Documentación de la Librería Distributed Resource Machine

Alberto Cuesta Cañada

Una tésis presentada para el título de
Máster en Computación Paralela y Distribuida

dirigida por

Eva Alfaro Cid
Jordi Bataller i Mascarell

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

6 de noviembre de 2007

Dedicado a María José

Documentación de la Librería Distributed Resource Machine

Alberto Cuesta Cañada

Entregado para el título de

Máster en Computación Paralela y Distribuida

6 de noviembre de 2007

Resumen

La librería Distributed Resource Machine (DRM) sirve para implementar aplicaciones masivamente distribuidas mediante programas Java que pueden ejecutarse y viajar a través de internet. DRM crea topologías altamente escalables y difunde información en ellas de un modo eficiente mediante el algoritmo newscast.

El objetivo de esta tesis es documentar la plataforma mediante el análisis del algoritmo newscast y de la implementación de la librería DRM. Se completa esta información mediante un tutorial de uso con seis ejemplos.

Esta tesis, basada en investigaciones realizadas durante dos años de trabajo, supone la investigación más exhaustiva hasta la fecha sobre Distributed Resource Machine.

Agradecimientos

Gracias a Ken C. Sharman y a Anna I. Esparcia Alcázar por enseñarme tantas cosas que no se aprenden en un aula. Gracias a Nacho Blanquer, Francisco Muñoz y Antonio Vidal por ser profesionales ejemplares y personas excelentes. Gracias a Andrés, Eli, Rober, Gabri, Julio, Murilo, Miguel, Matías y Esther por acompañarme en la aventura de inaugurar este Máster. Gracias a mis padres, Alberto y Esperanza, por demasiadas cosas para esta página.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Definición del problema

Cuando el autor comenzó a estudiar la biblioteca DRM y el algoritmo newscast la información existente era escasa y fragmentada. Sobre el algoritmo newscast se podían consultar algunos trabajos que explicaban su funcionamiento y daban muestras de su efectividad y características, aunque esta información distaba mucho de ser un análisis completo.

No existía más información acerca de la implementación de DRM que el propio código y unas breves explicaciones dentro del mismo. Lo único disponible para desarrollar programas que la utilizasen era un breve tutorial y cuatro ejemplos prácticos.

Dos años de trabajo por parte del autor han producido suficiente material para remediar en parte esta situación. La implementación y uso de DRM han dejado de ser un misterio, y existe más información sobre el algoritmo subyacente que puede ayudar a evaluar la eficacia de DRM en contextos de uso real. Esta tesis recoge toda la información existente sobre DRM y newscast producida por el autor junto con algunos resultados previos de otros autores para dar una visión global de esta tecnología.

1.2. Objetivos y contribución

El objetivo de esta tesis es documentar la biblioteca DRM. Mediante la información aportada se pretende que quien lo desee pueda trabajar con esta herramienta partiendo de una base adecuada. Se espera que los usuarios de la biblioteca DRM puedan desarrollar aplicaciones distribuidas descentralizadas que aprovechen al máximo las oportunidades ofrecidas.

Para conseguir estos objetivos se ha trabajado en caracterizar el algoritmo newscast, que es la pieza fundamental de DRM. Este trabajo aporta el único intento de definir detalladamente este algoritmo hasta la fecha. También se ofrece el estudio más detallado existente de la topología de la red creada por este algoritmo basado en la teoría de sistemas complejos.

Como referencia y ayuda a la implementación de la biblioteca DRM se ha hecho una amplia descripción de los paquetes y clases que la componen. Se explican en detalle todos los elementos que forman una aplicación DRM distribuida: nodos, agentes, direcciones o mensajes, entre otros. Se detallan los procedimientos y protocolos utilizados para realizar acciones como crear o enviar agentes o mensajes. Se explica la implementación del algoritmo newscast mediante vistas parciales de un objeto global llamado colectivo. Muchos otros detalles de la implementación se exponen por vez primera.

Un trabajo como éste se quedaría incompleto sin ejemplos de utilización de la biblioteca, y por eso se han incluido aplicaciones DRM que ilustran los conceptos estudiados en este trabajo. Acompañan a estas aplicaciones otras creadas por el desarrollador original de DRM para componer un tutorial completo que es actualmente la mejor opción para iniciarse en el manejo de DRM.

1.3. Organización del documento

El capítulo 2 proporciona una base sobre la que estudiar el algoritmo newscast y la biblioteca DRM. Comienza por una introducción a la computación en redes de pares mediante la teoría de sistemas complejos. Éste es un enfoque no muy habitual que sin embargo fue el escogido para dar lugar al proyecto DR-EA-M que desarrolló

la biblioteca. El capítulo concluye con una exposición de este proyecto y del algoritmo subyacente en su capa de comunicaciones.

El capítulo 3 contiene los avances del autor sobre el algoritmo newscast que construye la topología y distribuye la información en la red DRM. Se muestra en este capítulo una definición detallada del algoritmo y una descripción de la topología creada a partir de la interacción de un gran número de elementos simples.

El capítulo 4 recopila, traduce y completa la información existente sobre la biblioteca DRM, construida con base en el algoritmo newscast para cumplir los objetivos del proyecto DR-EA-M. Se ofrece una descripción detallada de las clases y los protocolos empleados.

El capítulo 5 toma el único tutorial previo sobre DRM y lo completa mediante tres ejemplos nuevos que cubren las lagunas existentes en el uso de la biblioteca.

Por último, en el capítulo 6 se ofrece una pequeña discusión sobre la biblioteca, su pasado y su futuro, y sobre las contribuciones aportadas en este trabajo.

Capítulo 2

Estado del arte

Este capítulo expone varios conceptos que es conveniente conocer para situar esta investigación en un contexto adecuado, así como el estado del arte previo.

El primer concepto que se explicará será la teoría de los sistemas complejos. Estos sistemas se caracterizan por estar compuestos por un gran número de objetos individuales que interaccionan entre sí, y es un objetivo principal de esta teoría el investigar cómo comportamientos globales surgen de reglas sencillas seguidas por los individuos a partir de información limitada o local.

A continuación se pasará a hacer una revisión del estado del arte en los sistemas de redes de pares. En estos sistemas de computación distribuidos, y a diferencia de los entornos tradicionales de computación paralela, se tiende a ejercer un escaso control sobre los componentes de la red y a admitir participantes durante un tiempo indeterminado. El análisis de estos sistemas mediante la teoría de sistemas complejos otorga la posibilidad de comprender su comportamiento y da una idea del potencial de estas redes.

Por último se expondrá la materialización de estas ideas en el algoritmo newscast, la biblioteca DRM y el proyecto DR-EA-M. Esta sección hace las funciones de introducción a los capítulos siguientes, donde se expondrán los detalles de esta investigación.

2.1. Sistemas complejos

El estudio de los sistemas adaptativos complejos trata de comprender cómo comportamientos globales surgen de interacciones locales entre un gran número de agentes autónomos simples. Un sistema complejo es por su propia naturaleza impredecible, pero proporciona la capacidad de comprender *a posteriori* los resultados de las interacciones entre grupos infinitamente grandes de agentes y de cambios infinitesimalmente pequeños en ellos. Mediante el uso de simulaciones computacionalmente intensivas es posible aislar las causas concretas de los efectos percibidos y por tanto controlar sistemas anteriormente percibidos como caóticos e ingobernables [3].

Los sistemas complejos aparecen en un amplio rango de contextos incluyendo las ciencias físicas, sociales y biológicas y han sido utilizados para estudiar la evolución biológica, el cerebro, los sistemas inmunológicos de los mamíferos, colonias de insectos, mercados financieros, redes sociales y redes tecnológicas tales como la red eléctrica, Internet y la World Wide Web.

Tradicionalemente, la ingeniería se ha esforzado por mantener sus sistemas lineales, porque esto hace que sean más simples de construir y predecir. Aún así, muchos sistemas físicos son inherentemente sistemas complejos según la definición anterior, y la ingeniería práctica debe incluir elementos de investigación adecuados a su naturaleza.

Sin embargo, la confusión existente en la definición de la teoría de sistemas complejos y la ausencia de herramientas adecuadas para su estudio impiden un mayor avance en diversos campos importantes para esta investigación como los sistemas de redes de pares, de los que hablaremos más adelante.

2.1.1. Orígenes e historia

Los sistemas complejos son una nueva aproximación a la ciencia que estudia cómo las relaciones entre partes producen comportamientos colectivos en un sistema y cómo el sistema interactúa y forma relaciones con su entorno [17].

Los origenes de la teoría de la complejidad se pueden trazar hasta Adam Smith, creador de la idea del libre mercado. Adam Smith fue un filósofo y economista

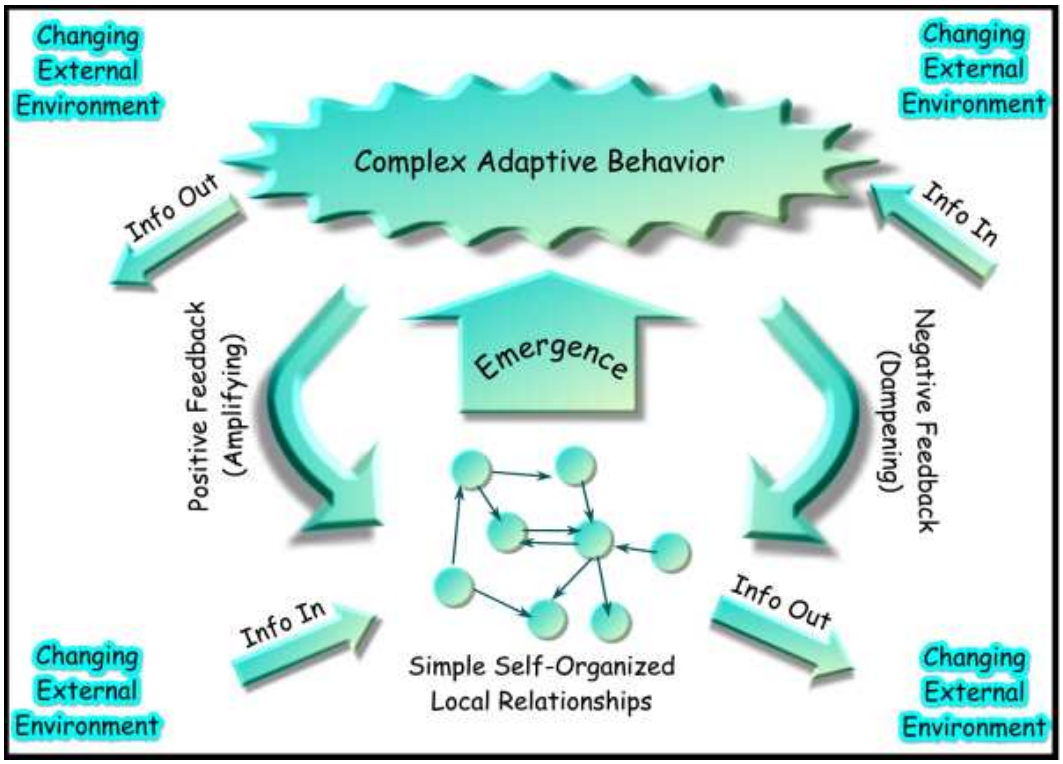

Figura 2.1: Representación de un Sistema Complejo.

perteneciente a la ilustración escocesa del siglo XVIII. Escribió el tratado "An Inquiry into the Nature and Causes of the Wealth of Nations", que fue uno de los primeros intentos de estudio sistemático del desarrollo histórico de la industria y el comercio en Europa. Este tratado introduce la idea del libre mercado, donde un gran número de agentes, trabajando cada uno para sus propios intereses, obtiene en el sistema un comportamiento emergente beneficioso para todos.

Otra rama científica que terminaría provocando el alumbramiento de los sistemas complejos sería la biología evolutiva, de la mano de Charles Darwin. En el siglo XIX, y consciente de los trabajos de Adam Smith, Charles Darwin expuso la evolución de las especies como un proceso en el cual los seres vivos competían entre sí en gran número, y de esta competición por los recursos emergería una población de individuos cada vez más adaptados al medio natural en que viven.

Basándose en esto y desde el siglo XIX a principios del XX, la escuela austríaca desarrolló el problema del cálculo económico y el concepto de conocimiento disperso, que fueron utilizados en los debates contra el keynesianismo dominante en aquella

época. Este debate llevaría a economistas y políticos entre otros a explorar la cuestión de la complejidad computacional.

Un pionero en el campo, e inspirado por los trabajos de Karl Popper y Warren Weaver, el Nobel de economía y filósofo Friedrich Hayek dedicó gran parte de su trabajo, durante el siglo XX, al estudio de los fenómenos complejos, sin limitar su trabajo a la economía sino explorando también otros campos como la psicología, la biología y la cibernética.

Sin embargo, el estudio de los sistemas complejos se dispararía con el incremento en capacidad computacional barata y accesible entre los años 70 y 90 del siglo XX. El campo de estudio conocido como la Teoría del Caos mostraría los resultados que se podían obtener mediante simulaciones en computadoras, siendo un excelente ejemplo de esta etapa la recreación por parte de Lorenz de un sistema climatico dentro de su computadora, mediante cinco reglas simples que interactuaban entre sí.

En 1984 científicos procedentes del Los Alamos National Laboratory fundaron el Santa Fe institute, donde han trabajado investigadores como Murray Gell-Mann, John H. Holland, John D. Farmer y Nick Metropolis. Su misión original era diseminar la noción de un área independiente e interdisciplinar de investigación, la teoría de la complejidad. Recientemente el instituto ha anunciado que su misión original ha sido realizada, haciendo notar que han sido fundados numerosos institutos y departamentos dedicados al estudio de la complejidad.

2.1.2. Características de los sistemas complejos

Los sistemas complejos en la naturaleza tienen las siguientes características [26].

Las relaciones no son lineales

En términos prácticos, esto significa que una pequeña perturbación puede causar un gran efecto, un efecto proporcional, o incluso ningún efecto en absoluto. En los sistemas lineales, el efecto es siempre proporcional a la causa.

Las relaciones contienen ciclos de realimentación

Se encuentra realimentación tanto positiva como negativa a menudo en los sistemas complejos. Los efectos del comportamiento de un elemento son realimentados al sistema de tal modo que el elemento en sí mismo es alterado.

Los sistemas complejos son abiertos

Los sistemas complejos en la naturaleza son sistemas abiertos, existen en un gradiente termodinámico y disipan energía. Este flujo energético es empleado para contrarrestar la segunda ley de la termodinámica y crear orden a partir del caos.

Los sistemas complejos tienen memoria

La historia de un sistema complejo puede ser importante. Dado que los sistemas complejos son sistemas dinámicos, cambian con el tiempo, y estados previos pueden tener una influencia sobre estados posteriores.

Los sistemas complejos pueden anidarse

Los componentes de un sistema complejo pueden a su vez ser sistemas complejos. Por ejemplo, una economía está compuesta de organizaciones, que están compuestas de personas, que están compuestas de células - todos estos elementos son sistemas complejos.

Los límites son difíciles de determinar

Puede ser díficil determinar los límites de un sistema complejo. La decisión final depende del observador.

Red dinámica de multiplicidad

Al igual que las reglas de acoplamiento, la red dinámica de un sistema complejo es importante. Redes de mundo pequeño o de escala libre que poseen muchas interacciones locales y un número más reducido de interacciones entre áreas se utilizan a menudo. Los sistemas complejos naturales a menudo exhiben estas topologías. En el cortex humano, por ejemplo, se pueden observar una conectividad local muy densa y unos pocos axones que llegan a distancias muy alejadas entre regiones dentro del cortex y hacia otras regiones del cerebro.

Se pueden producir efectos emergentes

Los sistemas complejos pueden exhibir comportamientos emergentes, lo que significa que aunque los resultados pueden ser deterministas, también pueden provocar efectos que sólo pueden ser estudiados a un mayor nivel de abstracción. Por ejemplo, las hormigas en una colonia tienen una fisiología, bioquímica y biología que se encuentran en un nivel de análisis, pero su comportamiento social y constructivo es una propiedad que emerge de la colección de hormigas y debe ser estudiada a un nivel distinto.

2.2. Computación en redes de pares

Los sistemas complejos se caracterizan por estar compuestos de un gran número de entidades relativamente simples que interactúan mediante reglas sencillas que utilizan conocimiento local. De estos sistemas pueden emerger estructuras globales que permiten utilizar el conjunto de entidades simples con un proposito común sin necesidad de un coordinador de la actividad. Es por tanto prometedor el identificar los sistemas de computación en redes de pares como sistemas complejos.

Los sistemas de computación en redes de pares son la base sobre la que se conciben sistemas como Gnutella [27], SETI@Home [39], OceanStore [38] y muchos otros. Estos sistemas se caracterizan generalmente por la compartición directa de recursos computacionales (ciclos de CPU, almacenamiento, contenidos) en lugar de requerir la intermediación de un servidor centralizado.

La motivación para basar aplicaciones en arquitecturas de redes de pares se deriva principalmente de su habilidad para funcionar, crecer, y auto-organizarse en presencia de una población altamente transitoria de nodos, conexiones y fallos computacionales, sin la necesidad de un servidor central y la sobrecarga de su administración. Estas arquitecturas típicamente tienen como características inherentes la escalabilidad, resistencia a la censura y control centralizado, y un acceso incrementado a recursos.

Administración, mantenibilidad, responsabilidad por la operación, e incluso la noción de "propiedad" de sistemas peer-to-peer también son distribuidos entre los

usuarios, en lugar de ser manejados por una única compañía, institución o persona. Finalmente, las arquitecturas peer-to-peer tienen el potencial de acelerar los procesos de comunicación y reducir los costes de colaboración a través de la administración ad hoc de grupos de trabajo [1].

