.

El servicio de archivos implementa cuatro objetos remotos: *ClienteLecturaDatos*, *ClienteEscrituraDatos*, *ClienteLecturaTexto* y *ClienteEscrituraTexto*.

La clase *ClienteLecturaDatos* implementa un flujo de lectura de datos. Esta clase abre el archivo remoto para lectura y lee el contenido del archivo a un buffer local. También tiene operaciones como *getInt*, *getDouble* para leer datos del buffer local.

La clase *ClienteEscrituraDatos* implementa un flujo de escritura de datos. Esta clase abre el archivo remoto para escritura y crea un buffer local de datos. También tiene operaciones como *putInt*, *putDouble* para escribir datos al buffer local. Cuando se cierra el flujo, se graba el buffer local de datos al archivo remoto.

La clase *ClienteLecturaTexto* implementa un flujo de lectura de texto. Esta clase abre el archivo remoto para lectura y lee el contenido del archivo a un buffer local. También tiene la operación *readLine* para leer una línea de texto del buffer local.

La clase *ClienteEscrituraTexto* implementa un flujo de escritura de texto. Esta clase abre el archivo remoto para escritura y crea un buffer local de texto. Con la operación *writeLine* se graba una línea de texto al buffer local. Cuando se cierra el flujo, se graba el buffer local de

datos al archivo remoto.

Con el servicio de archivos concluye el capítulo sobre implementación de *D-Explorer*, en donde se explican detalles de implementación del canal de servicios y de los servicios internos. En el siguiente capítulo se describe el material de apoyo de *D-Explorer*.

5 Material de apoyo

El material de apoyo fue hecho con el fin de facilitar el aprendizaje de *D-Explorer*. Este material está almacenado en un archivo comprimido (*.zip*) e incluye el código fuente, las herramientas empacadas en una biblioteca (*.jar*), la documentación de clases en un formato similar a la documentación de clases de *Java* y varios tutoriales sobre el uso de las herramientas. El material de apoyo está organizado en un conjunto de páginas *html* con enlaces convenientes a los diferentes elementos.

Hay tres tutoriales que enseñan el uso de *D-Explorer*. Cada uno consiste de una aplicación distribuida, con el código fuente y el código ejecutable, instrucciones para ejecutar la aplicación y explicaciones de las diferentes clases de la aplicación.

El *tutorial 01* explica la implementación de una aplicación distribuida sencilla. En este tutorial se explica el uso de los elementos básicos de las herramientas: el canal de servicios, el mecanismo de descubrimiento y el servicio de directorio. También explica la implementación de un servicio sencillo. El servicio implementado es una calculadora que suma números enteros. La aplicación está compuesta por el servidor principal y el programa cliente. El programa cliente envía dos números al servidor. El servidor realiza la suma y devuelve el total al cliente.

El *tutorial 02* explica el uso del servicio de notificaciones para implementar operaciones que no bloquean el cliente. El servicio implementado, cuanto recibe una solicitud, le responde inmediatamente al cliente acusando el recibo de la solicitud y despacha un hilo para hacer el cálculo solicitado. El resultado del cálculo es enviado al cliente a través del servicio de notificaciones.

En el *tutorial 03* se implementan dos servicios alojados en dos programas diferentes y la manera como interactúan estos servicios: el cliente usa el servicio alojado en el servidor y el servidor usa el servicio del cliente. Este tutorial también muestra el uso del mecanismo de búsqueda de puertos libres: el servidor usa un puerto de escucha predefinido pero los clientes busca el próximo puerto libre para evitar conflictos entre ellos.

Después de describir el material de apoyo, en el siguiente capítulo se explica el proceso de validación funcional de las herramientas.

6 Validación de la funcionalidad

La validación funcional de *D-Explorer* tiene la finalidad de demostrar que las herramientas han alcanzado un nivel de correctitud adecuado para que sean usadas por terceros para validar su usabilidad en ambientes académicos. La presencia de defectos en *D-Explorer* podría arruinar o sesgar la validación de usabilidad, el cual, como se explicará en el siguiente capítulo, es un proceso complicado que consume mucho tiempo.

En la validación funcional, primero se valida la correctitud del canal de servicios y de cada servicio interno por separado, y luego se desarrolla una aplicación distribuida usando componentes de *D-Explorer*. En el resto del capítulo se explican estas dos actividades.

6.1 Evaluación individual de cada componente

En el proceso de evaluación individual de los componentes se valida que cada componente funcione correctamente en ambientes concurrentes antes de permitir su uso en el desarrollo de una aplicación distribuida completa. Este proceso está relacionado con el cuarto objetivo específico: verificar la correctitud de los componentes distribuido mediante un conjunto de pruebas. En este proceso se diseña, implementa y ejecuta una batería de pruebas para el canal de servicios y los servicios de directorio, notificaciones, sincronización y archivos.

En el diseño de las baterías de pruebas de cada componente se debe incorporar diferentes niveles de concurrencia y los resultados para determinar la correctitud de la prueba deben ser independientes de la concurrencia, es decir, no pueden variar debido a la concurrencia.

A continuación se describen las actividades realizadas para validar el canal de servicios. Las actividades realizadas para validar los otros componentes son bastantes similares a las realizadas para validar el canal de servicios y se omiten en este documento.

6.1.1 Validación del canal de servicios

En las pruebas del canal de servicios se usan cuatro tipos de procesos: el servicio *eco*, un servicio que recibe un mensaje de texto y devuelve el mismo mensaje texto, el servicio *mayúscula*, el cual recibe un mensaje de texto y devuelve el mismo mensaje convertido a

mayúsculas, el cliente *eco*, un proceso que envía 500 mensajes al servicio eco y espera las respuestas, y el cliente *mayúscula*, un proceso que envía 500 mensajes al servicio *mayúscula* y espera las respuestas. Estos procesos usan el canal de servicios para enviar y recibir mensajes. Se diseñan cuatro pruebas con diferentes niveles de concurrencia.

En la primera prueba se involucra un servicio *eco*, un cliente *eco* y un canal de servicios. El servicio *eco* se registra en el canal y el proceso *eco* usa el mismo canal para enviar y recibir mensajes del servicio *eco*. En esta prueba no hay concurrencia sino que el cliente se comunica de manera secuencial y sincrónica con el servicio *eco*.

La segunda prueba es una variante de la primera prueba que incorpora tres clientes *eco* corriendo concurrentemente. El servicio *eco* recibe y procesa solicitudes concurrentemente.

En la tercera prueba se usa un servicio *eco*, tres procesos clientes *eco* corriendo concurrentemente y cuatro canales de servicios, un canal para cada proceso. El servicio *eco* se registra en un canal y cada uno de los clientes *eco* usa su propio canal.

En la última prueba se usa un servicio *eco*, un servicio *mayúscula*, tres clientes *eco*, tres clientes *mayúscula* y siete canales de servicios. Los dos servicios se registran en un mismo canal y cada uno de los clientes tiene su propio canal. En esta prueba se incorpora concurrencia a nivel de servicios y de clientes: los seis clientes se ejecutan al mismo tiempo y los dos servicios procesan solicitudes concurrentemente.

Para determinar el resultado de una prueba, se evalúan las siguientes condiciones:

1. El cliente *eco* verifica que el mensaje enviado es idéntico al mensaje retornado por el servicio *eco*.
2. El cliente *mayúscula* verifica que el mensaje enviado convertido a mayúscula es idéntico al mensaje retornado por el servicio *mayúscula*.
3. Al final de la prueba, se verifica que el total de mensajes enviados por los clientes es igual al total de mensajes recibidos por los servicios.

Estas tres condiciones se evalúan durante la ejecución de cada una de las cuatro pruebas descritas anteriormente. Con las condiciones 1 y 2 se valida que el canal de servicios envía

y recibe correctamente los mensajes, es decir, el canal de servicios no corrompe los mensajes enviados. Con la condición 3 se valida que el canal de servicios no pierde ni agrega mensajes. Si no se cumple alguna de estas condiciones, la prueba reporta un error. En estos casos, se busca el defecto que ocasionó el error, se corrige el mismo y se repite la prueba hasta que cumpla con las tres condiciones descritas.

En esta sección se explicó la evaluación funcional de los componentes. La certeza de que estos funcionan correctamente en ambientes concurrentes permite su uso para implementar una aplicación distribuida. En la siguiente sección se explica esta actividad.

6.2 Uso integrado de los componentes de *D-Explorer*

En el uso integrado de los componentes de *D-Explorer* se desarrolla una aplicación distribuida de consultas para validar la interacción entre los componentes y la capacidad de las herramientas para implementar aplicaciones distribuidas. Este proceso está relacionado con el quinto objetivo específico: implementar un prototipo de aplicación distribuida o de algoritmo distribuido. Seguidamente se describe el diseño y la implementación del sistema de consultas, las pruebas realizadas sobre este sistema y los resultados obtenidos.

6.2.1 Sistema de consultas

El sistema de consultas es una aplicación distribuida en una LAN que resuelve consultas de clientes y tiene tres tipos de nodos: cliente, servidor principal y servidor auxiliar. En el procesamiento de una consulta participan el servidor principal y los servidores auxiliares. Este enfoque permite usar recursos de diferentes nodos para procesar una consulta. En la figura 6.1 se muestra la arquitectura de este sistema.

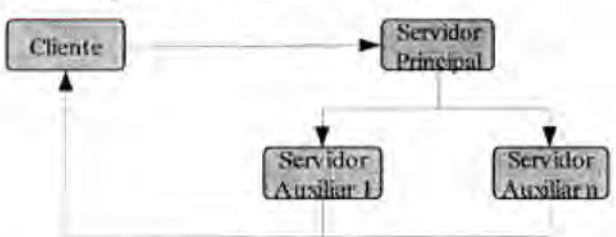

Figura 6.1. Arquitectura del sistema de consultas. La aplicación contiene tres tipos de nodos: cliente, servidor principal y servidor auxiliar. Las flechas indican el flujo de mensajes para resolver una consulta originada en el cliente.

En el sistema de consultas, una consulta se procesa de la siguiente manera:

- El cliente envía una consulta al servidor principal.
- El servidor principal analiza la consulta, distribuye el trabajo entre los servidores auxiliares disponibles y le responde al cliente indicándole que su consulta está siendo procesada.
- Cada servidor auxiliar involucrado en la consulta realiza su parte del trabajo y envía su parte de la solución al cliente.
- El cliente recibe las diferentes partes de la solución. Cuando ha recibido todas las partes, ensambla la respuesta completa.

En la implementación del sistema de consultas se desarrollan tres tipos de programas: *Servidor Principal*, *Servidor Auxiliar* y *Cliente*. En la figura 6.2 se presenta la implementación del sistema usando *D-Explorer*.

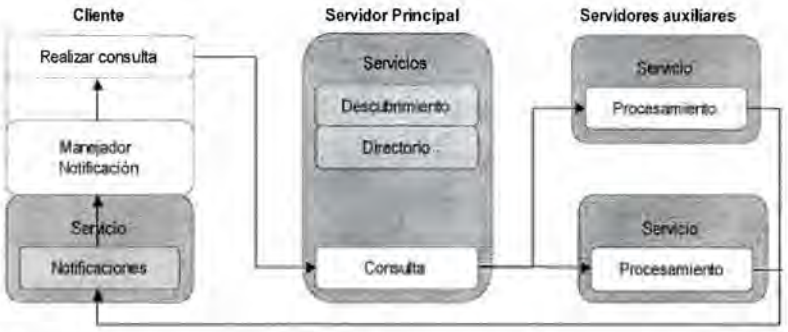

Figura 6.2. Implementación del sistema de consultas. Los componentes de *D-Explorer* se muestran en tono gris y los componentes particulares de la aplicación en blanco. Una consulta se inicia en el componente *Realizar consulta* del cliente. Las flechas indica el flujo de mensajes, entre los componentes de la aplicación, para resolver la consulta del cliente.

El servidor principal aloja los servicios de descubrimiento, directorio y consulta. El servicio de consulta se registra en el directorio y funciona de la siguiente manera: recibe una consulta del cliente, obtiene el conjunto de registros que satisfacen las condiciones de la consulta, localiza los servidores auxiliares existentes, les envía una parte de los registros a cada servidor auxiliar y le responde al cliente indicándole que su consulta está siendo procesada. Para simplificar la lógica del servicio de consulta, los registros que satisfacen

una consulta son generados aleatoriamente. Esta simplificación no tiene efectos negativos en las pruebas para validar *D-Explorer*.

Un servidor auxiliar contiene un servicio de ordenamiento, el cual es registrado en el directorio del servidor principal. Este servicio recibe un conjunto de registros y la dirección del cliente, ordena los registros y los envía al servicio de notificación que está operando en la dirección del cliente.

El programa cliente contiene el servicio de notificaciones y registra un manejador con este servicio para recibir las respuestas de los diferentes servidores auxiliares. El programa cliente verifica que las respuestas recibidas estén ordenadas, y en caso negativo, reporta un error. Para obtener el resultado final, se aplica el *mergeSort* [34] sobre las diferentes partes.

Después de explicar las características del sistema de consultas, en la siguiente sección se explican las pruebas para validar el sistema.

6.2.2 Pruebas

Una prueba del sistema de consultas consiste en ejecutar, en este orden, un servidor principal, uno o más servidores auxiliares y uno o más clientes, y luego realizar consultas a través de los clientes. El resultado de la prueba es determinado por el programa cliente: si la respuesta de una consulta es incorrecta, el cliente emite un error. Se diseñan cuatro grupos de pruebas para validar diferentes aspectos de *D-Explorer*.

En la primera prueba se ejecutan un servidor principal, un servidor auxiliar y un cliente en la misma computadora. El cliente emite una consulta y valida la respuesta de la consulta. Esta prueba se usa para depurar el sistema de consultas. Una vez que este sistema funciona correctamente, se realizan las otras pruebas.

En el segundo grupo de pruebas se ejecutan un servidor principal, un servidor auxiliar y un cliente. Los programas se ejecutan en diferentes máquinas de una LAN. En cada prueba de este grupo, el servidor principal se ejecuta en una máquina diferente con el objetivo de probar el mecanismo de descubrimiento. Si este mecanismo funciona correctamente, los servidores auxiliares y los clientes deberían localizar el servidor principal sin importar en donde esté corriendo este.

En el tercer grupo de pruebas se ejecutan un servidor principal, un servidor auxiliar y un cliente. El cliente emite una consulta y valida la respuesta. Seguidamente, se activa un nuevo servidor auxiliar y se realiza una nueva consulta. Finalmente, se elimina un servidor auxiliar y se realiza una consulta desde el cliente. El objetivo de esta prueba es validar el funcionamiento del servicio de directorio. Cuando se activa o se desactiva un servidor auxiliar, este debe inscribirse o desinscribirse del servicio de directorio. Posteriormente, cuando el servidor principal recibe una consulta, este busca los servidores auxiliares activos en el directorio e involucra estos servidores auxiliares en el procesamiento de la consulta.

En el último grupo de pruebas se ejecuta un servidor principal, un servidor auxiliar y un cliente. Los programas se ejecutan en tres computadoras de la red con diferentes tipos de sistemas operativos. Una computadora tiene Windows XP, otra tiene Windos Vista y la tercera tiene Linux. El objetivo de esta prueba es verificar el funcionamiento del sistema de consultas en una red heterogénea.

A continuación se explican los resultados obtenidos de estas pruebas.

6.2.3 Resultados

Se ejecutan los cuatro grupos de pruebas descritos en la sección anterior. En cada prueba realizada, el cliente indica que la respuesta de la consulta es correcta. Este resultado indica que los componentes de *D-Explorer* usados en el sistema de consultas interactuaron correctamente para resolver la consulta: los clientes y los servidores auxiliares localizaron el servidor principal usando el servicio de descubrimiento, el servidor principal usó el servicio de directorio para localizar los servidores auxiliares activos y los servidores auxiliares enviaron sus respuestas al cliente a través del servicio de notificaciones. Al margen de los resultados de correctitud, la aplicación desarrollada presenta otras características interesantes que valen la pena mencionar:

- Con las pruebas del servicio de descubrimiento y del servicio de directorio se comprueba que la aplicación presenta independencia de localización: no existen referencias estáticas con las direcciones de cada programa (servidor principal, servidor de consulta y cliente) sino que estas direcciones son obtenidas en tiempo de

ejecución usando el mecanismo de descubrimiento y el servicio de directorio. El servidor principal puede correr en cualquier nodo de la red y se usa el mecanismo de descubrimiento mediante *broadcast UDP* para localizarlo. Los servicios de consulta y de ordenamiento pueden estar en cualquier nodo de la red y son localizados usando el servicio de directorio.

- Con las pruebas del servicio de directorio se valida, indirectamente, que la aplicación es escalable y distribuye la carga de trabajo entre varios nodos. El servidor principal se encarga de seleccionar los registros que calzan con las condiciones de la consulta. Los servidores auxiliares se encargan de ordenar una parte del resultado. El cliente se encarga de ensamblar el resultado final aplicando un *mergesort* sobre las particiones ordenadas. Además, el sistema permite incorporar nuevos servidores auxiliares dinámicamente.
- La aplicación es muy desacoplada. Aunque una consulta puede involucrar varios servidores, el cliente solo necesita localizar al servidor principal y no tiene referencias o conexiones directas con los servidores auxiliares. Los servidores auxiliares tampoco necesitan establecer conexiones directas con el módulo cliente sino que usa el servicio de notificaciones para hacerle llegar los resultados. En ambos casos, se introduce una indirección y se elimina la necesidad de establecer conexiones directas.
- Las herramientas facilitan el desarrollo de prototipos. La interacción entre los componentes se realiza usando las capacidades de las herramientas: el mecanismo de descubrimiento, el servicio de directorio y el servicio de notificaciones. La mayoría del código desarrollado se enfoca en funcionalidad (seleccionar registros, ordenar registros) y no en comunicación o interacción entre componentes.

En este capítulo se presentó la validación funcional de *D-Explorer*. Los resultados de las baterías de pruebas de los componentes y de las pruebas del sistema de consultas presentan evidencia razonable de la correctitud de *D-Explorer*. El nivel de correctitud alcanzado permite usar las herramientas para la implementación de aplicaciones por parte de los estudiantes. En el siguiente capítulo se explica la validación de la utilidad de *D-Explorer*.

7 Preparación de la validación de utilidad

En este capítulo se describen los preparativos previos al proceso de validación de la utilidad de *D-Explorer* como herramienta didáctica en ambientes académicos. El proceso de validación está relacionado con el cumplimiento del último objetivo específico: verificar la utilidad de las herramientas en ambientes académicos.

La metodología general de validación consta de dos etapas: preparación y ejecución. Esta separación es requerida pues la planificación de la etapa de ejecución depende de los resultados de la etapa de preparación. En este capítulo se describe la etapa de preparación y los resultados obtenidos en esta etapa. En el siguiente capítulo se presenta la etapa de ejecución. En el resto de este capítulo se describe el enfoque general de validación, la metodología general de validación y las actividades realizadas en la etapa de preparación.

7.1 Enfoque general

El enfoque general de validación de la utilidad de *D-Explorer* consiste en asignar tareas programadas que requieran el uso de *D-Explorer* a estudiantes de la Maestría en Computación e Informática de la UCR y medir el impacto del uso de *D-Explorer* en la implementación de las tareas mediante la aplicación de una encuesta a los estudiantes y el análisis del código fuente de las tareas implementadas.

La asignación de tareas programadas presenta varias ventajas: permite resolver diferentes tipos de problemas distribuidos, el proceso ocurre en un ambiente académico, hay interés en los involucrados (estudiantes) en completar la tarea pues forma parte de la nota del curso. La mayor desventaja de este enfoque es que hay muchos factores ajenos a *D-Explorer* que pueden incidir en el número de horas usadas por el estudiante para implementar la tarea programada, como por ejemplo la habilidad de programación del estudiante, la asignación de otros proyectos o exámenes en el mismo periodo de implementación de la tarea, factores sociales o laborales que afectan el rendimiento del estudiante, etc. El uso de la tarea programada también introduce una limitación en el experimento: la recopilación de observaciones es un proceso lento. Para obtener una

encuesta, se debe asignar una tarea programada y dar un plazo razonable para que el estudiante concluya la tarea. En un curso semestral se puede asignar como máximo tres o cuatro tareas, sin embargo, por limitaciones del curso este número podría ser menor.

Las ventajas de la encuesta como mecanismo de medición son las siguientes:

1. La encuesta en un instrumento flexible que permite obtener diferentes tipos de información como el perfil del estudiante (*sockets*, hilos, etc) o los resultados de una tarea programada (duración en horas, dificultad, etc).
2. La encuesta en un instrumento fácil de aplicar que no requiere una preparación previa del encuestado. El hecho de que el encuestado tenga que prepararse previamente puede introducir un sesgo en el experimento pues se desea que el estudiante pueda completar la tarea programada consultando la distribución de *D-Explorer* pero sin necesidad de estudiar esta distribución como si fuera a realizar un quiz o un examen.
3. La encuesta se realiza en un ambiente de poca presión. A diferencia de un quiz o un examen, los encuestados no tiene límites de tiempo y la encuesta no forma parte de la nota del curso. Estos factores permite obtener respuestas más confiables pues los encuestados no obtienen beneficios o castigos de sus respuestas.

En el análisis del código se determina la cantidad del código de *D-Explorer* en una tarea programada. Este dato mide la usabilidad de *D-Explorer*, pues se espera que entre menor sea esta cantidad de código, mayor sea la facilidad de uso de las herramientas. Además, este mecanismo permite diferenciar entre la lógica distribuida y la lógica no distribuida de la tarea. Por el contrario, en la encuesta el estudiante estima la duración en horas para completar la tarea. Esta duración indica la dificultad de la tarea pero no diferencia entre la lógica distribuida y la lógica no distribuida. Un estudiante puede durar mucho debido a la complejidad en la lógica distribuida, en la lógica no distribuida o en ambos tipos de lógica.

Después de explicar el enfoque general de validación, la siguiente sección describe la metodología general usada para implementar este enfoque.

7.2 Metodología General

La validación de la usabilidad de *D-Explorer* consta de una etapa de preparación y una etapa de ejecución, y cada etapa se divide, a su vez, en una serie de pasos:

1. Preparación. Esta etapa se debe completar antes de proceder con la segunda etapa.
    1. Seleccionar varios cursos en donde se puedan asignar problemas distribuidos.
    2. Diseñar una o más tareas programadas para cada curso seleccionado.
    3. Diseñar una encuesta para observar métricas de usabilidad.
1. Ejecución.
    1. Implementar tareas programadas. Los estudiantes de los cursos seleccionados usan *D-Explorer* para resolver las tareas programadas diseñadas. Se imparte una charla de *D-Explorer* cuando se asigna la primera tarea a los estudiantes y se aplica una encuesta a los estudiantes cuando entregan la tarea.
    2. Analizar las encuestas. Después de que los estudiantes han implementado todas las tareas, se analizan las encuestas y se presentan los resultados del análisis.
    3. Analizar el código fuente. Después de que los estudiantes han implementado todas las tareas, se analiza el código fuente de algunas tareas y se presentan los resultados del análisis.

A continuación se describen las actividades realizadas durante la etapa de preparación: selección de cursos, diseño de tareas programadas y diseño de encuestas.

7.3 Selección de cursos

El criterio principal para seleccionar un curso es la factibilidad de asignar problemas distribuidos como tareas programadas en el curso. Los cursos seleccionados son *PF-3327 Sistemas Distribuidos* y *PF-3392 Programación Java para Ambientes Distribuidos*, ambos cursos impartidos en el II ciclo del 2008. El curso de *Sistemas Distribuidos* (*SD*) es el candidato natural para usar *D-Explorer* pues las herramientas están orientadas al aprendizaje y desarrollo de algoritmos y aplicaciones distribuidas. El curso *Programación*

Java para Ambientes Distribuidos (*PJAD*) también es un buen candidato pues como parte del curso se estudian *sockets*, *RMI* y otros tipos de *middleware* y permite realizar una comparación de *D-Explorer* contra estas herramientas.

La escogencia del II ciclo del 2008 fue una imposición de las circunstancias. Estos cursos no son impartidos semestralmente sino que se imparten una vez al año o cada tres semestres. Si no se hubiera seleccionado los cursos de este semestre, el proyecto podría sufrir un retraso de al menos un año. Sin embargo, esta escogencia implicó la aceleración del desarrollo de *D-Explorer*. En vez de terminar el desarrollo en setiembre del 2008, como estaba estimado en la propuesta de la tesis, se finalizó a mediados de julio.

En el curso *SD* hay 9 estudiantes y en el curso *PJAD* hay 8 estudiantes. El número total de estudiantes es bajo. Este hecho podría afectar negativamente el proceso de análisis de las encuestas pues el peso de la opinión de un estudiante es alto. En el curso *SD* la opinión de un estudiante tiene un peso de 1/9 y en la población total un estudiante tiene un peso de 1/17. En estos casos, una respuesta muy negativa o muy positiva del estudiante podría sesgar el resultado global hacia uno u otro lado. Esta es una limitación del experimento pues para aumentar el número de estudiantes se debería incluir más cursos o extender el experimento a más semestres.

7.4 Diseño de las tareas programadas

Después de seleccionar los cursos, se procedió a diseñar las tareas programadas de los cursos. Esta actividad se realizó con el profesor del curso correspondiente antes de que empezara el curso. Los estudiantes deben usar *D-Explorer* para implementar estas tareas y no estaba permitido usar otras herramientas como *sockets* o *RMI*.

En el curso *SD* se diseñaron tres tareas con diferentes grados de dificultad y se asignaron de primero las tareas más fáciles. En particular, la primera tarea programada tenía como objetivo que el estudiante se familiarizara con *D-Explorer*. Asignar las tareas en orden creciente de dificultad tiene un efecto pedagógico positivo: el estudiante adquiere experiencia en cada tarea y usa esta experiencia acumulada para enfrentar problemas cada vez más difíciles. Las tareas diseñadas son: implementar un servicio sencillo, implementar

el esquema de caché con invalidación y programar un esquema de caché usando uno de los protocolo de *Lifetime Based Consistency*[35]. Las dos últimas tareas son conceptualmente similares. En ambos casos se programan cachés pero la diferencia está en el protocolo usado para mantener el caché. El protocolo de la tercera tarea es más complejo. Además, en la tercera tarea se introducen otros elementos para elevar su grado de dificultad.

En el curso de *PJAD* la escogencia de la tarea fue dada por las características del curso. Como parte del curso, los estudiantes deben desarrollar una versión no distribuida del juego cuatro en línea y luego implementar diferentes versiones distribuidas de este juego usando *sockets*, *RMI* y otras herramientas que se estudian en este curso. Dadas estas circunstancias, la tarea diseñada para este curso fue convertir la versión no distribuida del juego cuatro en línea a una versión distribuida usando *D-Explorer*. Esta tarea tiene la ventaja de que permite comparar *D-Explorer* contra *sockets* y *RMI* pues el estudiante debe realizar la misma tarea usando estas otras herramientas.

En el proceso de diseño de tareas programadas, el tipo de problema distribuido influenció poco en la escogencia de las tareas. Si se consideró el grado de dificultad del problema distribuido y, con excepción de la primera tarea del curso *SD*, se trató de que las tareas tuvieran un nivel de dificultad razonable a criterio del profesor.

A continuación se describen las tareas diseñadas. Para cada tarea se presenta una explicación del problema y una posible solución de la tarea. Las soluciones se enfocan en resolver el problema distribuido y se presentan en términos de procedimientos remotos (RPCs). Los procedimientos remotos pueden estar alojados en el servidor o en los clientes. Los procedimientos remotos del servidor son invocados por los clientes. Los procedimientos remotos del cliente son invocados por el servidor.

7.4.1 Implementar un servicio sencillo

La primera tarea del curso de *SD* (SD-1) consiste en implementar un servicio sencillo. El estudiante debe usar *D-Explorer* para implementar un programa servidor y un programa cliente. El programa servidor recibe un mensaje de solicitud del cliente, procesa la solicitud y devuelve un mensaje con la respuesta al cliente. El programa cliente envía una solicitud al

servidor y espera la respuesta. El tipo de procesamiento que realiza el servidor a las solicitudes queda a criterio del estudiante pues lo más importante en esta tarea es que el estudiante logre intercambiar mensajes entre el cliente y el servidor. Algunos ejemplos de servicios sencillos son: sumar dos números, contar las letras de un mensaje, etc.

Esta tarea tiene como objetivo introducir al estudiante en el uso de *D-Explorer* y permite obtener una evaluación inicial sobre la capacidad y la facilidad de uso de las herramientas.

La solución de esta tarea requiere un procedimiento remoto alojado en el servidor. Para el caso de un servicio que suma dos número, el procedimiento sería:

public int sumar(int a, int b)

7.4.2 Caché con invalidación

La segunda tarea del curso de *SD* (SD-2) consiste en implementar el esquema de caché con invalidación usando *D-Explorer*. El estudiante debe implementar un servidor de datos y dos clientes. El servidor mantiene una lista de datos identificados por una letra (A, B, C, ...) y cada dato almacena un valor numérico. Los programas clientes pueden leer y grabar datos del servidor de datos pero mantienen un caché local. Cuando un cliente necesita leer un dato, si el dato no está en su caché, se debe leer este dato del servidor y actualizar su caché. Cuando un cliente graba un dato, el cliente debe actualizar su caché y el servidor de datos. Si este dato está en el caché de algún otro cliente, el servidor debe enviar un mensaje a este cliente para que invalide (elimine) este dato de su caché. Por ejemplo, si el cliente 1 lee el dato A y posteriormente el cliente 2 graba el dato A entonces el servidor debería enviar un mensaje al cliente 1 para que invalide (elimine) el dato A de su caché.

La solución de esta tarea requiere tres procedimientos remotos en el servidor y uno en el cliente. Los procedimientos remotos del servidor son:

public int registrar()

public int leer(int cliente, char variable)

public void grabar(int cliente, char variable, int valor)

El procedimiento remoto del cliente es:

public void invalidar(char variable)

Con excepción del método *registrar*, los otros métodos se pueden derivar directamente de la explicación del problema. El método *registrar* es invocado por el programa cliente al inicio de su ejecución. Este método devuelve un número entero consecutivo con cada invocación. El primer cliente que invoca este método recibe *1*. El segundo cliente que lo invoca recibe *2*. El uso de número entero único para identificar a cada cliente facilita la lógica general de la aplicación.

7.4.3 Caché con protocolos de *Lifetime Based Consistency*

La tercera tarea del curso de *SD* (SD-3) consiste en implementar la variante del protocolo de *lifetime-based-consistency* [35] que usa relojes lógicos vectoriales[36]. El estudiante debe usar *D-Explorer* para implementar un servidor y cuatro clientes.