2.2.1. Definición de sistemas P2P

Un rápido vistazo a la literatura existente revela un considerable número de diferentes definiciones de sistemas "peer-to-peer", que se distinguen principalmente por el rigor con el que emplean el término.

Las definiciones más estrictas de sistemas "peer-to-peer puros" se refieren a sistemas totalmente distribuidos, en los cuales todos los nodos son completamente equivalentes en términos de funcionalidad y de las tareas que realizan.

Según Shirky [35] "peer-to-peer es una clase de aplicaciones que aprovechan recursos (por ejemplo almacenamiento, ciclos de CPU, contenidos, presencia humana) disponibles en los bordes de Internet". Aunque es un buen inicio, es justo decir que no hay un acuerdo general acerca de que "es" y que "no es" peer-to-peer. El artículo en sí es una buena exposición de las dificultades para definir este concepto y las razones de estas dificultades.

Pero la definición más precisa y la que tomaremos como guía en este documento, es la presentada en [1]:

Los sistemas peer-to-peer son sistemas distribuidos consistentes en nodos interconectados capaces de auto-organizarse en topologías de red con el propósito de compartir recursos tales como contenido, ciclos de CPU, almacenamiento o ancho de banda, capaces de adaptarse a fallos y acomodando poblaciones transitorias de nodos al tiempo que se mantienen una conectividad y eficiencia aceptables, sin requerir la intermediación o apoyo de una autoridad o servidor centralizado global.

Esta definición está pensada para acomodar "grados de centralización" que van desde los sistemas puros, completamente descentralizados, como Gnutella [27], a sistemas "parcialmente centralizados" como KaAzA [44].

2.2.2. Clasificación de sistemas P2P

La ejecución de cualquier sistema de distribución de contenidos en redes de pares depende de una red de computadoras (nodos), y conexiones entre ellos (aristas). Esta red está formada sobre -e independientemente de- la red subyacente de computadoras (típicamente IP) y por tanto se denomina una red superpuesta. La topología, estructura, grado de centralización y los mecanismos de enrutado y búsqueda de información que emplea para mensajes y contenido son cruciales para el funcionamiento del sistema, ya que afectan a su tolerancia a fallos, efectividad, escalabilidad y seguridad [1].

Las redes superpuestas pueden ser distinguidas por su centralización y estructura. Como ya se ha comentado, aunque en su forma más pura se supone que las redes de pares superpuestas son totalmente descentralizadas, en la práctica esto no es siempre cierto, y se pueden encontras sistemas con diversos grados de centralización. Específicamente se pueden ver las tres siguientes categorías:

Arquitecturas puramente descentralizadas. Todos los nodos en la red realizan exactamente las mismas tareas, actuando tanto como servidores y clientes sin coordinación central de sus actividades. Los nodos de estas redes son denominadas a menudo "servents" (SERVers + cliENTS).

Arquitecturas parcialmente centralizadas. La base es la misma que con los sistemas puramente descentralizados. Algunos de los nodos, sin embargo, asumen un rol más importante, actuando como índices centrales para recursos compartidos. El modo en el que la red les asigna su rol a estos supernodos varía entre diferentes sistemas.

Arquitecturas descentralizadas híbridas. En estos sistemas existe un servidor central que facilita la interacción entre pares manteniendo directorios donde se describen los archivos compartidos por los pares. Aunque la interacción y los intercambios de archivos puedan tener lugar directamente entre dos pares, los servidores centrales facilitan esta interacción realizando las búsquedas e identificando los nodos que almacenan los archivos. Los términos "peer-through-peer" o "broker-mediated" son a veces utilizados para describir estos sistemas. Obviamente, en estas arquitecturas existe un único punto de fallo en el servidor central. Esto típicamente

los convierte en sistemas inherentemente no escalables y vulnerables a censura, fallo técnico o ataque malicioso.

Las redes de pares también se pueden clasificar según su estructura. Con ello nos referimos a si la red superpuesta se crea indeterminísticamente (ad hoc) cuando se añaden nodos y contenidos, o si su creación se basa en reglas específicas. Según su estructura, se pueden categorizar las redes de pares como sigue:

No estructuradas. La localización de los contenidos (archivos) es completamente independiente de la topología de la red superpuesta. En una red no estructurada el contenido típicamente necesita ser localizado. Los mecanismos de búsqueda van desde métodos de fuerza bruta, tales como inundar la red con peticiones que se propagan en profundidad o anchura hasta que se localiza el contenido deseado, hasta estrategias más sofísticadas y eficientes en el uso de recursos como paseos aleatorios (random walks) e índices de enrutado. Los mecanismos de búsqueda empleados en redes no estructuradas tienen implicaciones obvias, particularmente en lo que se refiere a cuestiones de disponibilidad, escalabilidad y persistencia.

Los sistemas no estructuradas son generalmente más apropiados por acomodar poblaciones de nodos muy inconstantes. Algunos ejemplos representativos de sistemas no estructurados son Napster [14], Publius [40], Gnutella [27] o KaZaA [44].

Estructuradas. Éstas han emergido principalmente en un intento de resolver el problema de la escalabilidad con el que originalmente se enfrentaron los sistemas no estructurados. En las redes estructuradas la topología superpuesta es controlada firmemente y los archivos (o los enlaces a los archivos) son colocados en lugares específicos. Estos sistemas esencialmente proveen un mapeo entre contenido (e.g. identificador de archivo) y su localización (e.g. dirección del nodo), en la forma de una tabla distribuida de enrutado, tal que las peticiones pueden ser enrutadas eficientemente al nodo con el contenido buscado [21].

Los sistemas estructurados ofrecen una solución escalable para peticiones exactas (exact-match), esto es, peticiones donde el identificador exacto del objeto pedido es conocido (en comparación con las búsquedas mediante palabras clave). La utilización de búsquedas exactas como substrato para búsquedas por palabras clave permanece como un problema de investigación aún sin resolver para entornos distribuidos [43].

La desventaja de los sistemas estructurados es que es difícil mantener la estructura requerida para enrutar mensajes eficientemente frente a una población de nodos muy transitoria, en la que los nodos se unen y dejan la red a un ritmo muy rápido.

Ejemplos típicos de sistemas estructurados incluyen Chord [36], [32], [12] o [46].

2.2.3. Aplicaciones P2P

Las arquitecturas peer-to-peer han sido empleadas para varias categorias de aplicaciones, que incluyen las siguientes:

Comunicación y Colaboración. Esta categoría incluye sistemas que proporcionan la infraestructura para facilitar comunicación y colaboración directa entre computadoras, normalmente en tiempo real. Los ejemplos incluyen aplicaciones de chat, mensajería instantánea y telefonía, como ICQ [25], MSN [6], Jabber [15] o Skype [45].

Computación Distribuida. Esta categoría incluye sistemas cuyo propósito es utilizar la potencia de procesamiento de los pares. La aplicación más conocida de este tipo es Seti@home [39].

Soporte a Servicios de Internet. Ejemplos de este tipo de aplicaciones serían sistemas multicast, infraestructuras de indirección o aplicaciones de seguridad.

Distribución de Contenidos. La mayoria de los sistemas peer-to-peer actuales son diseñados con este objetivo en mente. Para compartir contenidos entre pares se ha realizado un gran esfuerzo de investigación y desarrollo que ha producido sistemas e infraestructuras que cubren un amplio rango de características según los medios que aportan para publicar, indexar, localizar, transferir o asegurar los contenidos o las transferencias. Ejemplos destacados son Napster [14], Gnutella [27], Freenet [4] o BitTorrent [5].

2.3. El Proyecto DREAM

DREAM proporciona un marco para la producción de sistemas basados en algoritmos evolutivos y sistemas de agentes que utilizan Internet para permitir procesamiento distribuido en redes de pares.

2.3.1. Ecosistemas de Información Universales

El programa "Future and Emerging Technologies" de la Unión Europea fue ideado para proporcionar recursos a ideas científicas novedosas y emergentes. Su misión es promover la investigación a largo plazo o que implica altos riesgos, compensados por el potencial de un impacto industrial o social significativo. Una de las iniciativas de este programa son los Ecosistemas de Información Universales (UIE) [16].

Las tendencias actuales muestran que, en el futuro, la cantidad de actividad social y económica que tendrá lugar a través de las infraestructuras globales de información será tan grande que no sólo habrá una dependencia de estas infraestructuras, sino también una enorme demanda de su efectividad y eficiencia en variadas e impredecibles situaciones aún desconocidas. Paradójicamente, a pesar de que la potencia de esta infraestructura en gran medida deriva de la cantidad de información y del rango de actividades que incluye, este mismo tamaño y complejidad la hará muy dificil de aprovechar para las necesidades específicas de individuos particulares u organizaciones. Este problema sólo puede agravarse mientras el entorno continúa creciendo en tamaño y en alcance.

Cosechar todos los beneficios potenciales de una infraestructura de información como ésta requerirá realizar una visión que vaya mas allá de extrapolaciones incrementales de los paradigmas tecnológicos actuales.

La iniciativa proactiva de Ecosistemas Informáticos Universales está dirigida a explorar y validar nuevas tecnologías y escenarios que pueden convertir la compleja infraestructura de información tal como emerge actualmente en un ambiente rico, adaptativo, reactivo y realmente abierto.

Los UIE brotan de la visión de un ecosistema emergente basado en información que constantemente escala hacia lo global o hacia lo individual, que evoluciona y se adapta para acomodarse a las demandas de su vasta y altamente dinámica población de "infohabitantes", de los que no es irracional pensar que sean trillones. El beneficio sería un entorno que ayuda a la creación dinámica de nuevos tipos de relaciones y actividades y que, por medio de ellas, crea valores y grados de escalabilidad, sostenibilidad y robustez que están más allá de lo que puede ser imaginado hoy día.

Algunos ejemplos de las características principales que podrían darse en este

ecosistema de información serían:

- Apertura y Universalidad: el acomodo sin trabas de nuevos tipos de infohabitantes, dominios de discurso y actividad y la evolución de los existentes
- Escalabilidad: la habilidad del ecosistema para escalar hacia lo global o hacia lo individual de acuerdo a las siempre cambiantes necesidades de una población dinámica de infohabitantes
- Oportunidad y Relevancia: el ecosistema se comporta de un modo que, en cualquier momento, todos y cada uno de los infohabitantes son conscientes de aquellas oportunidades del entorno que son relevantes a sus objetivos y actividades.
- Adaptabilidad a condiciones cambiantes: los ecosistemas implican reacción y ajuste continuo a condiciones cambiantes, por ejemplo a través de la creación, eliminación, evolución, migración, recombinación o reorganización de infohabitantes de un modo descentralizado mediante la activación de mecanismos de auto-estabilización.
- Realización de objetivos e intenciones: el entorno permitirá que el conocimiento y las capacidades de cada infohabitante sean mejoradas y recombinadas dinámicamente con las de otros de un modo tanto efectivo como eficiente para satisfacer los objetivos e intenciones de los individuos, grupos u organizaciones para los que operan.

Para llegar a esta visión se requerirá una reestructuración radical y el entendimiento de un rango de áreas relevantes de investigación. El éxito depende de una perspectiva amplia e interdisciplinar, acumulando experiencia de disciplinas tan diversas como por ejemplo biología, sistemas distribuidos, ingeniería del software, lógica computacional, inteligencia artificial y cibernética, así como economía, teoría organizacional y sociología. Rompiendo fronteras tradicionales, la iniciativa también espera estimular la creación de una comunidad investigadora creadora de tendencias que guíe a Europa hacia el liderazgo de este area multidisciplinar.

2.3.2. El proyecto DREAM

El proyecto DREAM [31] nació para implementar un Ecosistema Universal de Información. La intención fue ir más allá del estado del arte proporcionando las siguientes características dentro de un único sistema [30].

- Una herramienta que es un marco de desarrollo en el que crear instancias de aplicaciones, en lugar de tener los modelos o problemas directamente codificados en su interior.
- Una herramienta diseñada para permitir tanto soluciones a problemas de optimización industrial como el modelado del comportamiento de sistemas de gran tamaño.
- Una herramienta que permite la libre migración de infohabitantes a través de Internet, permitiendo la formación de diversos nichos de actividad.
- Una herramienta que permite el uso de ciclos sueltos de CPU de un modo automatizado y seguro.
- Una herramienta que permite observar comportamientos a un nivel macroscópico.
- Una herramienta diseñada para ser escalable y abierta.

Una red DREAM consistiría en un conjunto de infomundos habitados por infohabitantes. Estos infomundos se extenderían sobre un número potencialmente grande de computadoras separadas físicamente, siendo también posible la superposición de varios infomundos sobre una única computadora. Cada infomundo podría ser utilizado tanto para solucionar un problema como para simular alguna situación.

En el primer caso, una población evolutiva de infohabitantes sería capaz de atacar un problema de un modo adaptativo. Infohabitantes individuales o subpoblaciones podrían competir, esta competencia propiciaría mejoras en la calidad de cada uno. Usando principios darwinianos en simulaciones computerizadas, se pueden evolucionar soluciones de alta calidad a problemas complejos.

De un modo complementario a esta evolución existe también la posibilidad de cooperación o negociación entre infohabitantes individuales o subpoblaciones. Se espera que esto proporcione una inteligencia colectiva que opera de un modo efectivo dividiendo el problema y permitiendo a los infohabitantes generar una solución conjuntamente.

DREAM proporciona un entorno virtual que por su complexidad y variedad intrínsecas imita entornos reales mejor que los sistemas utilizados en la actualidad. DREAM podría ser utilizado para simular aspectos de las sociedades en el mundo real e investigar como las interacciones a un nivel microscópico pueden crear propiedades emergentes a nivel macroscópico. Se espera lograr un entendimiento no trivial en el campo de los sistemas sociales complejos.

2.3.3. Implementación de DREAM

La arquitectura de sistema se divide en cinco capas [2], donde cada una proporciona una interfaz de usuario a un nivel diferente de abstracción e interacción. La arquitectura se muestra en la figura 2.2 junto con los puntos de entrada para los usuarios. Los puntos de entrada proporcionan una variedad de interfaces, con diferentes niveles de facilidad de uso y potencia. Los niveles superiores son más fáciles de usar, pero la flexibilidad es más limitada. Los niveles inferiores requiren un mayor conocimiento, pero dan un mayor control sobre el sistema. Los cinco tipos de usuario se pueden categorizar como sigue:

- El usuario A no desea usar DREAM para realizar experimentos, pero desea donar sus ciclos sobrantes de CPU o monitorizar los experimentos de otros. Este tipo de usuario interactúa con DREAM sólo a través de la Consola.

- El usuario B o no es un programador experto o desea hacer un prototipo rápido sin la necesidad de programar explícitamente. Este usuario interactúa a través de la capa GUIDE que permite definir algoritmos evolutivos mediante una interfaz gráfica. La capa GUIDE se comunica con la capa EASEA mediante el lenguaje EASEA.

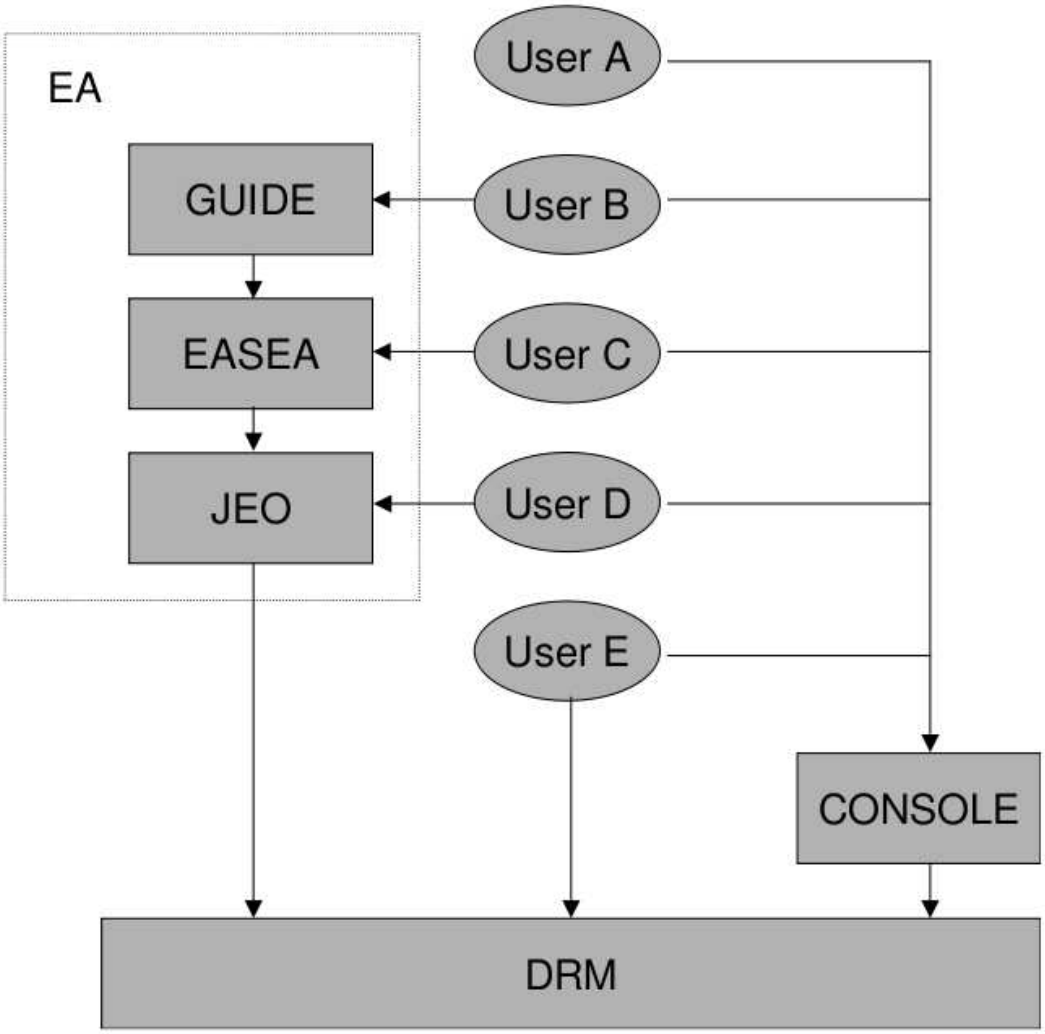

Figura 2.2: Arquitectura en capas de DR-EA-M.

- El usuario C programa el sistema mediante la capa EASEA, que proporciona un lenguaje de alto nivel para programar algoritmos evolutivos. Esta capa produce código Java que utiliza la biblioteca JEO mediante un compilador.
- El usuario D programa directamente en Java y utiliza la biblioteca JEO, que proporciona objetos y métodos para computación evolutiva y una API al núcleo DRM (Distributed Resource Machine).
- El usuario E es un usuario experto, que programa directamente utilizando la API de DRM. En este nivel se puede usar DRM para otros propósitos de procesamiento distribuido más allá de la computación evolutiva.

El grado de desarrollo de DRM y JEO es casi estable, aunque necesitado de retroalimentación por parte de los usuarios y varias evoluciones de código. EASEA, GUIDE y la Consola se encuentran en estados anteriores de desarrollo.

2.3.4. La biblioteca DRM

Una red DRM [24] es un conjunto distribuido de programas que pueden ejecutarse en distintas máquinas. Estos programas pueden ser dividido entre bases y agentes. Las bases se ejecutan asociadas a una máquina y mantienen enlaces de comunicación entre sí. Las bases pueden alojar agentes. Un agente es un programa móvil que realiza una tarea. Los agentes se alojan temporalmente en una base, siendo capaces de cambiar de base cuando lo deseen. Los agentes se pueden comunicar con el exterior de sus bases utilizando los servicios de comunicación que éstas ofrecen. La biblioteca DRM y las aplicaciones distribuidas con ella sirven para construir una red DRM y desarrollar agentes que la utilicen.

En un entorno tradicional con una única máquina una aplicación se compone de uno o más hilos de ejecución que son manejados por un sistema operativo (SO). El SO controla los hilos, les asigna recursos y se encarga de los diversos aspectos de seguridad. Cuando se adapta este esquema a entornos distribuidos a gran escala, no todos los aspectos pueden ser implementados exactamente del mismo modo debido a los relativamente altos costes de los intercambios de información.

Una característica clave del modelo DRM es que está pensado como un conjunto de agentes autónomos cooperantes. Un agente es análogo a un hilo de ejecución en un SO: una aplicación se ejecuta lanzando uno o más agentes que se pueden comunicar entre sí, que pueden tomar decisiones basandose en la información del sistema que los aloja o en la información que poseen de otros agentes. Los agentes pueden asimismo lanzar ellos mismos nuevos agentes. La DRM controla los agentes, los recursos a los que tienen acceso, la seguridad y otros aspectos que fueran relevantes a su nivel de ejecución.

Sin embargo estos agentes tienen mucha más libertad: son también móviles, pueden cambiar su localización física mientras realizan sus labores de computación y pueden continuar sus tareas en sus nuevas localizaciones.

El problema de la movilidad y de la seguridad fue solucionado escogiendo el lenguaje Java para implementar la DRM. Este entorno ofrece una solución natural para mover código ejecutable así como datos entre agentes y ofrece un amplio conjunto de características de seguridad.

Las aplicaciones que se tienen en mente como adecuadas para DRM son bastante especiales. Recursos habituales como memoria compartida son un lujo en el mundo de los sistemas distribuidos grandes. Otro problema es la naturaleza inestable tanto de los canales de comunicación entre los nodos y de los nodos en sí mismos. Esto restringe el área de aplicaciones a tareas que son robustas (que no se ven afectadas por perder un subconjunto de los agentes que las realizan) y masivamente paralelas (no se necesita mucha comunicación entre los agentes).

Una decisión de implementación importante fue que no se utilizarían servidores centrales. Esto se hizo para maximizar la escalabilidad y la robustez. Con esta restricción incluso mantener la conectividad de la red se convierte en un difícil desafío. Este problema y también el problema de la distribución de información en la DRM se soluciona utilizando un protocolo epidémico [10] con el nombre de algoritmo newscast [20].

2.3.5. El algoritmo newscast

La capa de comunicaciones de una red DRM se basa en su mayor parte en el algoritmo newscast, que se estudiará con detalle en el capítulo 3. Varios actores de una red DRM utilizan este algoritmo para construir una topología aleatoria entre ellos y difundir información de un modo eficiente.

El conocimiento previo de este algoritmo se encuentra recogido en un informe técnico de Jelasity y van Steen para la Vrije Universiteit [20], con fragmentos publicados como artículos revisados [23, 24, 33] y en otro informe técnico posterior centrado en la difusión de información [19]. El algoritmo newscast ha sido implementado en la biblioteca DRM, como parte del proyecto DR-EA-M [2, 31].

En [20] se describe el algoritmo newscast como un algoritmo de diseminación probabilística de información mediante imitación de patrones epidemiológicos. Cada nodo que ejecuta una instancia del algoritmo posee una vista local de la red, con conocimiento de un número limitado de otros nodos. A intervalos regulares cada nodo se comunica con uno de sus vecinos y ambos crean nuevas vistas locales según un método probabilistico a partir del conocimiento conjunto de ambos. En este informe también se demuestran varias propiedades interesantes del algoritmo:

La mayor parte de las aristas de la red creada conforman un grafo aleatorio.

El número de comunicaciones recibidas por un nodo se conforma a un distribución de Poisson de media 1.

La red es robusta, es necesario eliminar un gran número de nodos para dividirla en dos.

La información emitida por cualquier nodo llega en un tiempo finito a todos los demás.

La red creada presenta un crecimiento de los caminos entre los nodos logarítmico respecto al número de estos.

La red creada presenta un coeficiente de agrupamiento mayor que el de un grafo aleatorio.

En conjunto el algoritmo está definido en la documentación enunciada con anterioridad, e implementado en Java en la biblioteca DRM. El trabajo de investigación presentado enesta tesis aporta varios elementos ausentes como una definición detallada del algoritmo, una explicación de la topología creada, documentación para comprender la biblioteca que lo implementa y poder desarrollar basándose en ella y pruebas de que el algoritmo implementado se conforma a las simulaciones previas llevadas a cabo. Todo esto se presenta en los capítulos siguientes.

Capítulo 3

El algoritmo newscast

El algoritmo newscast fue desarrollado como una herramienta peer-to-peer para diseminar información en redes de gran tamaño como Internet. La mayor parte de la información previa sobre este algoritmo se encuentra recogida en los trabajos de Jelasity *et al.* [20] y de otros varios [19, 23, 33]. Este capítulo complementa estos artículos con la definición detallada del algoritmo y una explicación de la topología de red que construye, accesibles por separado por el autor en [8, 9].

3.1. Definición detallada

El objetivo de esta sección es ofrecer una definición detallada del algoritmo para facilitar que todos los trabajos e implementaciones relacionadas con el mismo sean fácilmente comparables.

3.1.1. Introducción

Newscast se basa en los patrones de difusión de epidemias biológicas para diseminar la información [10, 29]. En los modelos probabilísticos de epidemias reales, cada individuo afectado contagiará la epidemia a un número de individuos según una distribución probabilística. Por tanto la enfermedad (o la información) se expande de un modo estocástico, infectando a la población entera muy eficientemente y sin un control centralizado.

El modo exacto en el que el algoritmo imita este comportamiento es mezclando

dinámica y aleatoriamente las conexiones de la red. Cada conexión o arista es una estructura que contiene el momento de creación de la conexión y la dirección de su creador, también contiene un elemento de información denominado noticia o news item cuyo contenido es irrelevante para el funcionamiento de la red pero que puede ser utilizado para aplicaciones particulares. En analogía con las epidemias biológicas, cada una de estas estructuras es un virus, que se extiende sobre la población de nodos difundiendo las direcciones de los nodos de la red y la información que cada uno de ellos desea comunicar.

Las estructuras de datos utilizadas se detallarán a continuación, para posteriormente mostrar el funcionamiento del algoritmo. Para una explicación resumida pueden consultarse las tablas 3.1 y 3.2.

- A_i: Un *nodo* es una instancia en ejecución del algoritmo. Cuando discutamos las propiedades de la red en términos de teoría de grafos serán llamados *vértices*.

- c_i: Una *contribución* es una estructura de tres elementos que contiene:

  - El *nombre* (*name*) único del agente que la creó, el cual es suficiente para localizarlo en la red.
  - El *momento de creación* (*timestamp*), el instante local en el cual se creó la contribución.
  - De modo opcional, una *noticia* (*news item*), cuyo propósito depende de la aplicación que se quiera construir utilizando este algoritmo.

  Respecto de la red creada, una contribución es equivalente a un *arco* desde el nodo que lo tiene hasta el nodo que lo creó.

- C_i: Una *caché* es un conjunto que puede contener hasta un número máximo de contribuciones, llamado $\hat{C}$. Cada nodo tiene una caché, que define los nodos con los que puede comunicarse y la información que posee de ellos.

- Δt_v: La *tasa de refresco* o una *iteración* es el periodo fijo de tiempo que un nodo espera desde su última comunicación de salida hasta la siguiente.

Pseudocódigo:

```
procedure newscast():
    loop
        e = event()
        if e is timeout then
            peer.name = choose_peer()
            send_node_data(peer.name)
            peer = receive_node_data(peer.name)
            me.cache = merge(me,peer)
        end if
        if e is receive_merging(peer.name) then
            send_node_data(peer.name)
            peer = receive_node_data(peer.name)
            me.cache = merge(me,peer)
        end if
    end loop


procedure merge(in: me, peer) out: C+
    C+ = me.cache + peer.cache
    clean_cache(C+, me.name, peer.name)
    remove_old(C+)
    cut_to_size(C+)
    C+ = C+ + peer.contribution
    return C+
```

Cuadro 3.1: Descripción mediante pseudo-código del algoritmo newscast.

Estructuras de datos:

me: Información del nodo local, es una estructura que contiene

name: el nombre del nodo,
contribution: la contribución del nodo,
cache: un conjunto de contribuciones.

peer: Información de un par, tiene la misma estructura que *me*.

Eventos:

timeout: Han pasado Δt_i segundos desde la última mezcla (*merging*).

receive_merging(n): Un nodo n quiere mezclar su caché con la del nodo local.

Funciones:

send_node_data(n): Envía la información del nodo local a un nodo n.

receive_node_data(n): Recibe la información de un nodo n.

choose_peer(): Elige una contribución aleatoria de la caché y devuelve el nombre de su propietario.

clean_cache(C, n_1, n_2): Elimina todas las contribuciones de los nodos n_1 o n_2 de la caché C.

remove_old(C): Busca contribuciones duplicadas en C y elimina las más antiguas.

cut_to_size(C): Elimina contribuciones aleatorias de C hasta que su talla sea menor que $\hat{C}$.

continue: Ignora el resto de la iteración.

$+$: Concatena conjuntos y/o elementos.

Cuadro 3.2: Estructuras de datos utilizadas por el algoritmo newscast.

Para explicar el funcionamiento del algoritmo obviaremos por ahora cómo se inicia y nos situaremos en un punto intermedio cualquiera de su ejecución. Tenemos una red asíncrona, con nodos en hilos de ejecución distintos y posiblemente distribuidos entre varias computadoras. Cada nodo tiene una caché con contribuciones de hasta otros $\hat{C}$ nodos. Empezaremos analizando el comportamiento de un único nodo, que llamaremos *me*.

Tras esperar durante Δt_{me} segundos desde su última operación, *me* le envía su información al propietario de una contribución escogida al azar de su caché. El nodo que la recibe le corresponde enviando su propia información (*peer*).

Como se puede observar en la figura 3.1, *me* concatena *me.cache* y *peer.cache* en una única estructura que llamaremos C_+. A continuación elimina de C_+ cualquier contribución suya o de *peer*. Después *me* busca contribuciones duplicadas en C_+ y resuelve la duplicación eliminando las más antiguas. Una contribución se dice que es un duplicado de otra si ambas han sido creadas por el mismo nodo, aunque su momento de creación o su contenido sean distintos.

A continuación se eliminan contribuciones de C_+ hasta que su tamaño es menor que $\hat{C}$ y después se añade *peer.contribution*. En este punto C_+ ya es una caché válida y *me* la utiliza para sustituir *me.cache*.

El proceso no es muy distinto desde el punto de vista de *peer*. Éste inesperadamente recibe una comunicación desde *me*, que le envía su información. *Peer* responde inmediatamente enviando la suya. A continuación *peer* puede terminar cualquier acción que estuviera realizando (por ejemplo otra operación de mezcla) y cuando esté libre puede procesar la información enviada por *me* como si hubiera sido *peer* quién hubiera iniciado la transacción.

Veamos que sucede al principio de la ejecución. A un nodo newscast habitualmente se le proporciona el nombre (y por lo tanto la localización) de otro nodo cualquiera para que pueda conectarse a la red, como gnutella [27] y otros sistemas puros de redes de pares [37]. Este nodo recién iniciado tratará de mezclar su caché con el nodo que se le ha proporcionado de manera externa. Es trivial observar que su número de conexiones a otros nodos crecerá rápidamente hasta llegar a $\hat{C}$. Si a un nodo no se le proporciona una conexión inicial queda a merced de que algún otro

nodo, o un servicio especialmente diseñado para buscarlo, tenga conocimiento de él y lo conecte a la red.

Dado que las contribuciones son al mismo tiempo enlaces hacia un nodo y la información que éste difunde, esta mezcla de cachés por parte de los nodos tiene una doble función. Por una parte crea una topología aleatoria auto-organizativa que se recupera fácilmente de la posible eliminación de nodos, y por otra difunde la información de cada uno de los nodos con una gran eficiencia.

Modelo mediante Autómatas de Entrada / Salida

Los modelos de autómatas de entrada/salida [22] se utilizan para demostrar propiedades de sistemas distribuidos en entornos definidos por el creador del modelo. Al requerir del entorno que cumpla con unas condiciones específicas, podemos aceptar un comportamiento errático de los sistemas modelados cuando estas condiciones no se cumplen. De este modo podemos dividir el sistema en capas o módulos y demostrar algunas propiedades individualmente antes de continuar a sistemas más grandes y complejos.

Según explica la página web del Grupo de Teoría de Sistemas Distribuidos del MIT [28], "el modelo de autómatas de entrada/salida, desarrollado por Lynch y Tuttle, es un modelo transicional etiquetado para componentes de sistemas concurrentes asíncronos. Las acciones de un autómata de entrada/salida se clasifican en acciones de entrada, de salida e internas, donde se requiere que las acciones de entrada estén siempre habilitadas. Un autómata de entrada/salida tiene "tareas"; en una ejecución justa de un autómata de entrada/salida, se requiere que todas las tareas obtengan turnos infinitamente a menudo. El comportamiento de un autómata se puede describir en términos de trazas, o alternativamente en términos de trazas justas. Ambos tipos de nociones de comportamiento son composicionales".

Para el algoritmo newscast definimos que el comportamiento del sistema es que la red no tiene errores de transmisión. De este modo vamos a demostrar que el sistema distribuido creado por un número indefinido de agentes newscast nunca sufre problemas de interbloqueo.