Tabla 7.1. Orden de ejecución del sistema. La **S** denota al servidor y **Ci** denota a cada uno de los clientes. Los clientes no son interactivos sino que cada cliente pide una lista de operaciones de lectura y escritura al servidor, ejecuta esta lista de operaciones secuencialmente y le envía el historial local de la ejecución al servidor. Cuando el servidor recibe los historiales de todos los clientes, debe verificar que los historiales recibidos cumplen con la condición de consistencia del protocolo.

[S] Generar lista de operaciones de cada cliente (lecturas y escrituras de datos).

[Ci] Registrarse con el servidor y recibir como respuesta el identificador del cliente (1, 2, 3 o 4) y la lista de operaciones que debe ejecutar.

[S] Cuando todos los clientes se han registrado, enviar un mensaje a cada cliente para que ejecute su lista de operaciones.

[Ci] Ejecuta su lista de operaciones. En este paso, cada cliente realiza operaciones de lectura y de escritura sobre los datos. Estas operaciones pueden completarse localmente accesando únicamente el caché del cliente o pueden involucrar al servidor y otros clientes. Cuando termina de ejecutar la lista de operaciones, el cliente envía su historial local al servidor.

[S] Cuando recibe los historiales locales de todos los clientes, verifica que estos historiales cumplan con la condición de consistencia.

En la tabla 7.1 se describe la interacción entre el servidor y los clientes.

La solución de esta tarea requiere cuatro procedimientos remotos en el servidor y uno en el cliente. Los procedimientos remotos del servidor son:

public Resultado registrar(): este método retorna el objeto *Resultado*, el cual contiene el identificador del cliente (1, 2, 3 o 4) y la lista de operaciones que debe ejecutar.

public int leer(int cliente, char variable)

public void grabar(int cliente, char variable, int valor)

public void finalizar(Historial historial)

El procedimiento remoto del cliente es:

public void arrancarEjecucion()

public ResultadoBusqueda buscarDato(char v, LifeTime lt): busca un dato con ciertas características. Si en el caché del cliente existe el dato v con un *lifetime* [35] mayor que el parámetro *lt*, se retorna este dato. En caso contrario, se retorna **nulo**.

A partir de estos procedimientos remotos se puede implementar la lógica descrita en la tabla 7.1.

Cuando un programa cliente arranca, este invoca el método *registrar* para obtener su identificador y la lista de operaciones que debe ejecutar.

En el método *registrar* del servidor, cuando se registra el último cliente este método invoca el método *arrancarEjecucion* para cada uno de los clientes registrados.

Dentro del método *arrancarEjecucion* cada cliente ejecuta su lista de operaciones invocando los métodos *leer* y *grabar* del servidor. En este proceso el cliente también va actualizando su historial local. Al finalizar este proceso, el cliente invoca el método *finalizar* para enviarle el historial de su ejecución al servidor.

En el método *finalizar* del servidor, cuando se recibe el historial del último cliente se verifica la consistencia de los mismos.

7.4.4 Cuatro en línea

En la tarea del curso *PJAD* (PJAD-1), los estudiantes deben programar el juego *cuatro en línea* y luego convertir este programa a una aplicación distribuida usando *sockets*, *RMI* y *D-Explorer*. En la versión inicial no distribuida, un único programa interactúa con los dos jugadores en una misma computadora. Cada jugador se turna para usar el teclado y digitar su jugada. El programa contiene la lógica del juego, presenta el estado del juego en la pantalla y recibe jugadas del teclado. En las versiones distribuidas usando *sockets*, *RMI* y *D-Explorer*, el estudiante debe implementar el servidor del juego y dos programas clientes. El servidor contiene la lógica del juego: recibe mensajes con las jugadas de cada jugador, actualiza el estado del juego y retorna mensajes con el estado del juego. El cliente despliega el estado del juego, solicita la jugada al jugador, lo envía al servidor y espera su respuesta.

Esta tarea tiene como objetivo comparar *D-Explorer* con *sockets* y *RMI*: los estudiantes deben convertir la versión no distribuida a una versión distribuida usando cada una de estas tres herramientas.

La solución de esta tarea requiere un procedimiento remoto en el servidor y tres en el cliente. El procedimiento remoto del servidor es:

public registrar(String nombre)

Los procedimientos remotos del cliente son:

public String preguntar(String pregunta): despliega la pregunta en la pantalla del programa cliente y retorna la respuesta digitada por el usuario al programa servidor.

public void imprimir(String mensaje): despliega un mensaje en la pantalla del programa cliente.

public void terminarJuego(): le indica al programa cliente para que termine su ejecución.

La solución asume que se programó una versión no distribuida del juego cuatro en linea en donde el programa le pide el nombre a los dos jugadores y luego entra en el ciclo del juego. En este ciclo cada jugador se alterna en realizar una jugada hasta que alguno de los dos

gane el juego y el programa despliega el estado del juego para que cada jugador realice su jugada. A partir de esta solución, para implementar la versión distribuida se realizan los siguientes cambios:

El programa cliente pide el nombre al jugador e invoca el método *registrar* para pasarle el nombre del jugador al servidor.

En el método *registrar* del servidor, cuando se registra el segundo jugador este método ejecuta el ciclo de juego pero en vez de desplegar mensajes en la pantalla del programa servidor, usa el método *informar* para desplegar mensajes en la consola del programa cliente y el método *preguntar* para obtener las jugadas a través del programa cliente.

Después de describir las tareas programadas de ambos cursos, en la siguiente sección se compara el nivel de dificultad de las mismas.

7.4.5 Comparación de las tareas

Se clasifican las tareas usando una escala de niveles de dificultad con tres categorías: fácil, intermedio, difícil. Se aplicaron los siguientes criterios en la clasificación:

- Tipo de comunicación. Se identifican dos tipos de comunicación: *secuencial y sincrónico*, y *concurrente y asincrónico*. La comunicación *secuencial y sincrónica* es más fácil de implementar que la comunicación *concurrente y asincrónica*.
- Número de procedimientos remotos de las soluciones. Se considera que la lógica de procesamiento de una tarea es sencilla si su solución requiere no más de dos procedimientos remotos, es intermedia si requiere tres o cuatro procedimientos remotos y compleja si requiere más de cuatro procedimientos remotos.

La tabla 7.2 presenta la clasificación de las tareas programadas usando los criterios anteriores.

Tabla 7.2. Dificultad por tarea.

Tarea	Dificultad	Comunicación	# Procedimientos	Lógica Procesamiento
SD-1	Fácil	Secuencial y sincrónico	2	Sencillo
SD-2, PJAD-1	Intermedio	Secuencial y sincrónico	4	Intermedio
SD-3	Difícil	Concurrente y asincrónico	6	Complejo

La tarea SD-1 consiste de un cliente y un servidor. La comunicación entre estos dos programas es secuencial y sincrónica. El servicio contiene dos procedimientos remoto. La lógica de estos procedimientos consiste en sumar dos números, contar las letras de una hilera, etc.

Las tareas SD-2 y PJAD-1 consisten de un servidor y dos clientes. La comunicación entre los clientes y el servidor es secuencial y sincrónica pues el servidor interactúa con un cliente a la vez. En las dos tareas se implementan un total de cuatro procedimientos remotos y estos están alojados tanto en el cliente como en el servidor. La lógica de estos procedimientos son más complejos que la lógica de los procedimientos de la tarea SD-1: el método *imprimir* despliega un mensaje de texto en la pantalla, el método *leer* busca un dato en una lista y devuelve el valor del dato, etc. Aunque ambas tareas están en la misma categoría, la tarea de cuatro en línea distribuido es un poco más compleja que la tarea de caché con invalidación. En la aplicación cuatro en línea los programas clientes deben alternarse para realizar su jugada. Por el contrario, en la aplicación caché con invalidación, el cliente puede leer o grabar en cualquier momento.

La tarea SD-3 consiste de un servidor y cuatro cliente. La comunicación entre los clientes y el servidor es concurrente y asincrónico: los clientes realizan solicitudes al servidor al mismo tiempo y el servidor puede enviar solicitudes al cliente asincrónicamente. La solución de esta tarea necesita seis procedimientos remotos. La lógica de los métodos *arrancarEjecucion* y *finalizar* son bastante más complejas que la lógica de los procedimientos remotos de las otras tres tareas. La tarea también presenta un alto nivel de concurrencia pues el método *arrancarEjecucion* es ejecutado concurrentemente por los

cuatro clientes y en este método se realiza un conjunto de lecturas y escrituras de datos.

Con esta clasificación concluye la sección de diseño de las tareas programadas. En la siguiente sección se explican las características principales de la encuesta.

7.5 Diseño de la encuesta.

En esta sección se describe la encuesta usada para recolectar información de los estudiantes después de finalizar cada tarea programada y las reglas para tabular las encuestas.

Las encuestas son anónimas. El encuestado no necesita escribir su nombre en la encuesta. Se considera que el anonimato reduce la presión en el encuestado, sobre todo cuando desea responder negativamente o en el caso de preguntas relacionadas con el encuestado.

La mayoría de las preguntas de la encuesta se responde con un número entre 0 y 10. El número 0 representa una respuesta negativa (no está de acuerdo, influye muy poco, conoce muy poco). El número 10 representa una respuesta positiva (está totalmente de acuerdo, influye mucho, conoce mucho). Cuando una pregunta no aplica, se puede responder con **N/A**. Por ejemplo, si una pregunta consiste en comparar *D-Explorer* con *RMI* pero el encuestado no conoce *RMI*, se responde con **N/A**.

La encuesta está organizada en cuatro secciones. A los estudiantes se les aplican dos tipos de encuestas: la encuesta completa que incluyen las cuatro secciones y la encuesta parcial que solamente contiene la sección cuatro con las preguntas sobre la dificultad de la tarea, las horas invertidas en programar la tarea, etc. En el curso *SD* se aplican una encuesta completa después de la primera tarea y dos encuestas parciales para las subsiguientes tareas. En el curso *PJAD* se aplica una encuesta completa. En total se recolectan cuatro grupos de encuestas.

En la primera sección se obtiene la experiencia del estudiante en programación de redes:

1.a) Indique su nivel de conocimientos sobre el protocolo TCP/IP (0..10). Este protocolo es bastante común en las redes actuales.

1.b) Indique su nivel de experiencia programando *sockets* (0..10). El *socket* es una herramienta muy usada en la programación de aplicaciones distribuidas.

1.c) Indique su nivel de experiencia programando hilos (0..10). La concurrencia puede ser un elemento importante en muchas aplicaciones distribuidas.

1.d) Indique su nivel de experiencia en sincronizar hilos (implementar una región crítica, un semáforo, etc) (0..10). La coordinación de la sincronización también es importante en la programación distribuida.

En la segunda sección se evalúa la calidad del material de apoyo de *D-Explorer*:

2.a) El material de la distribución es suficiente para que un estudiante aprenda el uso de las herramientas (0..10). Se evalúa la completitud del material de apoyo.

2.b) La organización del material es adecuada para el aprendizaje de las herramientas (0..10). Se evalúa la organización del material de apoyo.

2.c) El proyecto de ejemplo *Tutorial 01* le ayudó a entender la implementación de un servicio usando *D-Explorer*. El proyecto escogido es adecuado y la explicación de las clases del proyecto es clara (0..10). Se evalúa la utilidad del *tutorial 01*.

2.d) El proyecto de ejemplo *Tutorial 02* le ayudó a entender el uso de las notificaciones. El proyecto escogido es adecuado y la explicación de las clases del proyecto es clara (0..10). Se evalúa la utilidad del *tutorial 02*.

La tercera sección evalúa el uso de *D-Explorer* en el desarrollo de aplicaciones distribuidas:

3.a) Pude programar una aplicación distribuida que implementa un servicio sencillo (0..10). Esta pregunta está ligada a la pregunta 2.c sobre el *tutorial 01* y evalúa si el estudiante pudo aplicar los conocimientos adquiridos en el *tutorial 01*.

3.b) Pude programar una aplicación distribuida que usa notificaciones (0..10). Esta pregunta está ligada a la pregunta 2.d sobre el *tutorial 02* y evalúa si el estudiante pudo aplicar los conocimientos adquiridos en el *tutorial 02*.

3.c) Pude ejecutar los programas que desarrollé en una misma computadora y no tuve problemas (0..10). Se evalúa la facilidad de ejecutar una aplicación desarrollada con *D-Explorer* en una computadora: si hay que instalar otras herramientas, configurar la computadora, etc.

3.d) Pude ejecutar los programas que desarrollé en varias computadoras de una LAN y no tuve problemas (0..10). Se evalúa la facilidad de ejecutar una aplicación desarrollada con *D-Explorer* en una red: si hay que instalar otras herramientas, configurar la red, etc.

3.e) Tengo experiencia programando *sockets*. Considero que muchas de las cosas que he programado usando *sockets*, también se pueden hacer con *D-Explorer* (0..10). Se realiza una evaluación subjetiva de si la funcionalidad de *D-Explorer* es comparable a la funcionalidad de *sockets*.

3.f) Siguiendo con la pregunta anterior, también considero que muchas de las cosas que he programado usando *sockets*, se podrían programar más fácilmente usando *D-Explorer* (0..10). Se realiza una evaluación subjetiva de si *D-Explorer* es más fácil de usar que *sockets*.

En la cuarta sección se obtiene información relacionadas con cada tarea programada.

4.a) Indique el nivel de dificultad de la tarea. El cero significa que es muy fácil. El diez indica que la tarea tiene un alto grado de dificultad (0..10). El nivel de dificultad se usa para comparar la dificultad relativa de las cuatro tareas.

4.b) Indique las horas dedicadas para programar la tarea. Indique únicamente el tiempo de programación. Excluya el tiempo usado para estudiar los tutoriales y la documentación de la distribución. Excluya también el tiempo dedicado a análisis y diseño del problema. La duración estimada en horas se usa para comparar la duración de las tareas.

4.c) Si ha realizado alguna tarea programada similar usando *sockets*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta. En las encuestas de la tarea PJAD-1, las respuestas de las preguntas 4.b, 4.c y 4.d se usan para realizar la comparación entre *D-Explorer*, *sockets* y *RMI*.

4.d) Si ha realizado alguna tarea programada similar usando *RMI*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta.

En las encuestas de la tarea PJAD-1, las respuestas de las preguntas 4.b, 4.c y 4.d se usan para realizar la comparación entre *D-Explorer*, *sockets* y *RMI*.

Después de las cuatro secciones anteriores se incluye una pregunta abierta para opinar sobre *D-Explorer*, reportar problemas, sugerir mejoras, emitir críticas, etc.

Con el diseño de la encuesta concluye la preparación de la validación de la utilidad de *D-Explorer*. En el siguiente capítulo se describe la ejecución de la validación de la utilidad.

8 Ejecución de la validación de la utilidad

Después de concluir con los preparativos de la validación de la utilidad, en donde se seleccionaron los cursos y se diseñaron las tareas programadas y las encuestas, se procede a explicar la etapa de ejecución de la validación de la utilidad de *D-Explorer*. Primero se presenta la metodología usada en esta etapa y luego se presentan los resultados obtenidos.

8.1 Metodología

La metodología aplicada en la etapa de ejecución consta de los siguientes pasos:

1. Implementar tareas programadas usando *D-Explorer*.
2. Analizar las encuestas.
3. Analizar el código fuente.

Se procede a explicar cada uno de estos pasos.