La descripción detallada de los estados internos del algoritmo servirá de base

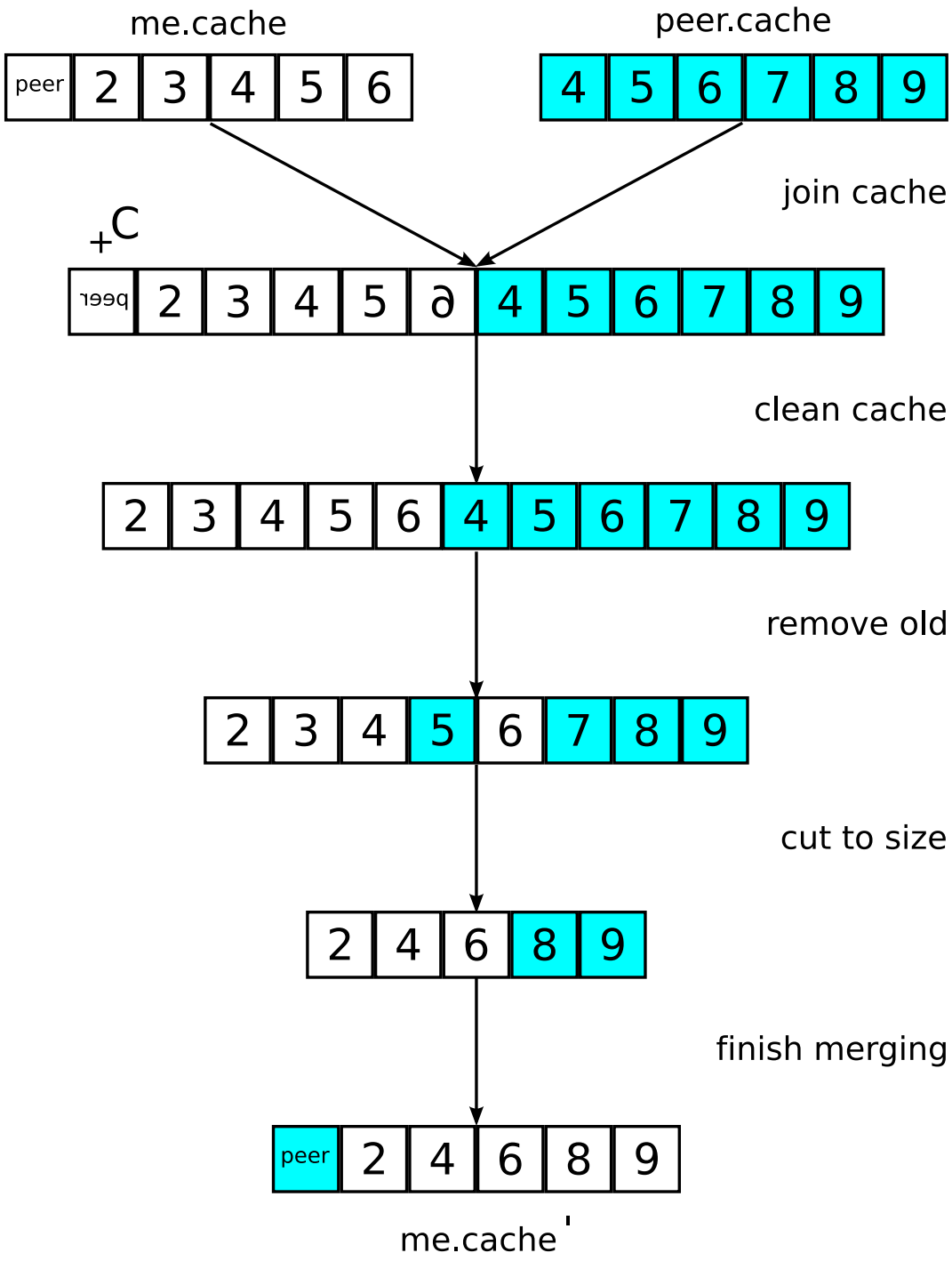

Figura 3.1: Operación de mezcla en el algoritmo newscast.

para implementaciones que exhiban este comportamiento.

Signatura:

(in) start_timeout

(in) receive_merging

(int) accept_merging

(out) propose_merging

(int) unqueue_peer_data

(int) join_cache

(int) clean_cache

(int) remove_old

(int) cut_to_size

(int) finish_merging

Estados:

peer: objeto para almacenar el nombre, caché y contribución del agente no local.

my: objeto para almacenar el nombre, caché y contribución del agente local. Puede ser considerado constante.

cache: objeto temporal para almacenar el futuro me.cache.

timeout_warning: si es true, una comunicación de salida espera ser efectuada.

pretending_peers: un conjunto que puede contener nombres de pares.

Inicio:

peer = null

my:

$me.name$ = el nombre único del agente local, que no cambia nunca.

$me.cache$ = null

$me.contribution$:

$me.contribution.name$ = me.name

$me.contribution.news$ = null

$me.contribution.timestamp$ = tiempo local en el momento de creación

$cache$ = una contribución de otro agente, proporcionada de algún modo

$timeout\_warning$ = false.

$pretending\_peers = \emptyset$.

Acciones:

start_timeout:

input: timeout

action:

$timeout\_waiting \leftarrow true$

propose_merging:

precondition:

$timeout\_waiting = true$

$peer = null$

action:

$timeout\_waiting \leftarrow false$

$peer.name \leftarrow random(cache).name$

$send\_queue(peer.name) \leftarrow send\_queue(peer.name) \cdot my$

output: merging($peer.name$)

nota: $random(set)$ devuelve una entrada aleatoria del set proporcionado.

receive_merging:

input: merging($name$)

action:

$pretending\_peers \leftarrow pretending\_peers \cdot name$

$send\_queue(name) \leftarrow send\_queue(name) \cdot my$

accept_merging:

precondition:

$timeout\_waiting = false$

$peer = null$

$pretending\_peers = name \cdot pretending\_peers'$

action:

$peer.name \leftarrow name$

$pretending\_peers \leftarrow pretending\_peers'$

unqueue_peer_data:

precondition:

$peer.name \neq null$

$peer.cache = null$

$peer.contribution = null$

$recv\_queue(peer.name) = peer' \cdot recv\_queue(peer.name)'$

action:

$recv\_queue(peer.name) \leftarrow recv\_queue(peer.name)'$

$peer.cache \leftarrow peer'.cache$

$peer.contribution \leftarrow peer'.contribution$

join_cache:

precondition:

$peer.name \neq null$

$peer.cache \neq null$

$peer.contribution \neq null$

$cache = null$

action:

$cache \leftarrow me.cache \cdot peer.cache \cdot \{peer.contribution\}$

clean_cache:

precondition:

$cache \neq null$

$\exists\ c \in cache\ /\ c.name = me.name$

action:

$cache \leftarrow cache - c$

remove_old:

precondition:

$cache \neq null$

$\exists\ c1, c2 \in cache\ /\ c1.name = c2.name$

$\wedge\ c1.timestamp \geq c2.timestamp$

action:

$cache \leftarrow cache - c2$

cut_to_size:

precondition:

$cache \neq null$

$\nexists\ c \in cache\ /\ c.name = me.name$

$\nexists\ c1, c2 \in cache\ /\ c1.name = c2.name$

$\wedge\ c1.timestamp \geq c2.timestamp$

$card(cache) > maximum\_cache\_size$

action:

$cache \leftarrow cache - random(cache)$

finish_merging:

precondition:

$cache \neq null$

$peer.name \neq null$

$peer.cache \neq null$

$peer.contribution \neq null$

$card(cache) \leq max\_cache$

$me.contribution \notin cache$

$\nexists\ c \in cache\ /\ c.name = me.name$

$\nexists\ c1, c2 \in cache\ /\ c1 \neq c2 \wedge c1.name = c2.name$

$\forall c \in cache \Rightarrow c \in me.cache \bigcup peer.cache \bigcup \{peer.contribution\}$

$\forall c1, c2\ /\ c1 \in cache \wedge c2 \in me.cache \wedge c1.name = c2.name$

$\Rightarrow c1.timestamp \geq c2.timestamp$

$\forall c1, c2\ /\ c1 \in cache \wedge c2 \in peer.cache \wedge c1.name = c2.name$

$\Rightarrow c1.timestamp \geq c2.timestamp$

action:

$me.cache \leftarrow cache$

$cache \leftarrow null$

$peer \leftarrow null$

El diagrama en la figura 3.2 puede ayudar a comprender mejor el flujo del algoritmo, aunque no es un sustituto de la explicación detallada. Este diagrama representa sólo parcialmente las implicaciones del modelo y otros comportamientos no observables en el diagrama podrían ser inferidos del modelo detallado.

Por ejemplo, aunque en el diagrama las acciones clean_cache y remove_old son secuenciales, no lo son en el modelo detallado, ya que podrían ser seguidos en cualquier orden, incluso en paralelo. Han sido dibujadas secuencialmente en el diagrama porque de este modo son más fáciles de entender.

En este diagrama los estados se representan como flechas y las acciones como cajas. Es necesario que algunos estados estén habilitados para activar una determinada acción, la cual producirá un distinto conjunto de estados que podrían activar otras acciones. Las etiquetas de los estados que podrían ser inferidas de las demás han sido eliminadas para evitar el exceso de información en el algoritmo.

En un modelo de autómatas de entrada/salida las acciones de salida de un autómata son las acciones de entrada de otro. Comparten el mismo nombre porque son la misma acción aunque sucedan en dos agentes distintos y tengan efectos distintos en cada uno de ellos. Se ha preferido no seguir esta regla etiquetando de un modo diferenciado las partes de entrada y de salida de una acción y conectandolas con una señal con un nombre común. Se puede ver un ejemplo en el diagrama donde la acción de salida *propose_ merging* se conecta a la acción de entrada *receive_ merging* por medio de la señal *merging*.

La señal *timeout* y la acción de entrada *start_ timeout* son iniciadas por el sistema operativo. Esto es más sencillo de representar que un reloj interno en el autómata.

Una acción puede a menudo producir diferentes estados, dependiendo de los estados habilitados en su entrada. He utilizado "diamantes" similares a los objetos "if" de los diagramas de flujo de datos para representar que la acción previa puede habilitar únicamente uno de los estados tras el "diamante". Por supuesto, si sólo uno de los estados está etiquetado el otro es su negación.

3.1.2. Análisis del algoritmo

El modelo de autómatas de entrada/salida se utiliza para demostrar formalmente propiedades de algoritmos y sistemas distribuidos. Estas propiedades se dividen normalmente entre propiedades de *viveza*, que describen estados que siempre serán eventualmente alcanzados a partir de ciertas entradas; y propiedades de *seguridad*, que serán ciertas en cada estado del algoritmo. El modelo permite modularizar y

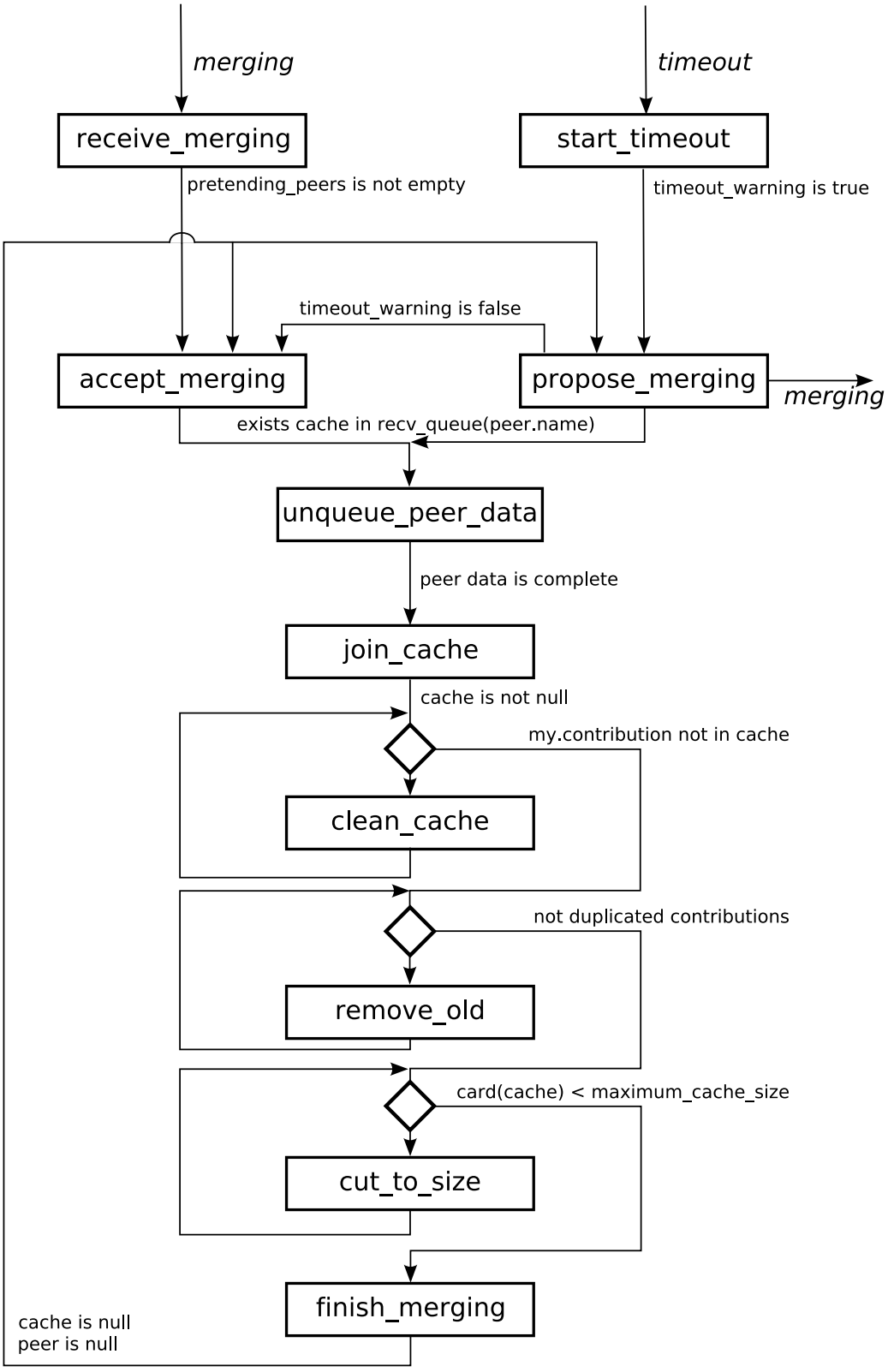

Figura 3.2: Autómata newscast

abstraer fácilmente los autómatas para demostrar estas propiedades, aunque esto no será necesario para demostrar que el algoritmo newscast no se bloquea.

El algoritmo nunca se bloquea

En el algoritmo newscast, cuando se recibe una señal *merging* se envía inmediatamente en respuesta la información del agente receptor (nombre, contribución y caché) por el mismo canal de comunicación. La caché se modifica en una operación atómica al final de cada operación de mezcla, por tanto no existe el riesgo de enviar información incorrecta aunque se esté realizando una operación de mezcla cuando se recibe la comunicación.

La información entrante del par que ha iniciado la comunicación se almacenará en una cola de comunicaciones hasta que se esté en condiciones de procesarla. Para garantizar esto se requiere una mínima lógica computacional que debería estar presente en cualquier máquina con más de un canal de comunicación. En el caso de que la señal recibida sea un *timeout* proveniente del sistema operativo la única reacción es activar un *flag* informativo.

Vista la reacción del algoritmo a una señal de entrada según las restricciones del modelo, podemos enunciar la primera propiedad del algoritmo. En un entorno *razonable* el algoritmo *nunca se bloquea*. Un entorno razonable significa que todas las estructuras de datos están bien formadas y que no se envía información incorrecta a ninguna instancia del algoritmo. Un entorno razonable también significa que la información enviada siempre llega sin errores a su destino, es decir, que no hay fallos de red o que éstos son solucionados por las capas inferiores de la infraestructura de red.

Para demostrar que el algoritmo no se bloquea empezaremos por las dos acciones que compiten por el acceso a la caché: `propose_merging` y `accept_merging`. Según el modelo es preciso demostrar que el algoritmo es *justo*, permitiendo que ambas acciones se lleven a cabo en algún momento de su ejecución.

`propose_merging` depende de los estados $timeout\_warning = true$ y $peer = null$. Después de que se active `timeout_warning` sabemos que si $peer \neq null$ es porque el algoritmo se halla entre los estados `accept_merging` o `propose_merging`

y el estado `finish_merging`. Dado que el entorno nos asegura que los mensajes llegan a su destino sabemos que la precondición ($\exists cache \in recv\_queue(peer.name)$) se cumplirá porque el par siempre enviará la información. El entorno también garantiza que la información está bien formada, por tanto tras este estado la acción `finish_merging` será finalmente realizada, permitiendo a `propose_merging` continuar. No es posible entonces un bloqueo si el algoritmo se encuentra en la acción `start_timeout` o alguna de las siguientes.

`accept_merging` tiene tres estados en su precondición, $timeout\_warning = false$, $pretending\_peers \neq \emptyset$ y $peer = null$. El flag $timeout\_warning$ está activado cuando se recibe una señal *timeout* y desactivado cuando se realiza una acción `propose_merging`. Suponiendo que Δt_i sea mayor que el tiempo necesario para llegar al final de una acción `finish_merging`, el estado $timeout\_warning$ será falso tras la ejecución de un `propose_merging` y su `finish_merging` correspondiente. Hemos mostrado con anterioridad que la acción `finish_merging` siempre se lleva a cabo tras una operación de mezcla, por tanto $peer = null$ será eventualmente cierto junto a $timeout\_warning = false$. Por último, $pretending\_peers \neq \emptyset$ es trivial en una red de más de un agente al cabo de un cierto periodo de tiempo. Entonces no existe la posibilidad de un bloqueo si el algoritmo se encuentra en la acción `receive_merging` o alguna de sus siguientes.

Es posible en este momento definir otro comportamiento razonable que debemos pedirle al entorno. Si el tiempo necesario para ejecutar una operación de mezcla es $t_{ex.i}$ entonces el ritmo máximo al que pueden llegar nuevas peticiones de mezcla es $(\Delta t_i - tex.i)^{-1}$. Como conocemos de los experimentos efectuados [7, 19] este ritmo sigue una distribución de Poisson con media igual a 1. Entonces debemos imponer que el $merging\_timeout$ sea $\Delta t_i \geq 2t_{ex.i}$.

Por tanto en un entorno ideal sin errores de red ni actores malignos el algoritmo nunca se quedará bloqueado en una única acción o ruta de acciones, sino que todas tendrán un tiempo para ser ejecutadas. Si ajustamos Δt_i a valores razonables las colas de comunicación permanecerán cortas.

Es sencillo extender este modelo para actuar en un entorno con errores de red, incorporando algún mecanismo que detecte comunicaciones fallidas cuando se reali-

zan las acciones `propose_merging` o `accept_merging` que abortarían esa ejecución y regresarían a un estado seguro. Por ejemplo en la implementación java de la biblioteca DRM las comunicaciones se realizan a través de sockets, que incorporan un mecanismo temporal que cierra el socket y devuelve una excepción cuando pasa algún tiempo sin que información viaje por el canal. Como cada procedimiento de mezcla se efectua en un hilo de ejecución distinto, la comunicación fallida se puede abortar sin consecuencias para el algoritmo.

Otras propiedades

Es interesante analizar también el algoritmo si lo comprimimos a un único estado y comprobamos su comportamiento. Entonces tenemos un autómata de un estado con dos entradas y una salida. Una de las entradas es la señal de operación entrante de mezcla, que es aceptada y no muestra reacción externa. La otra entrada proviene del reloj del sistema, y entonces el autómata muestra una señal de mezcla saliente en un periodo de tiempo no superior a $t_{ex.i}$. Nunca se verá una salida que no sea precedida por una señal de entrada.

Este comportamiento podría ser modelado como propiedades de seguridad y viveza (no hay salida sin una señal del reloj previa, una salida como máximo $t_{ex.i}$ tras una señal del reloj, ninguna reacción visible tras una señal entrante de mezcla). Las pruebas para estas propiedades podrían ser derivadas del análisis previo de bloqueo.

Es útil recordar que el algoritmo newscast está diseñado para apoyar otras aplicaciones proveyendo las herramientas de comunicación. Estas otras aplicaciones deberían tener sus propios modelos y deberían demostrar sus propias características de viveza y seguridad.

3.2. Análisis de la topología

El objetivo de esta sección es presentar una explicación detallada de la topología creada por el algoritmo newscast. Esto permitirá desarrollar aplicaciones sobre este algoritmo siendo más conscientes del comportamiento que pueden presentar. Esta

sección se basa en el análisis de la construcción de la red, de la evolución dinámica de las conexiones entre nodos y de la composición del coeficiente de agrupamiento como la suma de las afinidades entre un nodo y sus vecinos.

3.2.1. Plataforma de experimentación

Para la realización de pruebas que permitieran investigar aspectos del algoritmo de un modo práctico se implementó en C el algoritmo tal como está descrito en la sección 3.1, con la salvedad de que en lugar de ejecutar los nodos de un modo concurrente estos se ejecutarían en un orden secuencial escogido aleatoriamente en cada iteración.

El tamaño de la red escogido fue de 50000 nodos, con un tamaño de caché de 100 vecinos. El programa se ejecutó en una computadora común de sobremesa con 1GB de RAM y 2GHz de frecuencia de reloj.

3.2.2. Estudio del grafo newscast

Los conceptos de teoría de grafos utilizados y su notación están disponibles en el cuadro 3.3, para una explicación de los mismos se puede consultar [11, 41].

Aleatoriedad

Un ejemplo clásico de un grafo aleatorio es el propuesto por Erdös y Renyi [13] (grafos aleatorios ER). Para construir este grafo aleatorio podemos aplicar el procedimiento siguiente: Consideremos que vamos a definir una familia de grafos llamada $\mathcal{G}$, los cuales se construyen utilizando el mismo conjunto de vértices V. Para cada posible arista $e \in V \times V$ existe una probabilidad $p \in [0, 1]$ de existir en una instancia particular de $\mathcal{G}$.

Podemos utilizar este procedimiento para definir un grafo aleatorio G como una instancia de una cierta clase de grafos $\mathcal{G}$ donde la existencia de sus aristas viene determinada por una distribución de probabilidad dada [34]. Un grafo aleatorio ER tiene un grado medio entre sus vértices de $\bar{k} = p(N-1) \simeq pN$ que sigue una distribución de Poisson. Dicho grafo aleatorio ER tiene un coeficiente de agrupamiento

Grafo	$G = (V, E)$
Vértice	$v \in V$
Arista	$e = (v_1, v_2) \in E$
Grado de v	$k(v)$
Grafo dirigido	$DG = (V, DE)$
Arco	$e = (v_1, v_2) \in DE$
Grado de entrada de v	$k(v)_{in}$
Grado de salida de v	$k(v)_{out}$
Vecindario de v	$N(v)$
Coeficiente de agrupamiento de v	$C(v)$
Diametro de G	$D(G)$

Cuadro 3.3: Notación utilizada en la sección 3.2.2

medio de $C(G) = p \ll 1$ y un diámetro medio de $D(G) = ln(N)$.

Supongamos que tenemos una k-retícula dirigida y le aplicamos el algoritmo newscast. Cada vértice escogerá uno de sus vecinos y ambos unirán sus arcos para crear un conjunto de donde escogerán sus nuevos vecinos. Aunque el número de vecinos para escoger durante una operación de mezcla siempre sera menor que $2\hat{C}$, la distancia entre ellos respecto de la k-retícula original crece exponencialmente. La proporción de arcos reconectados tras i iteraciones p_i se puede calcular como

$$p_i = 1 - \frac{1}{2^i},$$

al cabo de unas cuantas iteraciones cuando se reconecte un arco el vértice final podrá ser cualquier vértice de la k-retícula original, y dado el número habitual de iteraciones en una ejecución del algoritmo newscast la topología del grafo creado será aleatoria, aunque es posible que surjan estructuras emergentes.

Patrón comunicacional

Cada vértice v inicia una comunicación con otro vértice cada Δt_v segundos, y es contactado por otros m_{in} vértices en este mismo periodo de tiempo [19, 24].

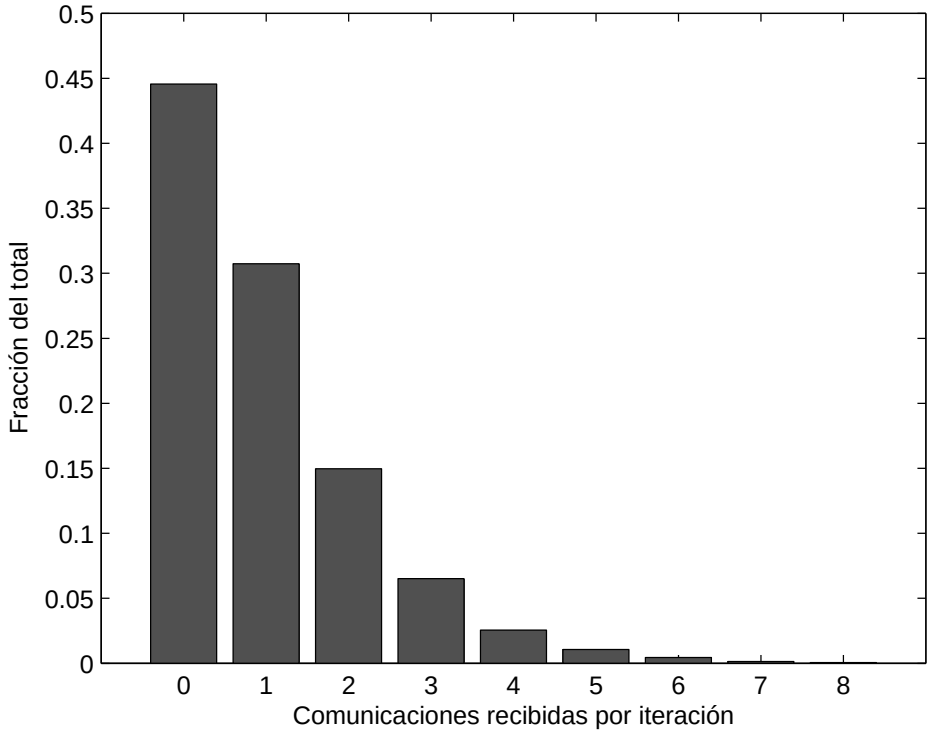

Figura 3.3: Distribución de la recepción de comunicaciones en un vértice.

Suponiendo que el contenido de las cachés es aleatorio entonces la distribución de m_{in} es una binomial $B(n,p)$ donde $n = |V| - 1$ y $p = 1/n$, la media y la varianza de esta distribución se definen como

$$\mu = np, \sigma^2 = np(1-p)$$

Para grafos newscast con un gran número de vértices $\mu = 1$ y $\sigma^2 \simeq 1$, en tales casos esta distribución de comunicaciones de mezcla recibidas puede ser aproximada por una distribución de Poisson con $\lambda = 1$, según [19].

Nuestros propios experimentos se alejan algo de una distribución de Poisson perfecta, como se ve en la figura 3.3. Sería deseable que estos datos nos permitieran definir un grafo aleatorio ER cuyos arcos son las comunicaciones entre vértices correspondientes a la última iteración, y por lo tanto definir la aleatoriedad de las comunicaciones en una red newscast, pero esto no es posible a partir únicamente de los datos presentados en este trabajo.

Coeficiente de agrupamiento

El coeficiente de agrupamiento tal como fue definido originalmente en [42] sólo permitía su aplicación en grafos no dirigidos. Para definir el coeficiente de agrupamiento en un grafo dirigido se toma un vértice v que tiene $|N(v)|$ vecinos, entre los cuales pueden existir como máximo $|N(v)|(|N(v)|-1)$ arcos, considerando que no existen arcos paralelos. Podemos definir el coeficiente de agrupamiento $C(v)$ de v como la fracción de los arcos que realmente existen entre los vecinos de v entre las que podrían existir.

$$E_{N(v)} = \{e(v,w)\}, e \in E, w \in N(v)$$
$$C(v) = \frac{|E_{N(v)}|}{|N(v)|(|N(v)|-1)}$$
$$A(v,w) = \frac{|N(v) \cap N(w)|}{|N(v)|}$$

Sea $G = (V, E)$ un grafo newscast con grado de salida k_{out} en todos sus vértices. Definimos la *afinidad* de dos vértices $A(v, w)$ como la fracción de vecinos comunes que poseen. La afinidad media de un vértice con sus vecinos es el coeficiente de agrupamiento de ese vértice.

El escenario de mayor agrupamiento posible para un vértice v sería uno en el que siempre fuera v quien inicia una comunicación y ningún otro nodo en la red lo hiciera. La afinidad de este vértice ideal con el último vértice contactado w_0 sigue una distribución hipergeométrica $A(v, w_0) = H(\frac{1}{2}, \frac{1}{2}, 1)$, de media 1/2. Su afinidad con w_0 decrecería a la mitad cuando v contacte otra vez con otro vértice distinto. La afinidad de v con cada uno de sus vecinos se reduce a la mitad cada vez que contacta o es contactado por un nuevo vértice.

Esta sucesión y el máximo coeficiente de agrupamiento posible $C(v)^{max}$ se relacionan mediante la siguiente ecuación.

$$C(v)^{max} = \frac{\sum_{i=1}^{k_{out}} H(\frac{1}{2}, \frac{1}{2}, 1)\frac{1}{2^{i-1}}}{k_{out}} \tag{3.1}$$

Para una mayor sencillez de uso se puede aproximar esta ecuación a sus valores

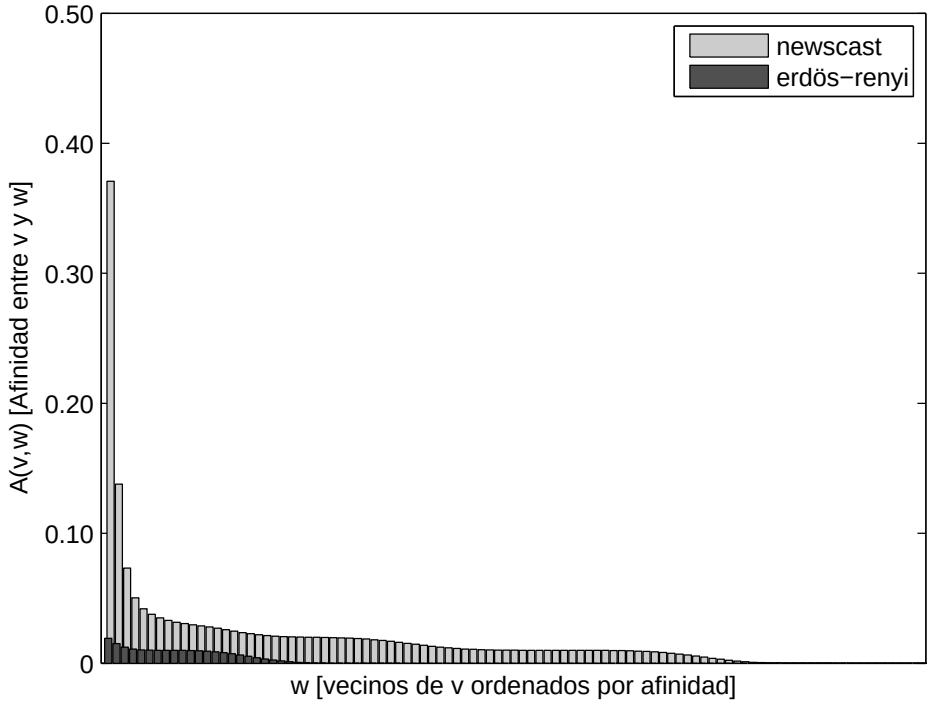

Figura 3.4: Distribución de la afinidad de un vértice con cada uno de sus vecinos.

medios.

$$\bar{C}(v)^{max} = \frac{\sum_{i=1}^{k_{out}} \frac{1}{2^i}}{k_{out}}$$

Cuando v y w efectúen una operación de mezcla, todos los vértices que les apunten verán disminuido su coeficiente de agrupamiento dependiendo de su afinidad con v o w. Exactamente cada uno de los antiguos vecinos de v y w verá dividida por dos la aportación a su coeficiente de agrupamiento de uno de los sumandos de la ecuación 3.1.

La figura 3.4 muestra la afinidad de un vértice con sus vecinos. Para crear esta figura se tomaron los vértices de un grafo newscast y para cada uno se ordenaron sus vecinos por afinidad en una lista. Se hizo lo mismo con un grafo aleatorio de Erdös-Renyi del mismo tamaño. La media de estas listas es la figura mostrada. En ella se puede ver la confirmación de la ecuación 3.1, cada vértice debe la mayor parte de su coeficiente de agrupamiento a su afinidad con un único vecino, disminuyendo luego la afinidad con el resto de sus vecinos.

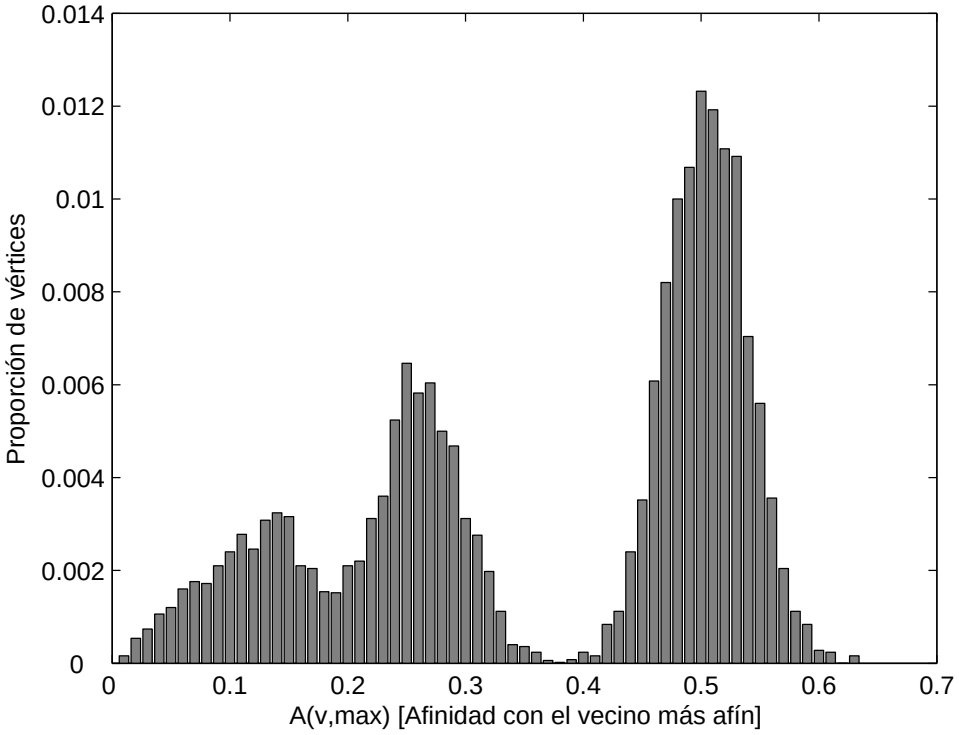

Figura 3.5: Distribución de la afinidad entre cada vértice y su vecino más afín.

La figura 3.5 muestra la afinidad de cada vértice con su vecino más afín. Esta figura indica el estado de cada uno de los vértices en la red, hay una gran cantidad de vértices que aún mantienen una afinidad de 1/2 con su vecino más afín. Estos vértices han contactado o sido contactados recientemente en esta iteración. Por otra parte hay otro grupo de vértices que mantienen una afinidad de 1/4 con su vecino más afín, lo que quiere decir que este vecino más afín ha tenido otra comunicación con posterioridad. Por último hay otro pico en torno a una afinidad de 1/8 y después la distribución se diluye. Esta figura se podría haber obtenido con los datos de la frecuencia de comunicaciones recibidas presentes en la figura 3.3 y la ecuación 3.1.