8.1.1 Metodología de implementación de las tareas programadas

Esta actividad se realiza en el II semestre del 2008. Durante este periodo, los estudiantes de los cursos *Sistemas Distribuidos* (*SD*) y *Programación Java para Ambientes Distribuidos* (*PJAD*) usan *D-Explorer* para resolver las tareas programadas diseñadas. Se imparte una charla de *D-Explorer* cuando se asigna la primera tarea a los estudiantes y se aplica una encuesta a los estudiantes cuando estos entregan la tarea. Después de la primera tarea programada del curso se aplica una encuesta completa. A partir de la segunda tarea del curso, se aplica una encuesta parcial que solo contiene la sección cuatro de la encuesta completa, con preguntas sobre la dificultad, la duración de la tarea programada, etc.

8.1.2 Metodología de análisis de encuestas

En el análisis de encuestas, se recolectan, se tabulan y se procesan las encuestas y se realiza un análisis diferente para cada curso.

En el curso *SD* se recolectan una encuesta completa después de la primera tarea y dos encuestas parciales para las subsiguientes tareas. En el curso *PJAD* se recolecta una

encuesta completa.

En la tabulación de la encuesta se aplican las siguientes reglas con el fin de depurar la información, y con excepción de estas reglas, en todos los demás casos se tabula la respuesta del encuestado:

- Una respuesta en blanco se tabula como **N/A** y no se usa en el procesamiento de la encuesta.
- En las preguntas 4.c y 4.d, las respuestas en cero se tabulan como **N/A**. En estas preguntas, el estudiante debe indicar el número de horas usadas para completar la tarea programada usando *sockets* o *RMI* y puede responder con **N/A** si nunca ha realizado esta actividad. En estas preguntas, la respuesta no puede ser cero pues indicaría que completó la tarea programada en cero horas.

En el procesamiento de las encuestas, se agrupan las encuestas por tarea programada (SD-1, SD-2, SD-3, PJAD-1) y se calcula el promedio de cada pregunta para cada grupo de encuestas. En el cálculo del promedio no se consideran las respuestas **N/A**.

En el curso *SD* se analizan las encuestas de las tareas SD-1, SD-2 y SD-3. Se analizan los promedios de las preguntas 4.a (dificultad estimada de la tarea en una escala de 0 a 10) y 4.b (tiempo estimado de duración de la tarea) para determinar si los estudiantes tuvieron tiempo suficiente para completar las tareas en el plazo dado.

En el curso *PJAD* se analizan los promedios de las preguntas 4.b (duración estimada en horas de la tarea con *D-Explorer*), 4.c (duración estimada en horas de la tarea con *sockets*), 4.d (duración estimada en horas de la tarea con *RMI*) para determinar cual es el menor de estos promedios y si el plazo para completar la tarea fue adecuado.

8.1.3 Metodología de análisis de código

Se analiza el código fuente de algunas de las tareas programadas implementadas en los cursos *SD* y *PJAD*. En la siguiente sección se explica el análisis de código del curso *SD*.

8.1.3.1 Metodología de análisis de *Sistemas Distribuidos*

Se analiza la tercera tarea del curso *SD* por ser esta la más compleja. En esta tarea hay un servidor y cuatro clientes. Los clientes realizan operaciones de lectura y de escritura contra el servidor. El servidor recibe solicitudes concurrentemente de los cuatro clientes y, dependiendo de la operación, puede enviar mensajes a los clientes para buscar un dato. Estos mensajes deben ser recibidos y procesados por el cliente mientras este todavía está ejecutando el hilo en donde realiza las lecturas y escrituras al servidor. La solución de esta tarea requiere que tanto el servidor como el cliente tenga la capacidad de recibir y procesar mensajes asincrónicamente y concurrentemente, y coordinar correctamente el envío y la recepción de mensajes para evitar que estos se traslapen.

El problema de traslape de mensajes ocurre cuando varios procesos envían mensajes concurrentemente y no sincronizan correctamente estos envíos. Por ejemplo, si se ejecutan al mismo tiempo varios hilos que usan un mismo *socket* para enviar los atributos *a* y *b*, el proceso al otro extremo de la conexión podría recibir la secuencia *a*, *a*, *b*, *b* si no se sincroniza correctamente el envío de los atributos entre los hilos.

En el análisis de código se realiza un análisis cuantitativo y un análisis de uso de servicios de *D-Explorer*. En las siguientes secciones se explican estos dos análisis.

A.- Análisis cuantitativo de código

En el análisis cuantitativo se calcula el *porcentaje de código de comunicación* (PCC), el cual se define como la razón entre el número *total de líneas de código de comunicación de D-Explorer* (TD) entre el número *total de líneas de código de métodos* (TM).

El indicador TM se obtiene analizando el código fuente con la aplicación *Metrics*[37]. Esta aplicación cuenta las líneas de código que forman parte de un método pero no considera las líneas en blanco o comentarios.

Se revisan los proyectos para estimar el número de líneas de código de comunicación de *D-Explorer*. Se cuentan las líneas de código relacionados con *D-Explorer* y se clasifican estas líneas en código ejecutable que está dentro de un método y código declarativo como encabezados de métodos, declaraciones de variables, etc. El código ejecutable, *a su vez, fue*

clasificado en las siguientes categorías:

- *Serialización:* incluye invocaciones de los métodos de la interface *IFlujoEntrada (getInt, getString, etc)* e *IFlujoSalida (putInt, putString)*. Estos métodos se invocan dentro de los métodos *grabarAtributosAlFlujo* y *leerAtributosDelFlujo* de la clase *Mensaje* o de subclases de esta clase.
- *Conector Servicio*: incluye invocaciones al método *conectar* de la interface *ICanal* y las operaciones *enviar*, *recibir* y *cerrar* de la clase *ConectorServicio*.
- *Notificaciones*: incluye la creación de instancias de la clase *ModuloNotificacion* y llamados a los métodos de esta clase, en especial los métodos *notificar* y *getDestinoNotificacion*.
- *Impresiones*: incluye invocaciones del método *imprimir* de la clase *Consola* y otros métodos para imprimir excepciones o agregar comentarios a las bitácoras.
- *Otros*: incluye invocaciones a otros métodos de *D-Explorer* no considerados en los puntos anteriores. Por ejemplo, métodos para arrancar el canal, activar el servidor de descubrimiento, registrar un servicio en el canal, etc.

Para obtener el número total de líneas de código de comunicación de *D-Explorer* (TD) se suman las líneas de las categorías de *Serialización*, *Conector Servicio*, *Notificaciones* y *Otros*. No se suman las líneas de *Impresiones* pues este no es código de comunicación.

El *porcentaje de código de comunicación* (PCC) se obtiene dividiendo TD entre TM. Este porcentaje es un indicador de la facilidad de uso de *D-Explorer* y se puede interpretar como el esfuerzo realizado para implementar la lógica de comunicación de la aplicación. Entre más bajo sea este porcentaje, menor es el esfuerzo realizado y mayor es el beneficio de usar *D-Explorer*.

B.- Análisis de uso de servicios.

En el análisis de servicios se revisa el código fuente de un proyecto para verificar que los estudiantes usan *D-Explorer* y no otras herramientas como *sockets* o *RMI*, y para determinar los servicios usados.

Con el análisis de servicios concluye la sección de análisis de código del curso *SD*. En la siguiente sección se explica el análisis de código del curso *PJAD*.

8.1.3.2 Metodología de análisis de *Programación Java para Ambientes Distribuidos*

En el análisis de código del curso *PJAD*, se obtienen tareas del juego cuatro en línea implementadas con *D-Explorer*, *sockets* y *RMI*, se usa el software *metrics*[37] para calcular el total de líneas de código de método (TM) de cada tarea, se agrupan las tareas por tipo de herramienta (*D-Explorer*, *sockets*, *RMI*), se calcula el promedio del TM para cada grupo de tareas y se comparan estos promedios. El promedio del TM puede usarse como un indicador de la facilidad de uso de la herramienta pues se espera que entre menor sea este promedio, mayor es la facilidad de uso de la herramienta.

Después de explicar los pasos de la metodología utilizada en la etapa de ejecución de la validación de utilidad de *D-Explorer*, seguidamente se describen los resultados obtenidos en cada uno de estos pasos, empezando con la implementación de las tareas programadas.

8.2 Resultados de la implementación de las tareas programadas

Durante el II ciclo del 2008, se dio la charla introductoria de *D-Explorer*, se asignaron las tareas programadas y se aplicaron las encuestas.

La charla tuvo una duración aproximada de 45 minutos y en ella se explicó el material de apoyo de *D-Explorer* y el *tutorial 01* de este material, en donde se implementa un servicio sencillo. En ambos cursos se dio la charla en el mismo día en que se asignó la primera tarea programada: primero se dio la charla y luego se asignó la tarea programada.

La primera tarea del curso *SD* fue asignada el 11 de agosto. Ese día se dio la charla y luego se asignó la tarea. Los estudiantes entregaron la tarea dos semanas después, el 25 de agosto, y ese mismo día se aplicó la encuesta completa. Un estudiante no pudo entregar la tarea porque tuvo problemas con los laboratorios de la UCR. Estos equipos están configurados con un *firewall* muy restrictivo y tienen bloqueados la mayoría de los puertos.

La primera tarea del curso *PJAD* fue asignada el 17 de setiembre. Ese día se dio la charla y luego se asignó la tarea. La tarea debía entregarse dos semanas después, el 1 de octubre. El

día de entrega, solo tres estudiantes concluyeron la tarea y el profesor extendió el plazo de entrega. Dadas estas circunstancias, ese mismo día se dio una segunda charla de *D-Explorer*. La encuesta no fue aplicada el día de entrega de la tarea sino que se aplicó el 5 de noviembre. Las razones en el retraso de la aplicación de la tarea fueron las siguientes: se amplió la fecha de entrega de la tarea, se tuvo que esperar a que los estudiantes terminaran la programación del juego cuatro en línea usando *sockets* y RMI, y por motivos personales.

El 3 de noviembre se asignó la segunda tarea del curso *SD*. Dos semanas después, el 17 de noviembre, los estudiantes entregaron la tarea y llenaron una encuesta parcial.

El 24 de noviembre se asignó la tercera tarea del curso *SD*. Dos semanas después, el 8 de diciembre, la mayoría de los estudiantes entregaron la tarea y llenaron una encuesta parcial. Un estudiante se atrasó 4 días y entregó la tarea el 12 de diciembre. Tres estudiantes no llenaron la encuesta: uno estaba fuera del país y otro entregó la tarea por correo.

En la tabla 8.1 se presenta una lista con las principales actividades en la implementación de las tareas y el número de estudiantes involucrados en cada actividad.

Tabla 8.1. Resumen de estudiantes por actividad. La primera columna describe el tipo de actividad. La segunda columna presenta el número de estudiantes del curso *SD* que realizaron cada actividad. La tercer columna muestra el número de estudiantes del curso *PJAD* que realizaron cada actividad.

Actividad	**Curso *SD***	**Curso *PJAD***
Empezaron el curso	9	8
Se retiraron del curso	1	0
Entregaron tarea SD-1 a tiempo	8	0
Entregaron tarea SD-2 a tiempo	8	0
Entregaron tarea SD-3 a tiempo	7	0
Se atrasaron con tarea SD-3	1	0
Entregaron tarea PJAD con *D-Explorer* a tiempo	0	3
Entregaron tarea PJAD con *sockets* a tiempo	0	5
Entregaron tarea PJAD con *RMI* a tiempo	0	3
Se atrasaron en tarea PJAD con *D-Explorer*	0	4
No entregaron la tarea PJAD con *D-Explorer*	0	1

Se presentan los resultados más importantes en la implementación de las tareas.

Los profesores de ambos cursos indican que las tareas funcionaron correctamente. En el curso *SD* participé en la evaluación de las tareas y pude constatar que la lógica de comunicación funcionó de acuerdo a lo solicitado. Con respecto a la correctitud de la lógica de procesamiento de las tareas del curso *SD*, se confió en el criterio emitido por el profesor aunque en la primera y segunda tarea del curso *SD* considero que la lógica de procesamiento está correcta.

Los estudiantes, en todo el semestre, solo reportaron un defecto de *D-Explorer*. En la primera tarea de *SD*, los estudiantes detectaron que la tarea desarrollada con *D-Explorer* incurría en un alto consumo de CPU aunque este problema no impidió que completaran la tarea. Este hecho indica el alto nivel de depuración de *D-Explorer* y confirma, de manera independiente, los resultados del proceso de validación funcional del capítulo 6.

Hay indicios de que las herramientas son fáciles de usar: los estudiantes hicieron pocas consultas, varios estudiantes usaron componentes de *D-Explorer* que no fueron explicados en la charla y hubo pocos atrasos en la entrega de las tareas.

Los estudiantes de *SD* no hicieron consultas y solo tres estudiantes de *PJAD* enviaron correos electrónicos con preguntas sobre *D-Explorer* y la tarea.

Varios estudiantes de *SD* usaron el servicio de notificación para implementar la segunda y la tercera tarea. Este servicio no fue explicado en la charla aunque si se explica su uso en el *tutorial 02* del material de apoyo. Este hecho indica que el servicio de notificación es fácil de usar y que el *tutorial 02* es fácil de entender pues estos estudiantes aprendieron a usar este servicio a partir del *tutorial 02* y usaron este conocimiento para implementar una tarea de nivel intermedio y una tarea compleja.

El hecho de que la mayoría de las tareas se entregaron a tiempo es otro indicio de la facilidad de uso de *D-Explorer*. Este hecho contrasta con experiencias anteriores en este mismo curso cuando el profesor Francisco Torres asignó tareas programadas que requerían el uso de *sockets:* los estudiantes dedicaban mucho tiempo en implementar la lógica de comunicación con *sockets* y poco tiempo en la lógica no distribuida, y al final pocos

estudiantes completaban la tarea programada[38]. Debido a estas experiencias, el profesor dejó de asignar este tipo de tareas y en cursos recientes de *SD* se asignaban tareas que no usaban herramientas de comunicación. En este tipo de tareas se modelaban los procesos distribuidos con estructuras de datos como listas enlazadas o vectores, y el intercambio de mensajes se simulaba incrementando un contador o un reloj lógico. El profesor Ronald Argüello, quien impartió este curso en el pasado, reporta experiencias similares con tareas programadas que requerían el uso de *sockets*[39]

Finalmente, la retroalimentación recibida de los estudiantes también es un resultado positivo. Cabe destacar la sugerencia de algunos estudiantes de *PJAD* para elaborar un tutorial que explique la implementación de un servicio en un nodo cliente y el uso de este servicio desde un nodo servidor. Este esquema de comunicación se podría usar para resolver la tarea del curso *PJAD* y la segunda tarea del curso *SD*, aunque en este último caso varios estudiantes optaron por usar el servicio de notificaciones.

Después de asignar las tareas programadas y aplicar las encuestas de los estudiante, se procede a tabular y analizar las encuestas.

8.3 Resultados del análisis de encuesta

Se presentan los resultados del análisis de encuestas, primero se presentan los resultados del curso *SD* y luego se presentan los resultados del curso *PJAD*.

8.3.1 Análisis de *Sistemas Distribuidos*

Se comparó la duración y la dificultad promedio de las tres tareas del curso *SD*.

Tabla 8.2. Promedio y desviación de dificultad y duración por tarea. La primera columna contiene la pregunta. Las siguientes seis columnas contienen el promedio y la desviación estándar de la pregunta para la tarea SD-1, SD-2 y SD-3 respectivamente.