Existe un componente del coeficiente de agrupamiento que es mucho menor en los grafos newscast estudiados debido al comportamiento de los vértices pero que aun así debe ser igualmente tenido en cuenta. Cuando v mezcla su caché con w, es posible que en la caché de w se encuentre algún vecino de los vecinos de v, digamos u, que si terminan en la caché de v harán incrementar la afinidad en una pequeña cantidad

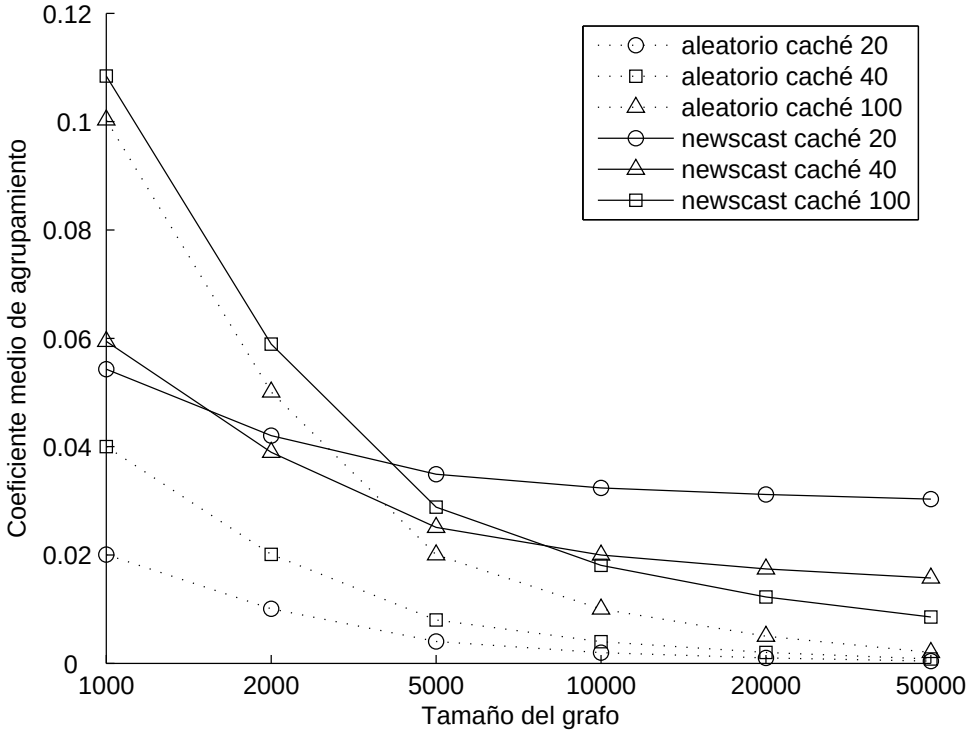

Figura 3.6: Coeficiente de agrupamiento de un grafo newscast dependiendo del tamaño y grado del grafo.

entre u y v. Este efecto proviene de la estructura aleatoria del grafo newscast, siendo equivalente al coeficiente de agrupamiento que tendría un grafo ER de la talla y grado del grafo newscast estudiado. El valor de este componente aleatorio del coeficiente de agrupamiento se define como

$$C_{vr} = k_{out}/|V|,$$

calculándose entonces tras esta corrección el coeficiente máximo de agrupamiento de v como

$$C_v^{max} = \frac{\sum_{i=1}^{k_{out}} \frac{1}{2^i}}{k_{out}^2} + k_{out}/|V|.$$

Entonces, como se muestra en la figura 3.6, si mantenemos el tamaño de la caché constante e incrementamos el número de vértices en el grafo newscast el coeficiente de agrupamiento tiende al valor de la componente newscast ya que el valor de la componente aleatoria decrece. Esto es debido a que $|V|$ sólo afecta a la componente

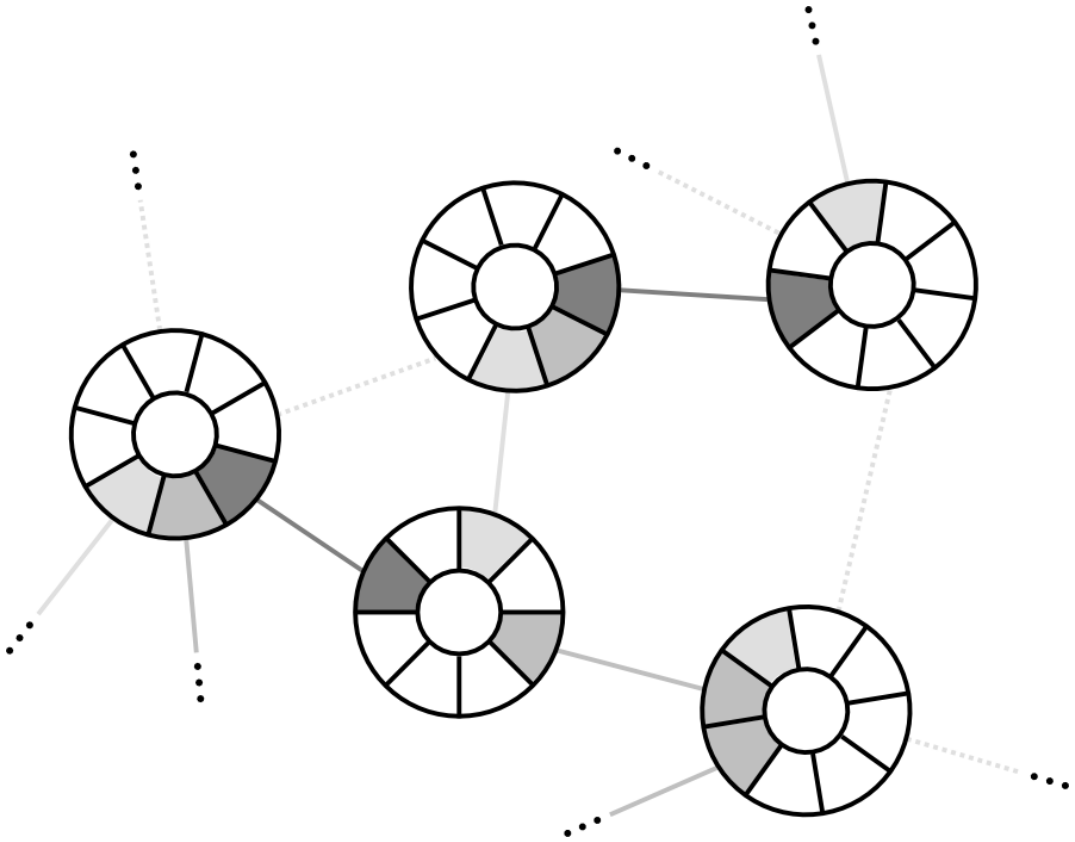

Figura 3.7: Detalle de la estructura de la red newscast.

aleatoria. Asimismo también se observa que si se mantiene el número de vértices $|V|$ constante pero se incrementa el tamaño de la caché k_{out} el coeficiente de agrupamiento se aproxima al valor de la componente aleatoria que es la incrementada por este cambio.

La estructura entonces de un grafo newscast si lo viéramos en un único instante de su evolución sería la de un grafo aleatorio donde aproximadamente la mitad de los vértices están pareados y comparten aproximadamente la mitad de sus vecinos como se ve en la figura 3.7. Aparte de su pareja, cada vértice está también fuertemente relacionado con otro vértice más, con el que comparte un cuarto de sus vecinos. Si observamos la evolución dinámica de este grafo newscast veremos también que estas relaciones se crean y destruyen continuamente, durando menos de una iteración del algoritmo.

Capítulo 4

Implementación de DRM

Este capítulo expone la implementación de la biblioteca Distributed Resource Machine (DRM), ampliando sustancialmente la documentación existente distribuida con el código fuente de DRM.

Una red DRM es un conjunto distribuido de programas que pueden ejecutarse en distintas máquinas. Estos programas pueden ser dividido entre bases y agentes. Las bases se ejecutan asociadas a una máquina y mantienen enlaces de comunicación entre sí. Las bases pueden alojar agentes. Un agente es un programa móvil que realiza una tarea. Los agentes se alojan temporalmente en una base, siendo capaces de cambiar de base cuando lo deseen. Los agentes se pueden comunicar con el exterior de sus bases utilizando los servicios de comunicación que éstas ofrecen.

El algoritmo newscast explicado en el capítulo 3 se implementa mediante un objeto conocido como colectivo. Una base que ejecuta el algoritmo newscast se denomina nodo, cada nodo que participe del algoritmo newscast posee una instancia del colectivo. Cada instancia del colectivo contiene una caché, que es una vista parcial de la red DRM compuesta por un cierto número de contribuciones de otros nodos. El colectivo ejecuta el algoritmo newscast de manera independiente al nodo que lo posee, utilizando los recursos de comunicación a su alcance y variando la vista local o caché continuamente. Un nodo le puede proporcionar a su instancia del colectivo una contribución para que se difunda por la red.

Este capítulo explica en primer lugar las estructuras de datos básicas que componen DRM, continúa con el funcionamiento de bases y agentes y termina con la

implementación del algoritmo newscast mediante colectivos. El diagrama de clases de la aplicación se muestra en la figura 4.1, que ha sido dividido en partes y repetido a lo largo de este capítulo para ilustrar los diversos objetos que componen la aplicación.

4.1. Clases básicas de DRM

El paquete `drm.agentbase` define una funcionalidad nuclear muy básica para manejar agentes móviles. Esta aplicación es más simple que muchos sistemas de agentes sofisticados. Esto es a causa de los requerimientos especiales y objetivos de la arquitectura explicados en [30].

El paquete define el concepto de una base y el agente como los dos bloques constructivos de un sistema de agentes. La idea es que la base es un contenedor o campo de acción para los agentes. La base provee servicios que los agentes pueden utilizar cuando se mueven a otra base o cuando necesitan información del entorno o cuando desean comunicarse con otros agentes o bases. La base proporciona servicios no sólo a sus agentes sino también a su entorno. Pueden añadirse agentes o se puede pedir la lista de agentes, entre otras acciones.

Otros dos conceptos básicos se definen: la dirección y el mensaje. Cada base y agente tiene una dirección y se pueden mandar mensajes entre bases y agentes, agentes y agentes, y bases y bases.

4.1.1. Estructuras de Datos Básicas: Address y Message

Existe una serie de estructuras que son básicas para el funcionamiento de la Distributed Resource Machine. Estas estructuras definen los roles y las acciones que se pueden tomar dentro de una red DRM.

Address

Cada actor (una base o un agente) en una red DRM tiene una dirección única mediante la cual puede ser localizado, esta dirección se denomina `Address` y contiene tres campos:

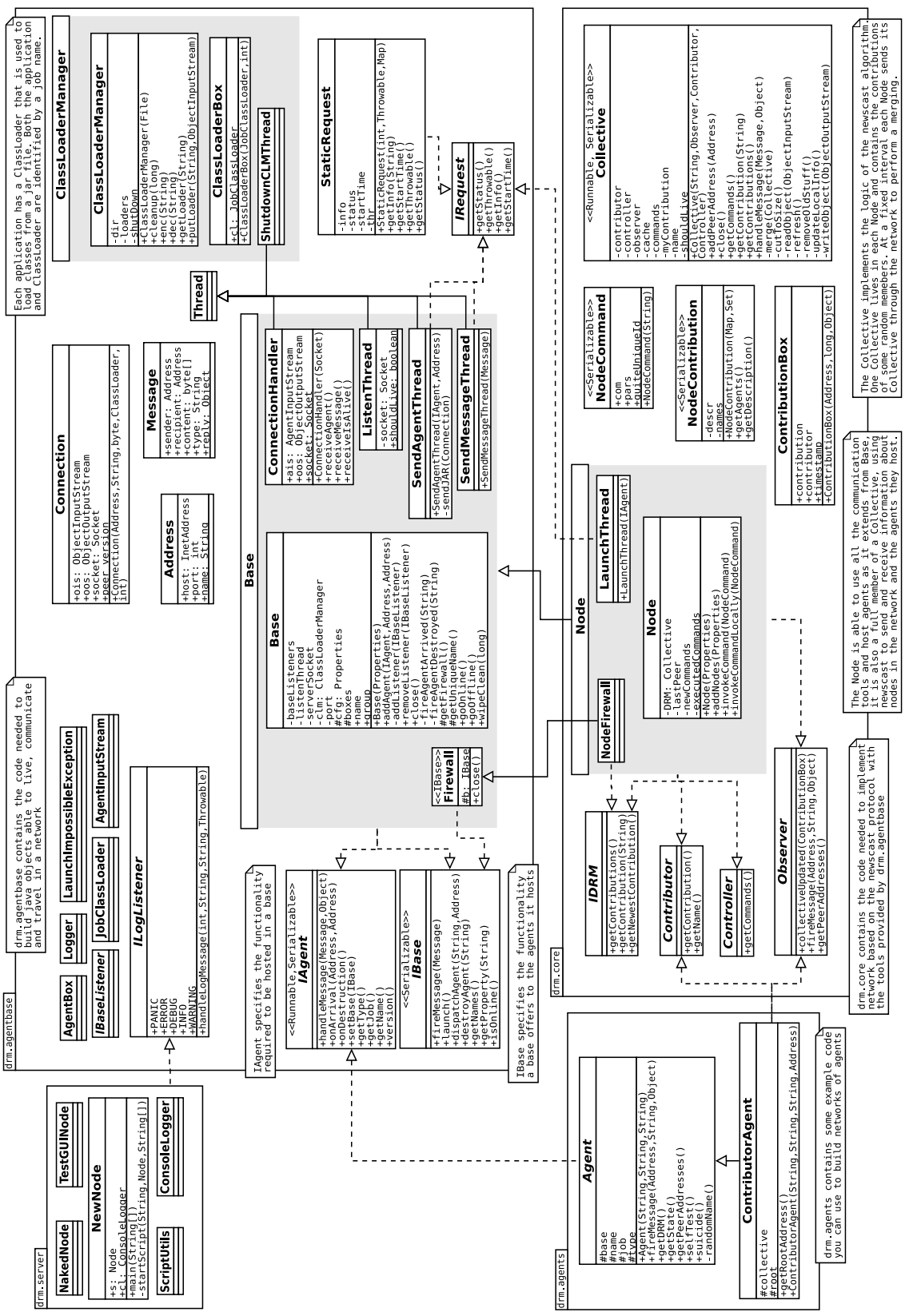

Figura 4.1: Diagrama de clases de DRM.

`String: name` - Esta cadena de texto identifica al actor unívocamente y es el único campo de una dirección que nunca puede ser nulo.

`InetAddress: host` - La dirección IP del actor.

`int: port` - Puerto en el que el actor escucha.

Una dirección *local* es una dirección que sólo contiene el nombre del actor. Si no se aporta más información sólo servirá para contactar con un agente que esté actuando en la misma base. Una dirección con valores válidos en todos los campos sirve para contactar con un actor que se encuentre en cualquier lugar de Internet. Un actor que sólo dispone de una dirección local le puede preguntar a otros actores para que le proporcionen la información necesaria para transformarla en global.

Message

Un mensaje es el elemento básico (aunque no el único) que los actores pueden enviarse entre sí.

`Address: sender` - La dirección del actor que envía el mensaje.

`Address: recipient` - La dirección del actor que debe recibir el mensaje.

`String: type` - Una cadena de texto que puede utilizarse para clasificar el mensaje.

`byte[]: content` - Un contenedor que puede utilizarse para incluir cualquier tipo de contenido con el mensaje.

`Object: reply` - Cuando se recibe un mensaje se puede asignar a este campo un objeto que será enviado de vuelta al remitente.

Esta estructura permite una gran flexibilidad en el envío de información entre actores. Dadas las facilidades del lenguaje Java para serializar objetos es posible enviar cualquier estructura de datos con un mensaje. El que el receptor del mensaje sepa interpretarla es otro problema que se estudia en la sección 4.1.7.

4.1.2. IRequest

`IRequest` es una interfaz que define el manejo de peticiones de comunicación asíncronas. Tiene un campo, `status`, que define el estado de la petición y que tiene tres valores posibles: `WAITING`, `DONE` y `ERROR`. Con estos estados se indica que la petición está efectuándose, ha terminado o ha fallado. Tiene funciones para obtener información acerca del momento de inicio de la petición, la excepción que hizo fallar la petición si ha fallado por esta causa y un método genérico para obtener otra información que sepamos que ha sido almacenada en la petición.

4.1.3. IBase

Esta interfaz está definida en el código fuente como "la funcionalidad que una base ofrece a sus agentes". Estos son los métodos que ofrece:

`String: getProperty(String prop)` - Sirve para obtener información de la base, como por ejemplo el nombre mediante la clave `drm.baseName`.

`Set: getNames()` - Devuelve los nombres de los agentes que habitan en la base.

`void: destroyAgent(String name)` - Elimina un agente de la base.

`IRequest: dispatchAgent(String name, Address destination)` - Envía el agente con nombre `name` a la base en la dirección `destination`.

`IRequest: launch(String method, IAgent agent, Object parameter)` - Envía una copia del agente `agent` a otra base según el parámetro `method`. Es posible también mediante este método duplicar un agente y ejecutarlo en la misma base. El parámetro `parameter` es un un comodín para darle información al método de lanzamiento que empleemos.

`IRequest: fireMessage(Message m)` - Envía el mensaje `m`, en el propio mensaje está incluida la dirección de destino.

`boolean: isOnline()` - Dice si la base está en condiciones de ofrecer servicios.

Tal como está definida, la interfaz `IBase` representa lo que un agente debe saber de la base que habita, la interfaz sin embargo no define lo que la base es. También existen algunos problemas de consistencia, como el tener que obtener el nombre de la base mediante `base.getProperty("drm.baseName")` cuando para un agente esto se obtiene mediante el método `getName()`.