	SD-1		SD-2		SD-3	
Pregunta	P	DS	P	DS	P	DS
Dificultad estimada por estudiante (1..10)	3.44	2.06	4.75	1.64	7.33	1.25
Duración estimada por estudiante (horas)	3.00	2.11	9.50	7.91	24.17	8.73

Los datos de la tabla 8.2 indican que la tarea SD-1 es la más sencilla, la tarea SD-2 presenta un nivel de dificultad intermedio y la tarea SD-3 es la más difícil y confirman los resultados

de la comparación de dificultad de las tareas (ver 7.4.5). La duración promedio de las dos primeras tareas deja mucha holgura para completar las tareas en dos semanas. La duración promedio de la tercera tarea deja poca holgura pero es factible, posiblemente tres semanas como tiempo de entrega hubiera sido un tiempo más justo.

8.3.2 Análisis de *Programación Java en Ambientes Distribuidos*

Se compararon las duraciones de las implementaciones del juego cuatro en línea usando las herramientas *D-Explorer*, *socket* y *RMI*. La comparación se muestra en la tabla 8.3.

Tabla 8.3. Promedio y desviación de la duración de la tarea por tipo de herramienta. La primera columna contiene la pregunta. Las siguientes dos columnas contienen el promedio y la desviación estándar de la duración en horas. En las fila se presentan la duración promedio en horas para cada herramienta.

Pregunta	Promedio	Desviación
Duración estimada en horas: D-Explorer	15.75	4.32
Duración estimada en horas: socket	11.25	3.73
Duración estimada en horas: RMI	9.75	3.53

La duración promedio de 15.75 horas en la tarea usando *D-Explorer* indica que la tarea se pudo completar en dos semanas y no justifica los atrasos en la entrega de la tarea.

Los resultados de la tabla 8.3 indican que los estudiantes duraron más tiempo en completar la tarea usando *D-Explorer*. Sin embargo, se debe indicar que esta comparación tiene varios sesgos en contra de *D-Explorer*.

El primer sesgo es que los estudiantes realizaron las tareas en este orden: *D-Explorer*, *sockets* y *RMI*. Resolver un problema nuevo por primera vez es más difícil que resolver un problema similar por tercera vez, y parte de los problemas que se resuelven en la primera tarea son usados en las siguientes tareas, por ejemplo, el diseño de los mensajes, el orden de los intercambios de mensajes, la organización de las clases.

El segundo sesgo es que los estudiantes podrían tener conocimientos previos de *sockets* o de *RMI*, accesar Internet para buscar ejemplos de código o consultar con conocidos que hayan usado estas dos herramientas. Por el contrario, *D-Explorer* es una herramienta nueva que pocas personas conocen y no hay ejemplos en Internet sobre su uso.

Un experimento más equitativo para comparar estas tres herramientas sería dividir el grupo

de estudiantes en tres subgrupos y solicitar que cada subgrupo realice las tres tareas en diferente orden. Este experimento eliminaría el primer sesgo pero siempre mantendría el segundo. Además, en el período asignado para realizar una tarea, el estudiante podría tener otras responsabilidades que afectan su dedicación y concentración, e inciden en la duración para completar la tarea. Por estas razones, la comparación de estas herramientas usando la metodología de asignar tareas programadas a personas no es un buen enfoque.

Después de analizar las encuestas de cada curso, en la siguiente sección se explica el análisis de código.

8.4 Resultados del análisis de código

Se presentan los resultados del análisis del código fuente de varias tareas de los cursos *SD* y *PJAD*. En el curso *SD*, el análisis se enfocó en calcular el porcentaje de código de *D-Explorer* de las tareas. En el curso *PJAD* se comparó la cantidad de código ejecutable de los proyectos según tipo de herramienta usada (*D-Explorer*, *sockets* y *RMI*).

8.4.1 Curso *Sistemas Distribuidos*

Se enviaron correos a los estudiantes para solicitar el código fuente de la tarea SD-3. Se obtuvieron tres proyectos por este medio. Se asignaron los nombres SD3-1, SD3-2 y SD3-3 a estos proyectos. Se realizaron el análisis cuantitativo y el análisis de usos de servicios sobre estos tres proyectos.

En el proceso de obtención de las tareas se presenta un sesgo pues este método favorece la obtención de las tareas de los estudiantes que desean enviarlas o que sienten que sus tareas son buenas. No obstante, el correo para solicitar la tarea fue enviado a todos los estudiantes, quienes habían realizado las tareas con éxito, y no había razones de peso que impidieran que un estudiante no la entregara excepto tal vez la desidia o la falta de tiempo.

A continuación se presentan los resultados de cada análisis.

8.4.1.1 Análisis cuantitativo del código

Se calcula el *porcentaje de código de comunicación* (PCC). En la tabla 8.4 se presenta un

desglose del número *total de líneas de código de comunicación de D-Explorer* (TD).

Tabla 8.4. Líneas de código de *D-Explorer* sumarizado por tipo de código. La primera columna indica el tipo de código de *D-Explorer*. Las siguientes tres columnas presentan la cantidad de líneas de cada tipo de código para cada proyecto.

Tipo de código	SD3-1	SD3-2	SD3-3
Serialización	61	56	118
Conector Servicio	9	8	16
Notificaciones	15	5	7
Otros	37	21	22
Total de líneas de código de comunicación de D-Explorer	122	90	163

En la tabla 8.5 se calcula el porcentaje de código de comunicación para cada proyecto.

Tabla 8.5. Porcentaje de código de comunicación por proyecto. La primera fila presenta el total de líneas de comunicación de *D-Explorer* de cada proyecto. La segunda fila muestra el total de líneas de código de métodos de cada proyecto. La tercera fila divide los valores de la primera fila entre los valores de la segunda fila para obtener el porcentaje de código de comunicación.

Descripción	SD3-1	SD3-2	SD3-3
Total de líneas de código de comunicación de D-Explorer	122	90	163
Total de líneas de código de métodos	1374	595	1470
Porcentaje de código de comunicación	9%	15%	11%

Se analizaron los tipos de código de *D-Explorer* sumarizados en la tabla 8.4 para encontrar posibles mejoras que reduzca la cantidad de código de *D-Explorer* en la aplicación. Una mejora detectada fue la sustitución del mecanismo de serialización de *D-Explorer* por la serialización de Java. Con esta mejora, se eliminan las líneas de código de serialización en la aplicación y el canal de servicios serializa el mensaje con la serialización de *Java*.

Tabla 8.6. Porcentaje de código de comunicación por proyecto sin el código de serialización.

Descripción	SD3-1	SD3-2	SD3-3
Total de líneas de código de comunicación de D-Explorer	61	34	45
Total de líneas de código de métodos	1374	595	1470
Porcentaje de código de comunicación	4%	6%	3%

En la tabla 8.6 se presentan los porcentajes de código de comunicación después de aplicar la mejora descrita anteriormente. Estos porcentajes no superan el 6% e indican la gran utilidad de *D-Explorer* en estas tareas. Gracias a *D-Explorer*, los estudiantes codificaron poca lógica de comunicación y más bien se enfocaron en la lógica de procesamiento. En particular, el manejo de la concurrencia y de la asincronía, necesarios para resolver la

tercera tarea de *Sistemas Distribuidos*, los realiza el canal de servicios. Por el contrario, en una solución con *sockets* de esta misma tarea, el estudiante tendría que manejar la concurrencia y la asincronía. Para recibir mensajes asincrónicamente, se tendría que usar un hilo para escuchar mensajes entrantes de *sockets*. Para procesar mensajes concurrentemente, se tendría que despachar hilos para atender cada solicitud. Además, se debería sincronizar correctamente estos hilos para evitar el traslape de los mensajes. En la solución con *sockets* el estudiante tendría que manejar más detalles de comunicación.

Sin embargo, en una aplicación distribuida más sencilla como la tarea PJAD-1, la solución con *sockets* sería más fácil de implementar. En esta tarea, el servidor le envía un mensaje al primer cliente para que juegue, el cliente le responde con la jugada, luego el servidor le envía un mensaje al segundo cliente para que juegue y el cliente le responde con la jugada. El servidor intercambia mensajes con un cliente a la vez hasta que alguno gane o se detecte un empate. Para implementar esta tarea usando *sockets* solo se requiere una conexión sincrónica entre el servidor y cada cliente, y ni el servidor ni los clientes reciben más de un mensaje al mismo tiempo. La solución con *sockets* tendría menos líneas de código que la solución con *D-Explorer* pues no se ocuparía implementar las funciones complejas de concurrencia y de asincronía que asume el canal de servicios.

8.4.1.2 Uso de servicios de *D-Explorer*

Se analizó el uso de servicios de *D-Explorer* en los proyectos SD3-1, SD3-2 y SD3-3. En estos proyectos se usaron los siguientes componentes de *D-Explorer*: el canal de servicios, el mecanismo de descubrimiento, el servicio de directorio, el conector de servicio, el componente de servicios y el servicio de notificaciones. En estos proyectos, en el servidor se aloja el servicio de datos y en los clientes se registra un manejador de notificaciones para recibir mensajes del servidor asincrónicamente. En el proceso de revisión del código no se detectaron usos de los servicios de archivos y de sincronización.

El servicio más sobresaliente fue el servicio de notificaciones. Varios estudiantes del curso *SD* usaron este servicio en la segunda y tercera tarea para enviar notificaciones del servidor a los clientes. Estos estudiantes aprendieron el uso del servicio de notificaciones por su

propia cuenta, estudiando el *tutorial 02* del material de apoyo de *D-Explorer*.

El servicio de directorio fue usado para obtener la dirección del servicio de datos. Sin embargo, como este servicio está alojado en el servidor, se podría haber usado la dirección obtenida con el mecanismo de descubrimiento en vez de consultar al servicio de directorio. En el análisis de código se detectó que el servicio de datos tiene una operación para registrar clientes. Esta operación es invocada por los clientes al inicio de su ejecución y el método retorna cierta información al cliente. Este hecho indica que la aplicación si mantiene un directorio con la lista de clientes pero no se usa el servicio de directorio de *D-Explorer* pues los procesos, además de registrarse, necesitan intercambiar información de inicialización. El servicio de directorio solo permite registrar el nombre, el tipo y la dirección del servicio pero no intercambia información con el cliente.

El servicio de archivos se diseñó para dos usos principalmente: mantener archivos de configuración que los procesos leen al arrancar la ejecución y grabar archivos de resultados al finalizar la ejecución. En el código analizado, al arrancar la ejecución de un cliente, este recibe un mensaje del servidor con la lista de lecturas y escrituras que debe ejecutar, y al concluir la ejecución de un cliente, este envía un mensaje al servidor con el historial local de la ejecución, por lo que no hace falta grabar los resultados a un archivo remoto. Aunque la necesidad funcional existe, los proyectos analizados no usaron este servicio sino que intercambiaron mensajes pues el canal de servicios permite transmitir objetos complejos.

El servicio de sincronización tampoco fue usado a pesar de que cada cliente debe esperar a que todos los clientes estén activos para empezar la ejecución de su lista de lecturas y escrituras. En este caso, se podría haber usado la barrera del servicio de sincronización para coordinar el inicio de la ejecución de los clientes. Sin embargo, en el código revisado no se usó la barrera sino que se asoció un mecanismo local de sincronización con un mensaje para implementar la barrera. La figura 8.1 muestra este mecanismo en seudocódigo.

```
// Hilo principal
haga semaforoIniciaEjecucion.wait
// iniciar ejecución de lecturas y escrituras

// Hilo del manejador de notificaciones
Si se recibe una notificación para empezar la ejecución entonces
   haga semaforoInicioEjecucion.notify()
```

Figura 8.1. Implementación de una barrera simple.

En esta sección se analizó el uso de los servicios de *D-Explorer* en los proyectos. El análisis sugiere que el servicio de directorio, el servicio de archivos y la operación *barrera* del servicio de sincronización se podrían eliminar de *D-Explorer* pues existen mecanismo alternos que los estudiantes pueden implementar fácilmente y no ocupan aprender un API adicional. En el caso del servicio de directorio, además de esta desventaja también se detectó que los proyectos no solo se registran al servidor sino que intercambia información con este.

Con el análisis de uso de servicios concluye el análisis de código de *Sistemas Distribuidos*. Una ventaja de usar *D-Explorer* en aplicaciones distribuidas con comunicación asincrónica o concurrente es que no se necesitaría codificar lógica para atender estos requerimientos sino que esto son manejados por el canal de servicios. En la siguiente sección se presentan los resultados del análisis de código del curso *PJAD*.

8.4.2 Curso *Programación Java en Ambientes Distribuidos*

En el curso *PJAD*, los estudiantes programaron el juego cuatro en línea con *D-Explorer*, *sockets* y *RMI*. Hay tres estudiantes que entregaron a tiempo estas tres tareas. Se analizaron las tres tareas de estos estudiantes para eliminar algunos sesgos en el análisis. Se supone que estos estudiantes, por entregar las tareas a tiempo, entendieron bien cada herramienta y dedicaron suficiente tiempo en las tareas. Por el contrario, estudiantes que se atrasaron en alguna de estas no entendieron bien la herramienta o no pudieron dedicar suficiente tiempo. En total se analizaron 9 tareas: 3 de *D-Explorer*, 3 de *sockets* y 3 de *RMI*.

En el proceso de análisis, primero se hicieron copias de las tareas que usan *D-Explorer* y se modificaron las copias eliminando el código de serialización de mensajes de *D-Explorer*. Luego, se usó el software *Metrics* para obtener el total de líneas de código de métodos para

las 9 tareas de los estudiantes y las 3 tareas modificadas. Finalmente, se calcularon promedios por grupo de tareas: tareas hechas en *D-Explorer*, *sockets*, *RMI* y en *D-Explorer* mejorado (sin código de serialización).

Los resultados de la tabla 8.7 indican que la solución con *D-Explorer* ocupa más código que las soluciones con *sockets* y *RMI*. En la sección de trabajo futuro se sugieren mejoras para reducir la cantidad de código necesario para implementar este tipo de aplicaciones usando *D-Explorer*.

Tabla 8.7 Promedios por grupo de tareas. La primera columna muestra el tipo de métrica. Las siguientes cuatro columnas presentan los promedios de cada tipo de métrica para cada grupo de tareas (*D-Explorer*, *RMI*, *sockets* y *D-Explorer* mejorado).

	D-Explorer	*RMI*	*Sockets*	*D-Explorer*
Descripción				Mejorado
Líneas de código de métodos	639	447	455	539

Después del análisis de líneas de código, se hizo una revisión visual de las tres tareas que usan *D-Explorer*. En esta revisión se analizó el uso de servicios de *D-Explorer*. En estas tareas se usaron los siguientes componentes de *D-Explorer*: el canal de servicios, el mecanismo de descubrimiento, el servicio de directorio, el conector de servicio y el componente de servicios. En esta revisión también se detectó que una de las tres tareas hacía un mal uso de las herramientas. En *D-Explorer*, es posible registrar varios servicios en un mismo canal de servicios. Sin embargo, en esta tarea se crean varios canales y en cada canal se registra un único servicio. Este mal uso de *D-Explorer* complica la lógica del programa y produce más lineas de código pues se debe administrar más canales de lo necesario.

Con el análisis de código concluye el capítulo de Validación de la Utilidad. En el próximo capítulo se presenta una lista de trabajos futuros sobre *D-Explorer*.

9 Conclusiones

Se analiza el cumplimiento de los objetivos específicos.

1. *Identificar un conjunto de servicios y funcionalidades que faciliten la exploración de sistemas distribuidos.* Se cumplió satisfactoriamente este objetivo. Se identificó la función fundamental de comunicación pero se incorporaron otras funciones generales como localización, sincronización y almacenamiento remoto de datos, las cuales le agregan mayor riqueza funcional a *D-Explorer* (ver 3.1).

2. *Definir abstracciones y operaciones que provean dichas funcionalidades, teniendo en cuenta que se usarán en ambientes educativos.* Se cumplió satisfactoriamente este objetivo. Se definieron los siguientes componentes para proveer las funciones identificadas: canal de servicios, conector de servicio, componente de servicios, servicio de notificación, servicio de descubrimiento, servicio de directorio, semáforos, barreras, flujos remotos de datos y de texto (ver 3.2 a 3.5). En las funciones de comunicación se simplificó la interfaz de programación dividiendo estas funciones en tres componentes, lo cual permitió que el componente internamente, absorba muchas responsabilidades pero externamente, presente una interfaz de programación sencilla.

3. *Diseñar e implementar un componente distribuido que cumpla con las características definidas para cada servicio.* Se cumplió satisfactoriamente este objetivo. El diseño y la implementación de las herramientas se realizó sin contratiempos. En el capítulo 4 se presentan detalles de implementación de las partes más complejas de *D-Explorer*. En particular, el canal de servicios es bastante modular y extensible (ver 4.1).

4. *Verificar la correctitud de los componentes distribuido mediante un conjunto de pruebas que están definidas en la metodologia.* Se cumplió satisfactoriamente este objetivo. Con las pruebas concurrentes de los componentes se alcanzó un alto nivel de depuración de las herramientas (ver 6.1). Además, el uso de *D-Explorer* en las tareas programadas también confirmó la estabilidad de las herramientas (ver 8.2)

5. *Implementar un prototipo de aplicación distribuida o de algoritmo distribuido para verificar la funcionalidad de las herramientas*. Se cumplió satisfactoriamente este objetivo. En el prototipo se implementó una aplicación distribuida desacoplada con una lógica de comunicación compleja (ver 6.2).

6. *Preparar material de apoyo para facilitar el aprendizaje de las herramientas*. Se cumplió satisfactoriamente este objetivo. El material de apoyo fue útil. Algunos estudiantes aprendieron el servicio de notificaciones a través del *tutorial 02* y usaron este servicio en las tareas SD-2 y SD-3 (ver 6.2).

7. *Verificar la utilidad de las herramientas en ambientes académicos*. Se alcanzó este objetivo. Los estudiantes completaron las tareas, incluyendo la tarea SD-3 con una comunicación concurrente y asincrónica. Las tareas fueron diseñadas conjuntamente con los profesores y tienen un nivel de dificultad adecuado para el curso (ver 7.4). El tiempo usado por los estudiantes para resolver cada tarea está dentro de lo razonable (ver 8.3.1 y 8.3.2). En la tarea SD-3, los estudiantes alcanzaron bajos porcentajes de código de comunicación, menores o iguales al 6% (ver 8.4.1.1) pero en tareas más sencillas con comunicación secuencial y sincrónica como la tarea PJAD-1, la solución con *D-Explorer* requiere más código que las soluciones con *sockets* o *RMI* (ver 8.4.2).

Se analiza el cumplimiento del objetivo general: *identificar, diseñar e implementar un conjunto de herramientas que faciliten el desarrollo de prototipos de sistemas distribuidos, pruebas de concepto de algoritmos distribuidos, y proyectos y tareas en cursos semestrales de sistemas distribuidos en ambientes académicos*. Este objetivo fue alcanzado en gran medida. En el lado positivo, un grupo de estudiantes que no tienen conocimientos previos de *D-Explorer* logran desarrollar aplicaciones distribuidas con una lógica de comunicación compleja, como es el caso de la tercera tarea de *SD*. En el lado negativo, en aplicaciones distribuidas sencillas como la tarea SD-2 o PJAD-1, las soluciones de *D-Explorer* contienen más código que una solución en *sockets*. Sin embargo, este código es sencillo de codificar y en el capítulo de Trabajo Futuro se presentan varias mejoras que reducen la cantidad de código de *D-Explorer* en aplicaciones distribuidas.

En las siguientes secciones se analizan aspectos de funcionalidad y de facilidad de uso de las herramientas.

9.1 Funcionalidad

Los servicios internos de *D-Explorer* fueron implementados bajo el supuesto de que esta funcionalidad adicional facilitaría el desarrollo de aplicaciones distribuidas. Algunos de estos servicios cumplieron satisfactoriamente con las expectativas, otros servicios fueron sustituidos por mecanismos alternos implementados por los estudiantes y algunos servicios no fueron usados del todo.

El servicio más sobresaliente fue el servicio de notificaciones. Este servicio fue usado por varios estudiantes del curso *SD* para implementar la segunda y la tercera tarea. En estas dos tareas, el programa servidor enviaba notificaciones de diferentes tipos a los programas clientes a través de este servicio. También hay que resaltar el hecho de que estos estudiantes aprendieron el uso del servicio de notificaciones por su propia cuenta, estudiando el material de apoyo (*tutorial 02*) de *D-Explorer*.

Con base en el análisis de la sección *8.4.1.2 Uso de servicios de D-Explorer*, para una nueva versión de *D-Explorer* se debe evaluar la posibilidad de eliminar el servicio de directorio, de archivos y la *barrera* del servicio de sincronización.

9.2 Facilidad de uso

Se presentan varios aspectos de *D-Explorer* relacionados con su facilidad de uso.

Presenta el mismo API de comunicación para desarrollar aplicaciones distribuidas con comunicación sincrónica o asincrónica, y secuencial o concurrente (ver 3.2.5). El estudiante puede aprender este API implementando una aplicación distribuida con comunicación sincrónica y secuencial. Posteriormente, el estudiante puede usar este mismo conocimiento para desarrollar aplicaciones distribuidas asincrónicas o concurrentes pues no necesita implementar lógica adicional para manejar concurrencia o asincronía. En ambos tipos de aplicaciones, se realizan invocaciones similares para enviar mensajes y recibir mensajes, y los servicios y los manejadores de notificaciones se implementan de la misma

manera. Este fue el caso de las tareas del curso *SD*, en donde la comunicación en la segunda tarea es sincrónica y secuencial pero en la tercera tarea es asincrónica y concurrente.

Presenta un modelo de desarrollo basado en eventos (ver 3.2.5). El programador escribe métodos que procesan mensajes y registra estos métodos en el componente de servicios o el servicio de notificaciones. De esta manera, el programador se enfoca en la lógica de procesamiento y deja que *D-Explorer* maneje la comunicación.

Presenta un modelo de desarrollo modular (ver 3.2.5). Con este modelo se puede organizar una aplicación distribuida en un conjunto de servicios que se integran en el canal de servicios. En las tareas programadas asignadas, los estudiantes desarrollaron sus propios servicios y los registraron en el canal de servicios junto con otros servicios de *D-Explorer*. Cada servicio fue desarrollado de manera independiente, y en algunos casos por personas diferentes. Cada servicio operó de manera independiente y, a través de diferentes conectores de servicio, los servicios mantuvieron diálogos con diferentes clientes.

Esconde mucha de la complejidad asociada a la comunicación. El canal de servicios opera los *sockets* en modo asincrónico (ver 3.2.1). El conector de servicio serializa los mensajes y evita el traslape de estos al sincronizar el envío y la recepción de los mensajes serializados (ver 3.2.2). El componente de servicios maneja la concurrencia (ver 3.2.3).

Con la sección de aspectos de facilidad de uso de *D-Explorer* concluye el capítulo de conclusiones de la investigación, en donde el balance general es positivo. En el siguiente capítulo se presenta una lista de trabajos futuros, incluyendo el desarrollo del Lenguaje de Definición de Sistemas Distribuidos, un proyecto interesante que podría tener un alto impacto en simplificar el proceso de desarrollo de aplicaciones distribuidas.

10 Trabajo Futuro

En el capítulo de trabajo futuro se presentan diferentes trabajos que se pueden realizar para mejorar el proyecto. Hay tres tipos de trabajo: trabajos inmediatos para resolver algunos de los problemas detectados y descritos en los capítulos anteriores, trabajos técnicos a nivel de componentes y servicios de *D-Explorer* para ampliar el alcance del proyecto, y trabajos en el área de generación de código para mejorar el factor de usabilidad de *D-Explorer* en el desarrollo de aplicaciones distribuidas. Algunos trabajos surgieron como sugerencias o comentarios de los estudiantes expresados oralmente o por escrito, otros surgieron como resultado de los análisis de código o de las encuestas, y otros son extensiones naturales de componentes existentes o técnicas usadas en otros proyectos que han dado buenos resultados. En las siguientes secciones se explican estos tres grupos de trabajos.

10.1 Solución de problemas inmediatos

Se presentan cuatro mejoras que resuelven problemas inmediatos.

La primera mejora elimina el código de *Serialización* y una parte del código de *Declaración* de *D-Explorer*. En el análisis de código se determinó que en los proyectos SD3-1, SD3-2 y SD3-3, el promedio de líneas de código de *Serialización* es 78 y el promedio de líneas de código de *Declaración* es 66. La mejora consiste en sustituir el mecanismo de serialización de *D-Explorer* por el mecanismo de serialización de Java. Para implementar esta mejora, en la clase *Mensaje* se debe implementar la interface *java.io.Serializable* y en el proceso de serialización y deserialización del mensaje se debe usar el mecanismo de Java para convertir un objeto *java.io.Serializable* a bytes y viceversa. Con esta mejora, en el mensaje se eliminarían las líneas de *Serialización* y la declaración de los métodos *leerAtributosDelFlujo* y *grabarAtributosDelFlujo*.

La segunda mejora simplifica el código de *Otros*. En el análisis de código se determinó que el promedio de líneas de código de *Otros* en los proyectos SD3-1, SD3-2 y SD3-3 es 27. La mejora consiste en encapsular el programa principal (cliente o servidor) en una clase abstracta que contiene la lógica de inicialización y finalización. Esta clase tendría atributos

para configurar el proceso de inicialización y finalización, y un método abstracto en donde se implementaría la lógica que se ejecuta después de la inicialización y antes de la finalización. En un servidor, el código de inicialización activaría el canal y opcionalmente, activaría el servicio de directorio y el mecanismo de descubrimiento. En un cliente, el código de inicialización activaría el canal de servicios y opcionalmente, podría usar el mecanismo de descubrimiento para localizar el servidor. Estas clases tendrían atributos para definir el puerto de escucha del canal, indicar si desea usar el mecanismo de descubrimiento, etc. La figura 10.1 presenta el resultado de implementar estas mejoras.

```
public class ProgramaCliente extends dexplorer.ProgramaClienteBase
{
  @Override
  protected void logicaCliente()
  {
    // Implementar lógica del cliente
  }

  public static void main(String[] args)
  {
    ProgramaCliente programa = new ProgramaCliente();
    programa.setPuerto(30001);
    programa.setNombreAplicacion("calculadora simple");
    programa.setUsarDescubrimiento(true);
    programa.ejecutar();
  }
}
```

Figura 10.1 Programa cliente mejorado. Este programa tiene poco código y es muy legible.

La tercera mejora consiste en escribir un tercer tutorial que extiende el tutorial *01*. En el tutorial *01* se explica la implementación de un servicio en el servidor y el uso de este servicio desde el cliente. En este nuevo tutorial se implementaría un nuevo servicio en el cliente y se explicaría el uso de este servicio desde el servidor.

La última mejora consiste en escribir una sección de preguntas frecuentes (FAQ) para proveer respuestas inmediatas a dudas de los estudiantes. Aunque este tipo de información se puede obtener leyendo los tutoriales, este proceso es lento pues se debe estudiar el tutorial completo.

Después de presentar varias mejoras que resuelven problemas inmediatos, en la siguiente sección se describen mejoras a nivel de componentes y servicios de *D-Explorer*.

10.2 Sugerencias a nivel de componentes y servicios

En las mejoras a nivel de componentes y servicios, las sugerencias principales son para el canal de servicios. A continuación se explican estas mejoras:

1. Implementar diferentes componentes de transporte que permitan integrar dispositivos móviles o que usen protocolos como JMS, SMTP/POP3, etc. los protocolos JMS y SMTP/POP3 no requiere una conexión directa entre los nodos sino que el emisor deposita el mensaje en una cola JMS o un buzón de correo y el receptor revisa la cola o el buzón en cuanto le sea posible o cuando le convenga.

2. Activar varios componentes de transporte al mismo tiempo. Esta mejora facilitaría el desarrollo de aplicaciones que usen varios mecanismos de transporte, pues el desarrollador siempre usaría conectores de servicio para interactuar con los diferentes componentes de transporte. Esta mejora permitiría conectar nodos de diferentes tipos como estaciones de trabajo y dispositivos móviles.

3. Mejorar el enrutamiento de mensajes entre procesos de un mismo nodo. Cuando se envían mensajes de un conector de servicio a otro conector de servicio del mismo canal, estos mensajes no deberían salir hacia el componente de transporte sino que el canal debe enrutarlos internamente de un conector de servicios a otro.

4. Establecer prioridades entre los servicios. Actualmente, se maneja una única cola de mensajes entrantes y se atienden solicitudes por orden de llegada. Se podría manejar varias colas de mensajes entrantes, ordenadas de mayor prioridad a menor prioridad. Cuando un servicio se registra al canal de servicios, el servicio debería indicar cual es su nivel de prioridad. Cuando llega un nuevo mensaje, se usaría la prioridad del servicio destino para incluir el mensaje en la cola respectiva.

   Para el manejo de las colas de prioridades se puede implementar diferentes políticas. La más sencilla es atender de primero las solicitudes de las colas de mayor prioridad pero esto puede causar que las solicitudes de las colas de baja prioridad no sean atendidas o tengan que esperar mucho para ser atendidas.

5. Procesamiento de mensajes: esta mejora consiste en implementar una lista de

procesos encadenados que realizan operaciones sobre un mensaje. En esta lista de procesos, cada proceso recibe un mensaje, lo procesa y lo pasa al próximo proceso. Los procesos pueden modificar el mensaje realizando operaciones como comprimir o encriptar, o realizar operaciones pasivas como analizar el mensaje para actualizar estadísticas del canal o mantener una bitácora de mensajes entrantes y salientes. Habría una lista de procesos para procesar los mensajes entrantes y otra lista para procesar los mensajes salientes. Con esta mejora se podría comprimir mensaje, encriptar mensajes, llevar estadísticas de uso, etc.

6. Implementar servicios adicionales como elecciones, estados globales, etc.

En esta sección se explicaron mejoras que aumentan la funcionalidad de *D-Explorer*. En la siguiente sección se explican mejoras que aumenta la usabilidad de las herramientas.

10.3 Generación de código

En su estado actual, *D-Explorer* es una herramienta estable que facilita el desarrollo de aplicaciones distribuidas. La herramienta está compuesta por un conjunto de clases de Java empacadas en una biblioteca. Para desarrollar una aplicación distribuida, el programador hace una referencia a esta biblioteca y codifica la lógica de comunicación y la lógica de procesamiento de aplicación distribuida.

Un trabajo de investigación en el área de generación de código que podría mejorar el factor de usabilidad de *D-Explorer* en el desarrollo de aplicaciones distribuidas sería el desarrollo de un lenguaje de definición de sistemas distribuidos (*DSD*). Usando este lenguaje se definirían las características de una aplicación distribuida en términos de elementos de *D-Explorer* y esta definición se usaría para generar el esqueleto de la aplicación distribuida en código Java. Este esqueleto tendría la lógica de comunicación del sistema distribuido y el programador solo tendría que codificar la lógica de procesamiento.