4.1.4. IAgent

La interfaz `drm.agentbase.IAgent` define la funcionalidad que se le exige a los agentes, que son programas Java capaces de residir en una base y de reaccionar a la recepción de mensajes, además de poder realizar una tarea propia. Esta interfaz extiende las interfaces `Runnable` y `Serializable`. Un agente debe implementar los siguientes métodos:

`void: onArrival(Address from, Address to)` - Este método se llama cuando un agente llega a una base después de haber sido informado de los servicios que ésta le ofrece mediante el método `setBase(IBase b)`. `to` es la dirección de la base a la que ha llegado, lo cual le sirve para saber su propia dirección. `from` es la dirección de la base que le lanzó a su posición actual, la cual puede servir entre otras cosas para conectar con los demás agentes iniciados por la misma base.

`void: onDestruction()` - Este método es llamado antes de que el agente sea destruido por la base, incluyendo cuando el agente se mueve de una base a otra justo tras la serialización del mismo. Es la última oportunidad del agente para usar servicios de la base.

`String: getType()` - El nombre de un agente está compuesto de tres secciones separadas por puntos, este método devuelve la que identifica el tipo del agente, que podría interpretarse como la clase del agente, su función específica.

`String: getJob()` - Este método devuelve la sección del nombre que identifica al agente como perteneciente a un grupo de trabajo.

`String: getName()` - Este método devuelve la sección del nombre que identifica el agente entre otros del mismo tipo que realizan el mismo trabajo, también llamada `name`.

`void: setBase(IBase b)` - Este método se llama cuando un agente llega a una base para permitirle llamar a los métodos de ésta filtrados a través de un `Firewall`.

`boolean: handleMessage(Message m, Object o)` - Este método se llama para entregarle un mensaje a un agente, que devolverá cierto si es capaz de interpretarlo o falso si no. El parámetro `o` puede contener el campo `content` del mensaje deserializado por la base.

`int: version()` - Este método devuelve un entero que identifica la versión del agente por si existieran problemas de compatibilidad entre implementaciones.

Existen varios problemas con esta definición de agente. Es evidente que un identificador y a una sección del mismo no pueden denominarse de la misma manera, como sucede con el campo `name`. Tampoco parece una solución muy elegante el pasarle como parámetros a la función `handleMessage(m,o)` un objeto serializado y su copia deserializada.

Pero el problema más grave es que no se entiende la interfaz como una definición de un agente al exigir que pueda recibir mensajes, cuando no exige que los pueda mandar. Es más lógico definir un agente como un programa que reside en una base y realiza una tarea y, si se desea, añadir que tiene capacidades de comunicación con otros actores.

4.1.5. Base

Cada base implementa dos interfaces, `IAgent` e `IBase`. Estas interfaces se han detallado en las secciones 4.1.4 y 4.1.3. En la figura 4.2 se muestra el diagrama de clases relacionado.

La interfaz `IBase` define los métodos que una base ofrece a los agentes que la habitan. Incluye dos métodos para replicar agentes y extender la red, un método para

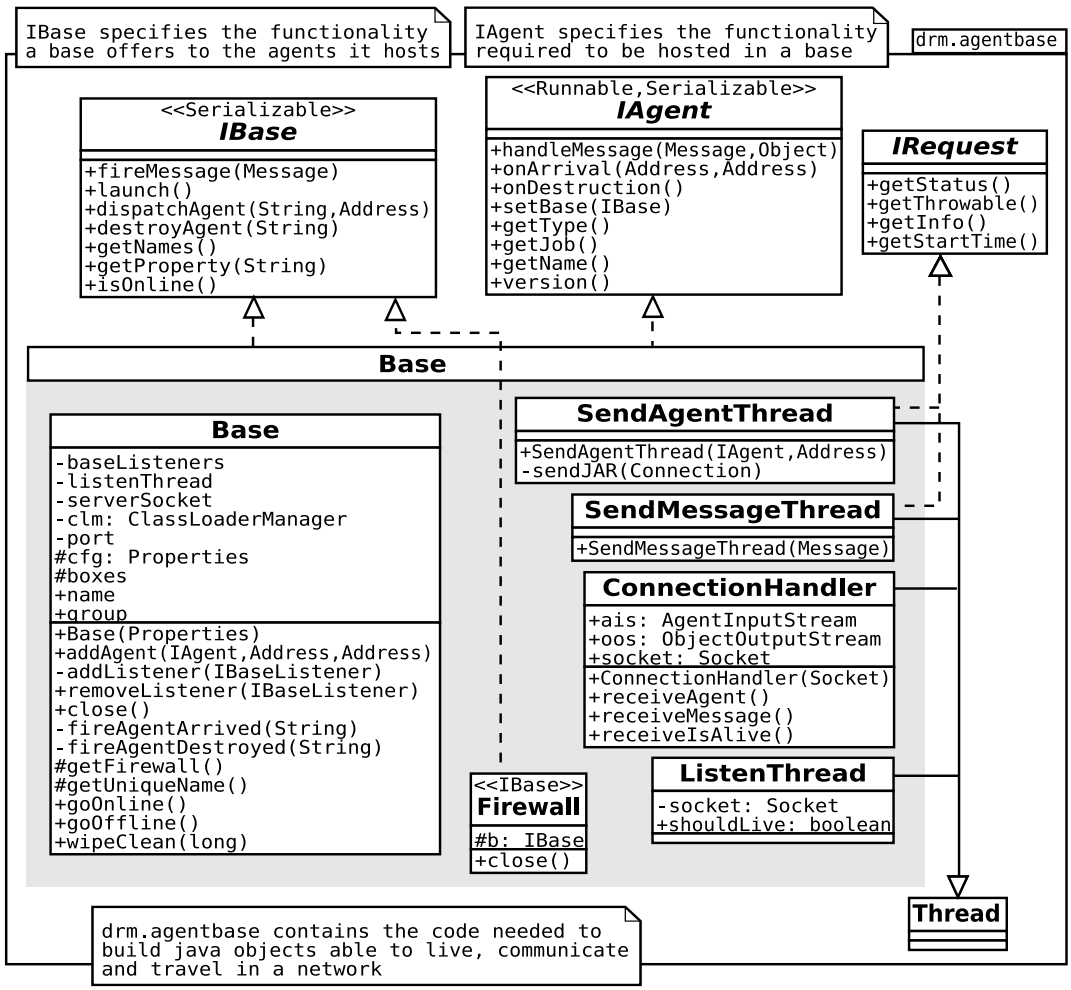

Figura 4.2: Diagrama parcial de clases relativo a `drm.agentbase.Base`

ofrecer información sobre los agentes que acoge, un método para eliminar agentes y por último un método para enviar mensajes.

Como ya se ha comentado, en la implementación de `Base` se utiliza tanto la interfaz `IBase` como `IAgent`. La interfaz `IBase` es necesaria para que cualquier agente encuentre el mismo conjunto básico de métodos en cada base. No se entiende muy bien que se utilizase también la interfaz `IAgent` en la implementación de `Base`, ya que se niegan la mayoría de los métodos.

La clase `Base` dispone de una serie de estructuras para almacenar datos, como los agentes en ejecución que acoge y las bibliotecas de las que dependen.

`Socket: serverSocket` - El socket donde la base puede recibir comunicaciones, está vigilado por un `ListenThread`.

`ClassLoaderManager: clm` - La estructura que gestiona las bibliotecas de las que dependen los agentes. Es capaz de leer estas bibliotecas a través de la red y almacenarlas en disco, también puede proporcionarlas más tarde si la base las necesita.

`Map: boxes` - La estructura donde de guardan los agentes junto con algo de información adicional, en la forma de un `AgentBox`. Cada agente se ejecuta en un thread propio y dispone de una referencia a los métodos de la base a través de un `Firewall` que garantiza que sólo se puede acceder a los métodos definidos en la interfaz `IBase`.

La clase `Base` también contiene una serie de clases internas que habilitan las comunicaciones de modo asíncrono.

`ListenThread` - Es un thread asíncrono que cuando detecta una comunicación de entrada en el `serverSocket` invoca un thread `ConnectionHandler` para que lo maneje. Es accesible desde la variable `listenThread` de `Base`.

`ConnectionHandler` - Es un thread asíncrono que gestiona una comunicación de entrada. Ejecuta primero un protocolo básico de comunicación, descrito en la sección 4.2.2, para asegurar que la comunicación cumple unas condiciones

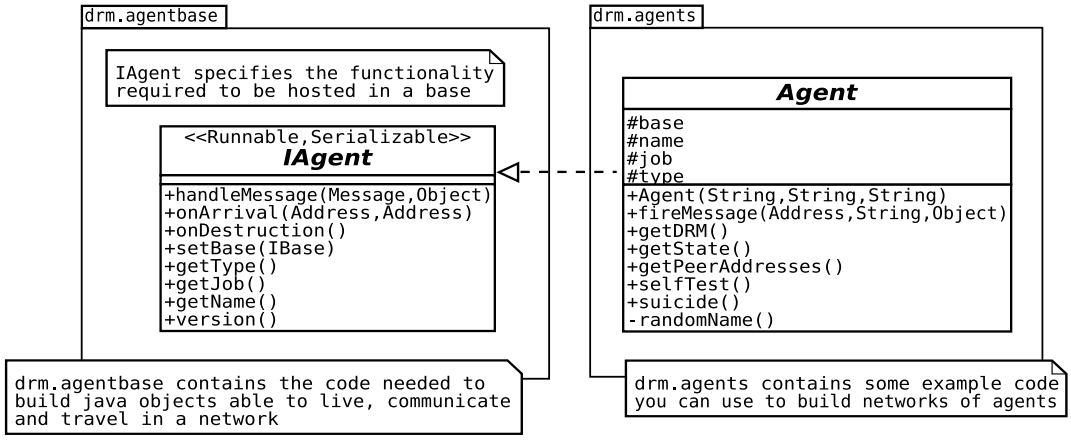

Figura 4.3: Diagrama parcial de clases relativo a `drm.agents.Agent`

básicas y posteriormente llama a un método específico para el tipo de comunicación que se esté efectuando.

`SendAgentThread` - Un thread asíncrono que implementa `IRequest`. Abre una `Connection` a una dirección remota y envía un agente.

`SendMessageThread` - Un thread asíncrono que implementa `IRequest`. Abre una `Connection` a una dirección remota y envía un mensaje.

4.1.6. Agent

Esta implementación de `IAgent` se encuentra en el paquete drm.agents y su diagrama de clases se muestra en la figura 4.3. Se espera que los usuarios la extiendan para sus propios usos, para ello deben programar la tarea que realizan los agentes en el método `run()`.

El acceso al entorno que acoge al agente se realiza a través de la variable `base`, y en el caso de que el agente se aloje en un nodo se puede acceder a su colectivo mediante `getDRM()`. La base que aloja al agente periódicamente consulta el valor de la variable `shouldLive` para saber cuando eliminarlo.

`void: onArrival()` - Le da a `shouldLive` el valor de `true` y marca el inicio de su actividad.

`void: setBase()` - Inicializa la variable `base` para poder utilizar los servicios de comunicación.

`boolean: handleMessage()` - Mediante este método el agente sabe como interpretar mensajes del tipo "`selfTest`" y "`getState`".

`IRequest: fireMessage()` - Este método permite al agente enviar mensajes mediante la base.

`IDRM: getDRM()` - Recupera la variable `collective` del nodo.

`void: suicide()` - El agente no puede detener el mismo su hilo de ejecución ni actualizar las variables necesarias en la base. Mediante este método le pide a la base que lo haga.

`void: onDestruction()` - Le da a `shouldLive` el valor de `false` y provoca que el agente detenga su ejecución.

Es importante diferenciar entre la variable `base`, que da acceso a los servicios de la misma, y el método `getDRM()` que devuelve la variable `collective` del nodo y que permite acceder a los servicios de la red newscast.

4.1.7. ClassLoaderManager

Hay tres clases que intervienen en el proceso de variar el `CLASSPATH` dinámicamente, indicadas en la figura 4.4.

Un `JobClassLoader` es una clase que extiende de `URLClassLoader` y que carga definiciones de clase de un fichero almacenado en disco.

Un `AgentInputStream` es un stream que extiende `ObjectInputStream` y que al crearlo se le puede dar como parámetro un `ClassLoader`. Éste se utiliza para deserializar un objeto si fallan los `ClassLoader` del sistema.

Por último, el `ClassLoaderManager`, que tiene dos funciones principales:

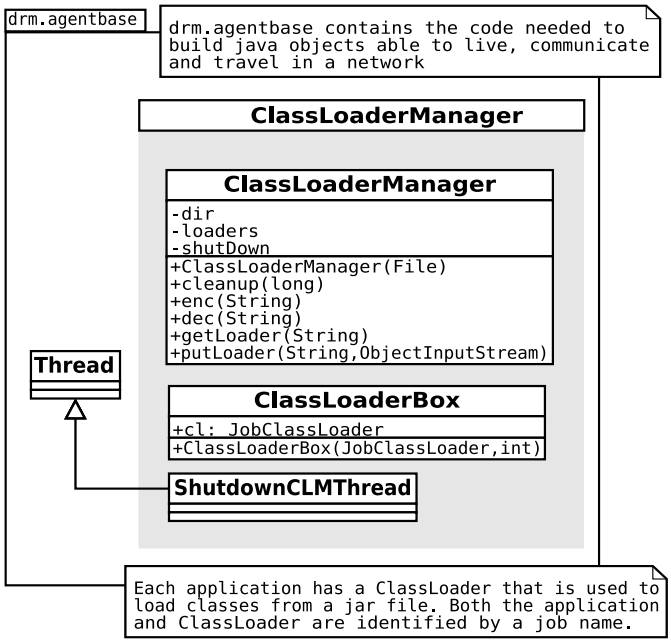

Figura 4.4: Diagrama de clases relativo a `drm.agentbase.ClassLoaderManager`

Almacena en disco los ficheros .jar que se le entregan a través de un `ObjectInputStream` y sabe como acceder posteriormente a ellos si se le proporciona el trabajo o grupo al que sirven. Almacena un fichero con el método `putLoader(String job, ObjectInputStream ois)` y lo recupera con el método `getLoader(String job)`.

Controla qué bibliotecas están siendo utilizadas y cuáles no, y elimina aquellas que llevan un tiempo sin ser utilizadas.

4.1.8. Firewall

Un agente no llama directamente a los métodos que ofrece la base que lo aloja, sino que invoca los métodos de una clase interna llamada `Firewall` que también implementa `IBase`. `Firewall` hace las peticiones a la base y devuelve los resultados al agente.

El fin de `Firewall` es el de ofrecer a los agentes de una base acceso a los métodos de la misma presentes en la interfaz `IBase` mientras se le esconden los otros métodos públicos de la base.

4.2. Acciones principales en una red DRM

4.2.1. Inicio de una base

Para iniciar una base basta con seguir los siguientes pasos:

```
Base startBase(Properties cfg){
     Base b = new Base(cfg);
     if(b.goOnline(10000,10010)<0)
          Logger.error(“Couldn’t go online”);
     b.start();
     return b;
}
```

El constructor de `Base` realiza gran parte del trabajo. Su primer objetivo es establecer el nombre y el grupo (o trabajo) de la base, introducidos en el argumento de tipo `Properties` con las claves `drm.baseName` y `group`. Después crea un `ClassLoaderManager` que se encargará de manejar las bibliotecas que puedan necesitar los agentes que habiten en la base. Acepta un parámetro con clave `java.io.tmpdir` para designar el directorio donde se buscarán y almacenarán las bibliotecas.

La función `goOnline(...)` trata de abrir un socket receptor en el primer puerto disponible del rango que se le proporciona. Si lo consigue crea un thread del tipo `ListenThread` que se encargará de reaccionar a las comunicaciones que lleguen. Este thread se inicia en la última línea.

4.2.2. Inicio de una comunicación

Una comunicación entre dos bases se inicia con el emisor creando una clase `Connection` con la dirección del receptor de la comunicación, el grupo al que debe pertenecer el receptor, el tipo de comunicación (`mensaje, agente, isalive`) y el `ClassLoader` necesario para deserializar el agente. Esta `Connection` abre un canal de comunicación sobre la IP y el puerto indicados en la dirección destino.

Cuando el `ListenThread` en la base destino detecta en su socket que alguien quiere enviar información inicia un `ConnectionHandler` que se hace cargo.

Según la información que vaya a ser transmitida entre los dos el protocolo puede variar, pero los primeros pasos realizados por la clase `Connection` hasta que se sabe qué tipo de comunicación se va a realizar son siempre iguales. A continuación y en la figura 4.5 se expone el protocolo utilizado:

1. El emisor crea un socket hacia la IP y puerto del receptor.
2. Si el socket se abre el emisor crea un `ObjectOutputStream` (oos) sobre él.
3. El receptor crea un nuevo `AgentInputStream` (ais) sobre el socket.
4. El emisor envía la versión del protocolo que utiliza.
5. El emisor envía el grupo al que debe pertenecer el otro extremo.
6. El emisor envía el identificador del tipo de comunicación que se va a realizar.
7. El emisor abre un `AgentInputStream` (ais) sobre el socket para recibir información.
8. El receptor abre un `ObjectOutputStream` (oos) sobre el socket para enviar información.
9. Si los grupos no coinciden el receptor envía `GROUP_MISMATCH` y ambos abortan la conexión.
10. Si las versiones no coinciden el receptor envía `VERSION_MISMATCH` y ambos abortan la conexión.
11. Si todo es correcto el receptor envía `OK` y se continúa.
12. En el emisor se continúa con el protocolo específico del thread que invocó la `Connection`. El receptor ejecuta el método específico al tipo de comunicacion: `receiveMessage()`, `receiveAgent()` o `receiveIsAlive()`.