Los elementos principales de una aplicación distribuida que usa *D-Explorer* son: los programas, los cuales activan una instancia del canal de servicios y contienen código para localizar y contactar otros programas, los servicios que se alojan en el canal de servicios y los manejadores de notificaciones que se agregan al servicio de notificaciones. En el

lenguaje *DSD* se definirían cláusulas del lenguaje que representen estos tres elementos de *D-Explorer*. Por ejemplo, se podrían definir las cláusulas *nodo*, *servicio* y *notificacion* para representar estos elementos. Además, entre los nodos del sistema se podrían definir relaciones como el *nodo A usa el servicio calculadora del nodo B* o *el servicio X envía mensajes al manejador de notificaciones Y*.

A continuación se muestran ejemplos de código del lenguaje *DSD*.

```
servicio Datos
{
  int leer(char variable);
..void grabar(char variable, int valor);
}

notificacion Invalidar
{
  void invalidar(char variable);
}
```

Figura 10.2 Declaraciones del servicio Datos y el manejador Invalidar. Los servicios y las notificaciones contienen una lista de las operaciones que soporta. A partir de estas declaraciones se generarían las clases en Java del servicio *Datos*, la notificación *Invalidar* y los mensajes que estos usan. Para detectar los atributos de los mensajes, se analizarían los parámetros de las operaciones. Por ejemplo, en el mensaje del servicio *Datos* se ocuparía el atributo *variable* de tipo *char*, el atributo *valor* de tipo *int* y un atributo entero para almacenar el resultado del método *leer*.

En la figura 10.2 se declaran el servicio *Datos* y el manejador de notificaciones *Invalidar*.

```
nodo ProgramaServidor
{
  puerto = 30001;
  servicio Descubrimiento;
  servicio Datos;
..usar notificacion Invalidar;
}

nodo ProgramaCliente
{
  puerto = automatico(30002);
  notificacion Invalidar;
  usar ServicioDatos;
}
```

Figura 10.3 Declaraciones del programa servidor y el programa cliente. La clausula *puerto* define el puerto de escucha del canal de servicios. En el programa servidor se define el puerto 30001. En el programa cliente la instrucción *automatico(30002)* indica que se use el primer puerto libre a partir de 30002. Las cláusulas *servicio* y *notificacion* en el cuerpo de la cláusula *nodo* definen los servicios y manejadores de notificaciones alojados en el nodo. La cláusula *usar* definen los servicios y manejadores de notificaciones usados por este nodo y se usaría para generar el código para conectarse a dichos servicios o manejadores de notificaciones.

En la figura 10.3 se declaran los nodos *ProgramaServidor* y *ProgramaCliente*.

Con las especificaciones de las figuras 10.2 y 10.3 se produciría el esqueleto de la aplicación distribuida. En este esqueleto habrían clases para representar los servicios, los manejadores de notificaciones, los mensajes, el programa servidor, el programa cliente, la lógica de inicialización y la lógica para conectarse a los servicios y manejadores.

Con la definición del lenguaje *DSD* concluye el capítulo de trabajos futuros.

Bibliografía

[1] Sun Microsystem, Inc., Java Platform, Standard Edition JDK 6, http://java.sun.com/javase/, accesado el 04-08-2007

[2] Postel, J., RFC 768 - User Datagram Protocol, http://www.faqs.org/rfcs/rfc768.html, 28-08-1980

[3] Postel, J., RFC 793 - Transmission Control Protocol, http://www.faqs.org/rfcs/rfc793.html, setiembre 1981

[4] Postel, J., RFC 791 - Internet Protocol, http://www.faqs.org/rfcs/rfc791.html, setiembre 1981

[5] Windows, marca registrada de Microsoft Corporation

[6] Linux, marca registrada de Linus Torvalds

[7] Palm OS, marca registrada de Palm Inc.

[8] Mac OS, marca registrada de Apple Inc.

[9] Lime Wire LCC, www.limewire.com

[10] SETI@HOME, http://setiathome.berkeley.edu/

[11] Winnet, J., RFC 147 - Definition of a socket, http://www.faqs.org/rfcs/rfc147.html, 07-05-1971

[12] Birrel, A., Nelson, B.J., Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, 1984

[13] Object Management Group, Common Object Request Broker Architecture: Core Specification, v3.0.3, http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf, accesado el 30 de setiembre del 2007

[14] Sun Microsystems, Inc., Java Remote Method Invocation (RMI), http://java.sun.com/javase/technologies/core/basic/rmi/, accesado el 30 de setiembre del 2007

[15] WebSphere MQ, marca registrada de International Business Machine Corporation

[16] MSMQ, marca registrada de Microsoft Corporation

[17] Object Management Group, CORBA 3.0 - IDL Syntax and Semantics chapter, http://www.omg.org/cgi-bin/apps/doc?formal/02-06-07.pdf, accesado el 30 de setiembre del 2007

[18] Smith III, H., A Simple Framework for Distributed Component-Based Systems, Journal of Computing Sciences in Colleges, 2007

[19] Bluetooth Special Interest Group, Core Specification v2.1 + EDR, http://www.bluetooth.com/Bluetooth/Technology/Building/Specifications/, accesado el 18 de marzo del 2008

[20] Coulouris, G., Dollimore, J., Kindberg, T., Distributed Systems Concepts and design, 2005

[21] Cristian, F., Probabilistic clock synchronization, Distributed Computing, 1989

[22] Gussella, R., Zatti, S., TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System, Proc. Summer 1984 USENIX, 1984

[23] World Wide Web Consortium, Extensible Markup Language (XML) 1.0 (Fourth Edition), http://www.w3.org/TR/2006/REC-xml-20060816, accesdo el 4 de octubre del 2007

[24] Market Share, Operating System Market Share, http://marketshare.hitslink.com/report.aspx?qprid=8, accesado el 14 de mayo del 2008

[25] Mockapetris, P., Domain Names - Concepts and Facilities, http://www.ietf.org/rfc/rfc1034.txt, noviembre 1987

[26] Sun Microsystems, Inc., Java Message Service Specification - version 1.1, http://java.sun.com/products/jms/docs.html, accesado el 4 de octubre del 2007

[27] Chandor, A., editor, Dictionary of Computers, 1970

[28] Wolfgang, E., Aoyama, M., Sventek, J., The Impact of Research on Middleware Technology, ACM SIGSOFT Software Engineering Notes, 2007

[29] Ricart, G., Agrawala, A., An optimal algorithm for mutual exclusion in computer networks, Communications of the ACM, 1981

[30] ITU/ISO, Recommendation X.500: Open Systems Interconnection - The Directory: Overview of concepts, models and services, 1997

[31] Maekawa, M., A Square Root N Algorithm for Mutual Exclusion in Descentralized Systems, ACM Transactions on Computer Systems, 1985

[32] Goetz, B. with Peierls, T. et. al., Java Concurrency in Practice, 2006

[33] Nelson, M., Welch, B., Ousterhout, J. , Caching in the Sprite Network File System, ACM Transactions on Computer Systems, 1988

[34] Knuth, D., Art of Computer Programming, volume 3: Sorting and Searching, 1998

[35] Torres-Rojas, F., Ahamad, M., Raynal, M., Lifetime Based Consistency Protocols for Distributed Objects, Proceedings of the 12th International Symposium on Distributed Computing, 1999

[36] Mattern, F., Virtual time and global states of distributed systems, Proceedings of the International Workshop on Parallel and Distributed Algorithms, 1988

[37] Metrics, metrics.sourceforge.net

[38] Torres-Rojas, F., Comunicación Personal, 2009

[39] ArgüelloVenegas, R., Comunicación Personal,

Apéndice A

Encuesta aplicada a los estudiantes de los cursos *Sistemas Distribuidos* y *Programación Java para Ambientes Distribuidos*. Hay dos tipos de encuestas, la encuesta completa y la encuesta parcial, las cuales se presentan a continuación.

A.1 Encuesta Completa

La encuesta completa se aplicó a los estudiantes de ambos cursos cuando estos entregaron su primera tarea programada.

D-Explorer- Encuesta 01

El presente cuestionario consta de un conjunto de preguntas sobre sus conocimientos en programación de redes y su experiencia con *D-Explorer* como herramienta de desarrollo de aplicaciones distribuidas.

En las siguientes preguntas, conteste con un número entre 0 y 10. El número 0 representa una respuesta negativa (no está de acuerdo, influye muy poco, conoce muy poco). El número 10 representa una respuesta positiva (está totalmente de acuerdo, influye mucho, conoce mucho). Si en alguna pregunta Ud. no puede responder, escriba **N/A** para indicar que no aplica. Por ejemplo, en la pregunta sobre el tutorial 02, si Ud no leyó el tutorial 02, responda con **N/A**.

1) Indique el nivel de conocimientos que tiene sobre los siguientes tópicos.

_____ a) Tiene conocimientos sobre el protocolo TCP/IP.

_____ b) Ha programado *sockets*.

_____ c) Ha programado hilos.

_____ d) Tiene experiencia en sincronizar hilos. Sabe como implementar una región crítica, un semáforo, etc.

2) La distribución de *D-Explorer* contiene el código fuente de las herramientas, la documentación de clases generado con JavaDoc, dos proyectos de ejemplo (tutorial 01 y tutorial 02) y un conjunto de páginas html que explican el uso de las herramientas.

_____ a) El material de la distribución es suficiente para que un estudiante pueda aprender el uso de las herramientas.

_____ b) La organización del material es adecuada para el aprendizaje de las herramientas.

_____ c) El proyecto de ejemplo *Tutorial 01* le ayudó a entender la implementación de un servicio usando *D-Explorer*. El proyecto escogido es adecuado y la explicación de las clases del proyecto es clara.

_____ d) El proyecto de ejemplo *Tutorial 02* le ayudó a entender el uso de las notificaciones. El proyecto escogido es adecuado y la explicación de las clases del proyecto es clara.

3) En esta sección se incluyen preguntas relacionadas con el uso de *D-Explorer* en el desarrollo de aplicaciones distribuidas.

_____ a) Pude programar una aplicación distribuida que implementa un servicio sencillo.

_____ b) Pude programar una aplicación distribuida que usa notificaciones.

_____ c) Pude ejecutar los programas que desarrollé en una misma computadora y no tuve problemas.

_____ d) Pude ejecutar los programas que desarrollé en varias computadoras de una LAN y no tuve problemas.

_____ e) Tengo experiencia programando *sockets*. Considero que muchas de las cosas que he programado usando *sockets*, también se puede hacer con *D-Explorer*.

_____ f) Siguiendo con la pregunta anterior, también considero que muchas de las cosas que he programado usando *sockets*, se podría programar más fácilmente usando *D-Explorer*.

4) En esta sección se incluyen preguntas relacionadas con la tarea programada completada.

_____ a) Indique el nivel de dificcultad de la tarea. El cero significa que es muy fácil. El diez indica que la tarea tiene un alto grado de dificultad.

_____ b) Indique las horas dedicadas para programar la tarea. Indique únicamente el tiempo de programación. Excluya el tiempo usado para estudiar los tutoriales y la documentación de la distribución. Excluya también el tiempo dedicado a análisis y diseño del problema

_____ c) Si ha realizado alguna tarea programada similar usando *sockets*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta.

_____ d) Si ha realizado alguna tarea programada similar usando *RMI*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta

5) Espacio abierto para críticas, comentarios, sugerencia sobre otro tipo de material que facilite el aprendizaje y uso de las herramientas, otros proyectos tutoriales, etc.

Gracias por su colaboración.

En el eventual caso de que tenga dudas o necesite aclaraciones sobre alguna de sus respuestas, me gustaría enviarle un correo para hacerle la respectiva consulta. Si no hay inconveniente, indique su correo electrónico en el siguiente campo.

Correo electrónico: _____

A.2 Encuesta Parcial

La encuesta parcial se aplicó a los estudiantes del curso de *Sistemas Distribuidos* cuando estos entregaron la segunda y la tercera tarea programada.

D-Explorer- Encuesta 02

El presente cuestionario consta de un conjunto de preguntas sobre la reciente tarea programada realizada usando *D-Explorer*.

1) En esta sección se incluyen preguntas relacionadas con la tarea programada realizada.

_____ a) Indique el nivel de dificultad de la tarea. El cero significa que es muy fácil. El diez indica que la tarea tiene un alto grado de dificultad.

_____ b) Indique las horas dedicadas para programar la tarea. Indique únicamente el tiempo de programación. Excluya el tiempo usado para estudiar los tutoriales y la documentación de la distribución. Excluya también el tiempo dedicado a análisis y diseño del problema.

_____ c) Si ha realizado alguna tarea programada similar usando *sockets*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta.

_____ d) Si ha realizado alguna tarea programada similar usando *RMI*, indique el número de horas usadas para programar dicha tarea. Indique únicamente el tiempo de programación. Excluya el tiempo de análisis, diseño y aprendizaje de la herramienta.

Gracias por su colaboración.

Apéndice B

Se presentan los datos de las encuestas aplicadas a los estudiantes de los cursos *Sistemas Distribuidos* y *Programación Java para Ambientes Distribuidos*.

B.1 Primera encuesta del curso *Sistemas Distribuidos*

Se presentan los datos numéricos de las encuestas y los comentarios de los encuestados correspondientes a la primera tarea programada del curso *Sistemas Distribuidos*.

Tabla B.1. Datos de la primera encuesta del curso de *Sistemas Distribuidos*. La primera columna corresponde a la pregunta. Las siguientes 9 columnas contienen las respuestas de los estudiantes. Las últimas dos columnas contienen el promedio y la desviación estándar de las respuestas de los estudiantes. Las celdas en blanco corresponden a respuestas en blanco o respondidas con **N/A** (no aplica) y estas celdas no se usan en el cálculo del promedio ni de la desviación.

	Estudiante										
Pregunta	1	2	3	4	5	6	7	8	9	Promedio	Desviación
1.a	10	5	5	2	7	5	6	7	5	5.78	2.04
1.b	10	6	0	7	5	0	10	7	5	5.56	3.44
1.c	10	8	5	10	10	0	10	8	7	7.56	3.13
1.d	8	8	1	10	8	0	10	9	3	6.33	3.68
2.a	7	9	6	10	8	10	7	8		8.13	1.36
2.b	9	9	9	8	4	8	9	8	8	8.00	1.49
2.c	9	10	8	6	10	9	10	9	8	8.78	1.23
2.d								7		7.00	0.00
3.a	10	10	7	10	10	10	0	10	10	8.56	3.17
3.b								9		9.00	0.00
3.c	10	10	5	5	10	10	0	10	10	7.78	3.42
3.d					8		0			4.00	4.00
3.e	6	8		8	5		10	10		7.83	1.86
3.f	4	9		9	8		5	10	7	7.43	2.06
4.a	3	2	3	2	2	5	1	8	5	3.44	2.06
4.b	2	8	5	3	2	2	1	3	1	3.00	2.11
4.c					24		3		3	10.00	9.90
4.d					24		3			13.50	10.50

Seguidamente se presentan los comentarios de cada estudiante. Se hicieron correcciones ortográficas en los comentarios.

Estudiante 1. Existen tecnologías que permiten hacer lo mismo, más fácil. Por ejemplo Web Services. Licencias de desarrollo son gratuitas para educación (.NET). Debería

implementar mecanismos estándares de serialización ya existentes.

Estudiante 2. A veces da problemas al usarlo en una máquina con VPN. Aunque no es muy crítico, si es un requisito que podría analizarse.

Estudiante 3. Quizás un poco de documentación más detallada puede ayudar a entender mejor la filosifía de *D-Explorer*. Algunas características que ahora conozco de esta herramienta no la pude descubrir mediante el tutorial.

Estudiante 4. Se "come" el procesador al 100%. Algún hilo debe estar sin un *sleep*. Más opción de configurar sin tener que cambiar código.

Estudiante 5. Sin comentarios.

Estudiante 6. El uso del CPU se elevó al 100%, lo cual no me permitió ejecutar el programa de una manera rápida y no pude tener otras aplicaciones abiertas al mismo tiempo. Otro punto es que los servicios deberían permitir implementar varios métodos, es decir, el servicio calculadora debería tener los métodos de *sumar*, *restar*, *dividir*, etc y no solo uno.

Estudiante 7. No corrió en los laboratorios de la maestría.

Estudiante 8. El framework provisto por *D-Explorer* hace un alto uso del procesador. Probablemente existe una espera activa. El mecanismo de descubrimiento implica que la computadora en que se desarrolla está conectada a la red, lo que restringe el desarrollo. Probablemente sería bueno contar con la capacidad de configurarlo sin necesidad de usar el mecanismo de descubrimiento.

Estudiante 9. Lo que pude ver mientras realizaba la tarea fue el problema del servidor que utilizaba el 100% del procesador, pero me gustó la base sobre la cual está montada.

B.2 Segunda encuesta del curso *Sistemas Distribuidos*

Se presentan los datos de las encuestas correspondientes a la segunda tarea programada del curso *Sistemas Distribuidos*.

Tabla B.2. Datos de la segunda encuesta del curso de *Sistemas Distribuidos*. La primera columna corresponde a la pregunta. Las siguientes 9 columnas contienen las respuestas de los estudiantes. Las últimas dos columnas contienen el promedio y la desviación estándar de las respuestas de los estudiantes. Las celdas en blanco corresponden a respuestas en blanco o respondidas con **N/A** (no aplica) y estas celdas no se usan en el cálculo del promedio ni de la desviación.

	Estudiante										
Pregunta	1	2	3	4	5	6	7	8	9	Promedio	Desviación
4.a	8	3	3	4	6	6	4	4		4.75	1.64
4.b	30	4	8	8	8	8	5	5		9.50	7.91
4.c			24	24						24.00	0.00
4.d				24						24.00	0.00

B.3 Tercera encuesta del curso *Sistemas Distribuidos*

Se presentan los datos numéricos de las encuestas correspondientes a la tercera tarea programada del curso *Sistemas Distribuidos*.

Tabla B.3. Datos de la tercera encuesta del curso de *Sistemas Distribuidos*. La primera columna corresponde a la pregunta. Las siguientes 9 columnas contienen las respuestas de los estudiantes. Las últimas dos columnas contienen el promedio y la desviación estándar de las respuestas de los estudiantes. Las celdas en blanco corresponden a respuestas en blanco o respondidas con **N/A** (no aplica) y estas celdas no se usan en el cálculo del promedio ni de la desviación.

	Estudiante										
Pregunta	1	2	3	4	5	6	7	8	9	Promedio	Desviación
4.a	7	8	5	7	9	8				7.33	1.25
4.b	16	25	36	10	30	28				24.17	8.73
4.c	36		40		40					38.67	1.89
4.d			40							40.00	0.00

B.4 Encuesta del curso *Programación Java en Ambientes Distribuidos*

Se presentan los datos numéricos de las encuestas y los comentarios de los encuestados correspondientes a la tarea programada del curso *Programación Java en Ambientes Distribuidos*.

Tabla B.4. Datos de la encuesta del curso de *Programación Java en Ambientes Distribuidos*. La primera columna corresponde a la pregunta. Las siguientes 8 columnas contienen las respuestas de los estudiantes. Las últimas dos columnas contienen el promedio y la desviación estándar de las respuestas de los estudiantes. Las celdas en blanco corresponden a respuestas en blanco o respondidas con **N/A** (no aplica) y estas celdas no se usan en el cálculo del promedio ni de la desviación.

	Estudiante									
Pregunta	1	2	3	4	5	6	7	8	Promedio	Desviación
1.a	10	4	5	7	7	6	7	7	6.63	1.65
1.b	8	9	10	8	8	7	8	5	7.88	1.36
1.c	8	9	10	8	7	7	7	5	7.63	1.41
1.d	7	8	7	7	6	8	6	5	6.75	0.97
2.a	8	0	3	7	4	6	6	6	5.00	2.40
2.b	8	4	5	8	4	6	6	7	6.00	1.50
2.c	9	1	8	9	5	7	8	7	6.75	2.49
2.d		0	5				7		4.00	2.94
3.a	10	10	10	10	7		10	8	9.29	1.16
3.b	7	5		7	7		0		5.20	2.71
3.c	10	10	10	8	8		9	7	8.86	1.12
3.d							10		10.00	0.00
3.e	7	5	4	9		6	9	5	6.43	1.84
3.f	5	1	4	7	6	5	8	5	5.13	1.96
4.a	7	8	8	7	7	10	5	5	7.13	1.54
4.b	12	20	21	10	10	15	18	20	15.75	4.32
4.c	8	10	8	12	8	20	12	12	11.25	3.73
4.d	6	8	8	8	10	12	18	8	9.75	3.53

Seguidamente se presentan los comentarios de cada estudiante. Se hicieron correcciones ortográficas en los comentarios.

Estudiante 1. En la tarea programada del curso, me tomó menos tiempo desarrollar en *sockets* y *RMI* que en *D-Explorer*. Considero que esto se debe al diseño específico del juego mismo, ya que no era la arquitectura tradicional de cliente/servidor, sino que ambas partes de la comunicación era al mismo tiempo clientes y servidores, lo cual a nivel de

implementación con *D-Explorer* significó mucho tiempo extra invertido en entender y resolver conflictos, principalmente en cuanto a puertos y nombres de los servicios en el directorio. Con respecto a este punto, tal vez sería adecuado agregar un ejemplo de este tipo para que los estudiantes puedan aprender más fácilmente como se resuelve. De igual manera podría servir para comprender más a fondo como funciona *D-Explorer* en general.

Estudiante 2. Creo que la parte que más se puede reforzar es la cantidad y/o variedad de ejemplos. Los 2 ejemplos que incluye la documentación son muy sencillos y expone el potencial del *tool*. Por ejemplo, como integrarle *D-Explorer* a una aplicación más compleja que una suma. Algo que tenga más objetos. Otra cosa que hace falta es la implementación de ejemplos que incluya más de 2 actores con diferentes roles. Me parece que el tutorial o el documento está bien escrito en cuanto a la prosa, pero no es muy amigable. Podría ser útil seguir un tutorial modelo como el estilo *Deitel y Deitel*, obviamente habría que investigar, hacer un censo. Agradezco mucho la disponibilidda del creador para aclarar dudas.

Estuduante 3. Siento que el tutorial no es suficiente para un proyecto más complejo que el explicado en la documentación. Tuve que revisar el código fuente en varias ocasiones para intentar entender la lógica y esta debería ser transparente para el usuario si es que la idea es ocultar la parte tediosa o "cajonera" del manejo de *sockets*. Siento que la herramienta es buena pero poco flexible y restringe las posibilidades al no tener tanto control de las cosas como con *sockets*.

Estudiante 4. Se debería incluir un ejemplo de un modelo donde cada "nodo" pueda ser cliente y servidor a la vez. Sentí que tal vez no hubo necesidad de "rehacer" muchas clases que ya se incluye en Java. Su manejo de excepciones podría ser mejor, por lo menos en los ejemplos. En una ocasión tuve problemas para determinar que tipo de excepción estaba arrojando la aplicación. Esto se debía a que el uso de los *try/catch* no era el óptimo. Es mi caso, invertí más tiempo tratando de utilizar el framework en un modelo en el que una misma terminal podía ser cliente y servidor a la vez.

Estudiante 5. Me parece que los ejemplos debieron ser más complejos y más numerosos. Sería útil que en la documentación incluyera más ejemplos y que estos, progresivamente, se

fueran haciendo más complejos.

Estudiante 6. En realidad me costó mucho adaptar, por medio de los ejemplos que se muestran en la distribución, a la tarea programada asignada al curso, el número de horas que hay que dedicar para entender la lógica. Por cuestiones de tiempo no pude implementar por completo, entonces sinceramente no digo que la distribución sea mala o buena, sin embargo hay que dedicar tiempo.

Estudiante 7. En general me parece fácil de "montar ", sin embargo yo personalmente me siento mejor guiado con el *javadoc* de una aplicación, que me pareció que faltó en el caso de *D-Explorer*.

Estudiante 8. En las preguntas 3.c y 3.d no poseo mucha experiencia en programar *sockets*, solo el ejemplo de la tarea, por lo cual la comparación se basa solo en ese ejemplo. Además, dicha comparación se puede ver reflejada en el número de horas invertidas en la tarea. Con respecto a *D-Explorer*, considero que debería proveer más ejemplos, preferiblemente sencillos, en que se vea más comunicación bidireccional, ya que en el ejemplo de la tarea la comunicación unidireccional no fue tan complicada como para poder establecer servicios del cliente para el servidor, ya que no es simplemente copiar lo mismo del cliente para el servidor, había que tomar en cuenta cosas que ya habían sido abiertas o creadas para la comunicación de vuelta, entre otras cosas, que no son tan transparentes para usuarios inexpertos en *D-Explorer*.

Apéndice C

Se presentan los enunciados de las tres tareas programadas del curso *Sistemas Distribuidos* y el enunciado de la tarea programada del curso *Programación Java para Ambientes Distribuidos*.

C.1 Implementar un servicio sencillo

La primera tarea programada del curso *Sistemas Distribuidos* consiste en implementar un servicio sencillo usando *D-Explorer*.

Enunciado

Implementar un servicio sencillo usando *D-Explorer*. El estudiante debe implementar un programa servidor que aloja el servicio y un programa cliente que usa el servicio. El programa servidor recibe un mensaje de solicitud del cliente, procesa la solicitud y devuelve un mensaje con la respuesta al cliente. El programa cliente envía una solicitud al servidor y espera la respuesta. Algunos ejemplos de servicios sencillos que pueden implementar son: sumar dos números, contar las letras de un mensaje de texto, etc.

C.2 Tarea Caché con Invalidación

La segunda tarea programada del curso *Sistemas Distribuidos* consiste en implementar el esquema de caché con invalidación usando *D-Explorer*.

Enunciado

Implementar el esquema de caché con invalidación usando *D-Explorer*. El estudiante debe implementar un servidor de datos y dos clientes.

Programa servidor de datos.

El servidor de datos mantiene una lista de 10 datos enteros, cada dato está identificado por una letra (A, B, C, ...). El servidor tiene dos operaciones: leer un dato y grabar un dato.

Como consecuencia de la operación "*grabar un dato*", el servidor podría invalidar el caché de algún cliente. Por ejemplo, si el cliente 1 lee el dato A y posteriormente el cliente 2 graba el dato A entonces el servidor debería invalidar el dato A del caché del cliente 1. Sin embargo, si el cliente 1 no tiene el dato B en su caché y el cliente 2 graba el dato B, el servidor no debería enviar un mensaje al cliente 1 para invalidar el dato B.

El servidor debe desplegar una bitácora en pantalla de lo que está haciendo:

Cliente 1: leer(A) -> 3.

Cliente 2: grabar(A,15)

Invalidar A en cliente 1.

...

Programa cliente.

Para correr dos copias del programa cliente, el estudiante debe asignar diferentes puertos a cada programa cliente. Esta asignación se realiza en el método *configurarPuertoFijo*:

```
dexplorer.canal.Fabrica.configurarPuertoFijo( ??? );
```

Los programas clientes usan el servidor de datos pero mantienen un caché local. El estudiante debe programar una interfaz de usuario que le permita al usuario leer un dato, grabar un dato y desplegar el contenido del caché local. Puede ser una interfaz sencilla basada en una consola de caracteres o puede implementar una interfaz gráfica usando los componentes gráficos de Java.

Leer un dato. El usuario debe digitar el dato que desea leer (A, B, C, ...) y el programa cliente debe desplegar el valor de este dato. Si el dato está en el caché local, se usa este dato y se despliega su valor al usuario. Si el dato no está en el caché local, el cliente debe obtener este valor del servidor y actualizar su caché.

Grabar un dato. El usuario debe digitar el dato a modificar (A, B, C, ...) y el nuevo valor del dato (un número entero). El programa cliente debe actualizar su caché local y debe actualizar el servidor de datos.

Listar el contenido del caché. Se debe desplegar los datos que están en el caché local del cliente: A=3, D=1, F=4.

El programa cliente puede recibir mensajes de invalidación del servidor de manera asíncrona. En estos casos, el programa debe desplegar un rótulo que indique cuál dato se está invalidando y el nuevo estado del caché.

C.3 Caché con protocolos de *Lifetime Based Consistency*

La tercera tarea programada del curso *Sistemas Distribuidos* consiste en implementar el esquema de caché con el protocolo *Lifetime Based Consistency* [35] usando *D-Explorer*.

Enunciado

Implementar los protocolos de *lifetime-based-consistency* descritos en el artículo ***Lifetime based Consistency Protocols for Distributed Objects*** de *Francisco Torres-Rojas, Mustaque Ahamad y Michel Raynal*. Se debe implementar la variante que usa relojes lógicos vectoriales.

Para cada variante, el estudiante debe usar *D-Explorer* para implementar un servidor y cuatro clientes. Los clientes no son interactivos sino que le pide una lista de operaciones de lectura y escritura al servidor, ejecuta esta lista de operaciones y le envía el historial local de la ejecución al servidor. Cuando el servidor recibe los historiales de todos los clientes, debe verificar que los historiales recibidos cumplen con la condición de consistencia del protocolo.

El siguiente diálogo describe el orden de ejecución del sistema. La S denota al servidor y Ci denota a cada uno de los clientes.

S:Generar lista de operaciones de cada cliente (lecturas y escrituras de datos)

Ci:Registrarse con el servidor y recibir como respuesta el identificador del cliente (1, 2, 3 o 4) y la lista de operaciones que debe ejecutar.

S: Cuando todos los clientes se han registrado, enviar un mensaje a cada cliente para que empieze a ejecutar su lista de operaciones.

Ci:ejecuta su lista de operaciones. En este paso, cada cliente realiza operaciones de lectura y de escritura sobre los datos. Estas operaciones pueden completarse localmente accesando únicamente el caché del cliente o pueden involucrar al servidor y otros clientes. Cuando termina de ejecutar la lista de operaciones, el cliente envía su historial local al servidor y queda en espera sin apagar el canal pues podría recibir mensajes de invalidación.

S:Cuando recibe los historiales locales de todos los clientes, verifica que estos historiales cumplan con la condición de consistencia. El servidor debe salvar las listas de operaciones, los historiales locales y la información necesaria para verificar que la corrida cumple con la condición de consistencia.

Verificación.

El servidor debe recibir los historiales de cada cliente y verificar que satisfacen la condición de consistencia para el protocolo B de relojes vectoriales. La verificación debe realizarse de acuerdo a los procedimientos explicados por el profesor.

C.4 Cuatro en Línea Distribuido

La primera tarea programada del curso *Programación Java para Ambientes Distribuidos* consiste en convertir la versión no distribuida del juego *Cuatro en Línea* a una versión distribuida para dos jugadores usando *D-Explorer*.

Enunciado

El estudiante debe usar *D-Explorer* para implementar el juego cuatro en línea distribuido para dos jugadores. Se debe implementar el servidor del juego y dos programas clientes. El servidor contiene la lógica del juego: recibe mensajes con las jugadas de cada jugador, actualiza el estado del juego y retorna mensajes con el estado del juego. El cliente despliega el estado del juego, solicita la jugada al jugador, lo envía al servidor y espera su respuesta.