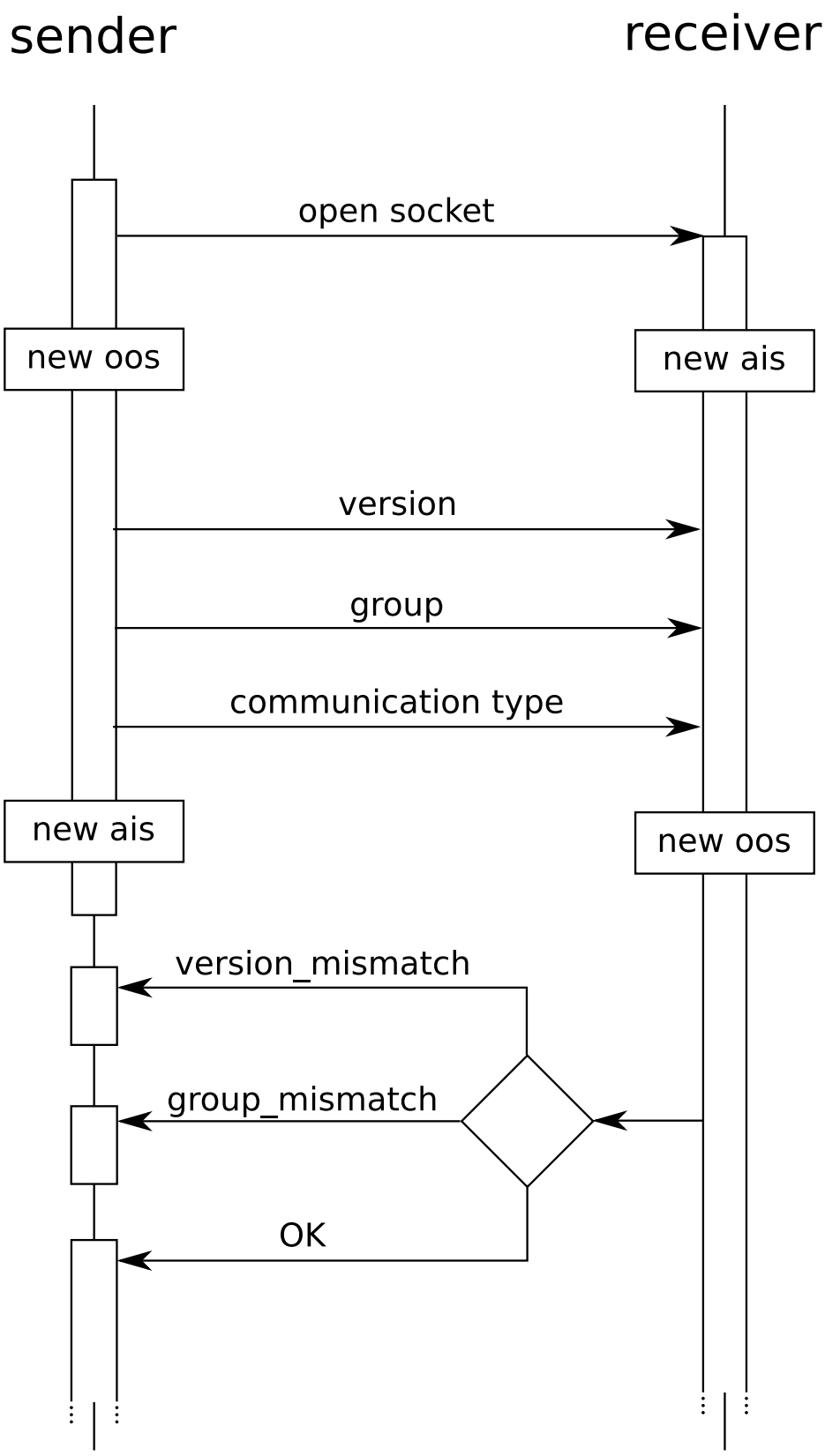

Figura 4.5: Protocolo inicial de toda comunicación entre actores.

4.2.3. Transmisión de un agente

Para envíar un agente una `Base` dispone de una clase interna privada llamada `SendAgentThread` que extiende `Thread` e implementa `IRequest`. Esto quiere decir que enviar un agente es una tarea asíncrona que no interfiere con el resto de las actividades de la base. Para crear un `SendAgentThread` hace falta consultar si el dispositivo de seguridad de Java lo permite con `AccessController.doPrivileged()`. Si la base detecta que el receptor es uno de sus agentes entonces no se crea un `SendAgentThread` sino que se hace directamente.

Un `SendAgentThread` se crea como respuesta a una invocación de `base.launch()` o `base.dispatchAgent()` y sólo precisa como parámetros el agente a duplicar en destino, la dirección del destino y si es preciso destruir el agente local tras la ejecución (así es posible escoger entre duplicar un agente o moverlo a otra base).

En el receptor, `ConnectionHandler` llama al método `receiveAgent()` en el momento que sabe que la comunicación cumple los requisitos básicos de coincidencia de protocolo y de grupo y que va a recibir un agente como se describe en la sección 4.2.2. Este método realiza las gestiones necesarias para disponer de las bibliotecas adecuadas para deserializar el agente y entregárselo al método que le dará un thread para que se ejecute. El procedimiento, ilustrado en la figura 4.6, es el siguiente:

1. El emisor envía el grupo (o trabajo) del agente. Para el `ClassLoaderManager` esta cadena es el identificador de la biblioteca donde sabe que puede encontrar el tipo de datos del agente, lo cuál es necesario para deserializarlo.

2. El receptor comprueba con el método `clm.getLoader(grupo)` si dispone de la biblioteca necesaria. Si es así se envía `OK` y se continúa, si no se envía una señal `GET_JAR` y se le pasa el `ais` al `ClassLoaderManager`.

3. Si el emisor recibe una señal `GET_JAR` llama a la función `sendJAR()` que envía el fichero .jar o el directorio con los archivos .class al receptor. En el receptor el `ClassLoaderManager` utiliza la función `putLoader(grupo,stream)` para almacenarlo y poder utilizarlo posteriormente.

4. Si la transmisión del .jar ha sido correcta el receptor deja continuar la transac-

ción enviando una señal `OK`.

5. El emisor envía el agente y el receptor lo reconstruye con las herramientas de serialización provistas por las bibliotecas de entrada/salida de Java.

6. El emisor envía también su dirección puesto que el receptor no conoce todavía su nombre y el agente puede necesitarlo.

7. Del socket el receptor obtiene también su propia dirección. Con toda esta información el receptor ejecuta el método `addAgent()` que añadirá el agente a la estructura `boxes` y le dejará ejecutarse en un thread. Este proceso se describe en la sección 4.2.6

8. Dependiendo de si el paso anterior se ejecuta correctamente o no el receptor envía de vuelta un `OK` o un `NOT_OK`.

4.2.4. Transmisión de un mensaje

El envío de un mensaje es muy parecido al de un agente. El emisor utiliza igualmente un thread que implementa `IRequest` y es por tanto asíncrono. Este thread sólo precisa para su construcción del mensaje a enviar. `SendMessageThread` abre una `Connection` al receptor, ambos ejecutan el protocolo inicial descrito en la sección 4.2.2 y después prosiguen con la transferencia del mensaje. Al igual que con `SendAgentThread` el thread sólo se crea si el receptor no es local y la seguridad se gestiona mediante llamadas a `AccessController.doPrivileged()`. El procedimiento, ilustrado en la figura 4.7, es el siguiente:

1. El emisor envía el nombre del agente al que está dirigido.

2. El receptor comprueba si existe un agente con ese nombre o si esta dirigido a la base. Se informa mediante `OK` o `NOT_OK` del resultado de esta comprobación.

3. Si el emisor recibe un `OK` entonces se envía el mensaje mediante las herramientas de serialización de Java.

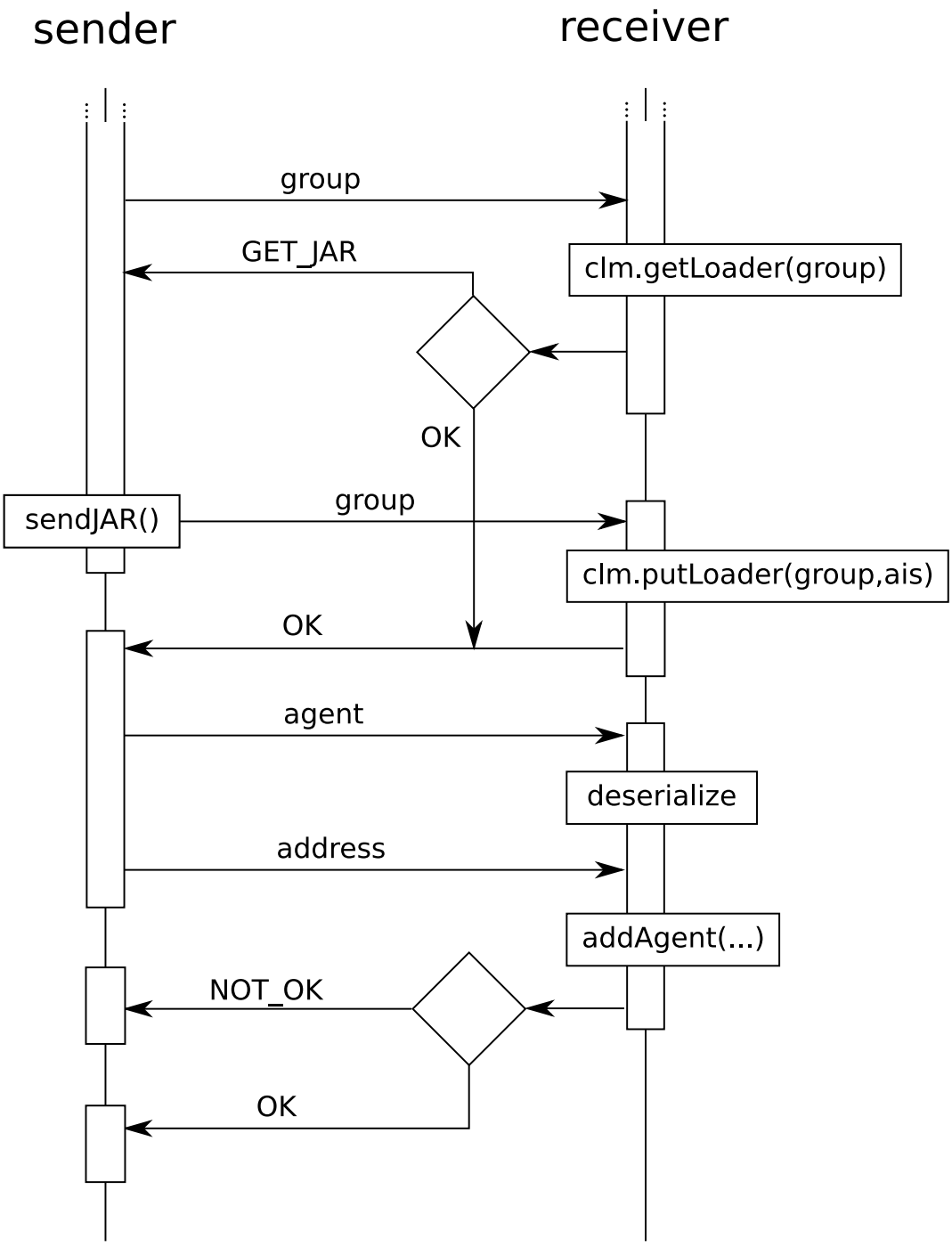

Figura 4.6: Protocolo de transmisión de un agente.

4. El receptor incorpora al mensaje la dirección IP del emisor, obtenida del socket. Se decodifica en una variable aparte el objeto adjunto si existe (porque el agente no tiene acceso al `ClassLoaderManager` y no puede decodificarlo). Por último, se llama al método `handleMessage(Message m, Object o)` del agente.

5. Si cualquiera de estos pasos falla o el agente falla al interpretar el mensaje se le informa de ello al emisor mediante un `NOT_OK`.

6. El receptor tiene la posibilidad de rellenar un campo `reply` en el mensaje que sera inmediatamente transmitido al emisor. Si todo ha ido bien un `OK` por parte del receptor finaliza la comunicación.

4.2.5. Comprobación de la existencia de una base.

El método `base.getBaseName()` es muy breve y no posee un thread propio, por lo tanto es un método de comunicación síncrono. Este método sirve para saber si existe una base de un determinado grupo `grp` escuchando en el puerto `p` de la dirección IP `h`. Puede equivaler a un ping.

El emisor abre una `Connection` como en todas las comunicaciones y se lleva a cabo el protocolo de comunicación común. El `ConnectionHandler` en el emisor llama al método `receiveIsAlive()` que envía al emisor el nombre del receptor, con lo que la comunicación finaliza.

4.2.6. Añadir un agente a una base

El método `addAgent(IAgent a, Address from, Address here)` es privado y por tanto sólo la propia base puede llamarlo, tras haber recibido un agente por medio, por ejemplo, de una comunicación desde otra base.

La base debe construir un `AgentBox` para que el agente pueda ejecutarse. Este `AgentBox` requiere el agente, la hora actual, el firewall y el thread en el que se ejecutará el agente.

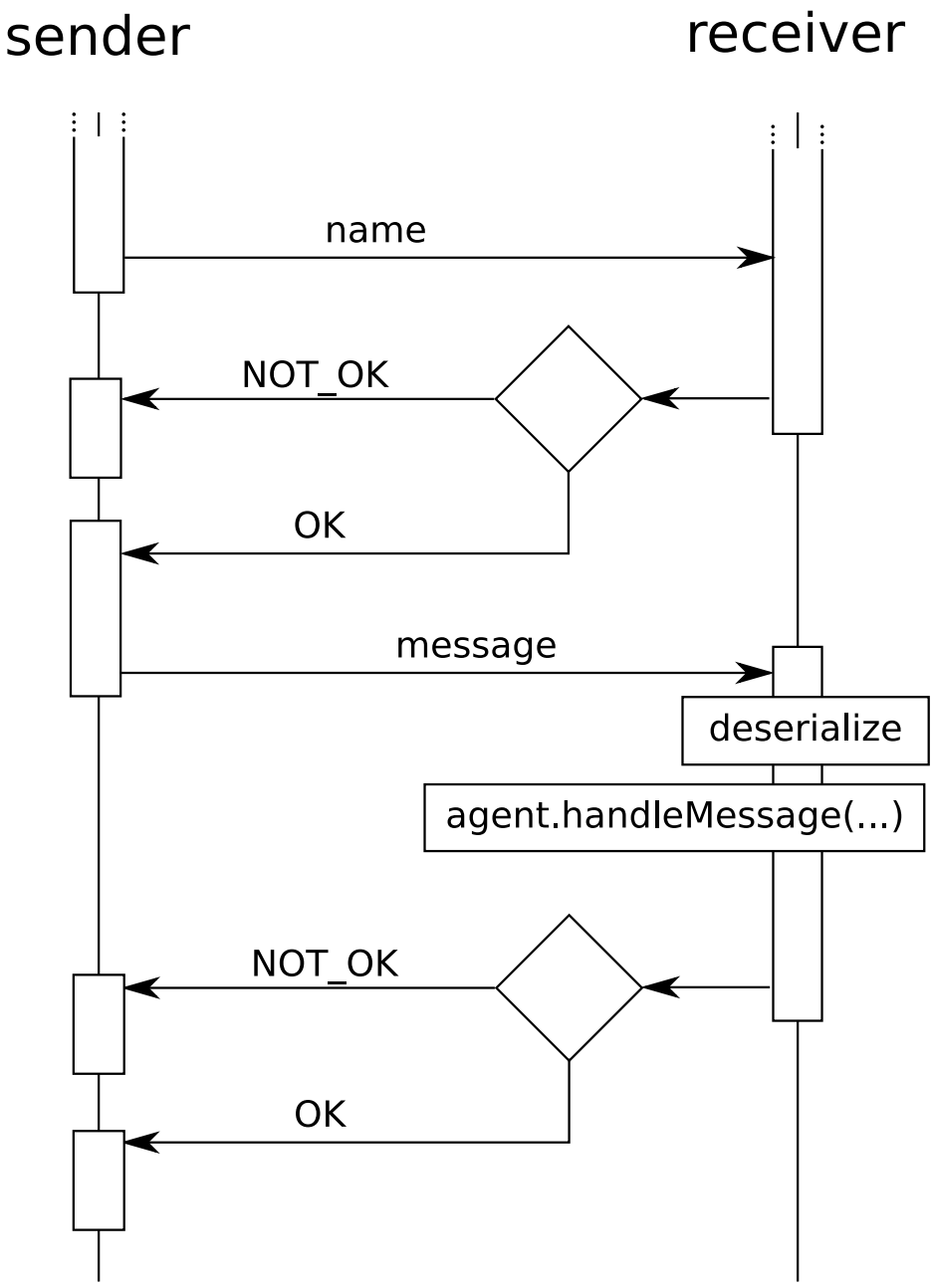

Figura 4.7: Protocolo de transmisión de un mensaje.

```
Firewall fw = getFirewall();
Thread th = new Thread(agent, agent.getName());
java.util.Date time = new java.util.Date();
AgentBox box = new AgentBox(agent, th, time, fw);
```

Una vez obtenido el `AgentBox` es el momento de habilitar al agente para usar los servicios de la base mediante el firewall, añadir el `AgentBox` a las boxes, avisar a los `BaseListener` de la acción, avisar al propio agente para que realice sus funciones de inicialización, avisar al `ClassLoaderManager` de que la biblioteca de ese agente está siendo utilizada para que no la borre y, por último, activar el agente.

```
agent.setBase(fw);
synchronized(boxes){
     boxes.put(agent.getName(), box);
     fireAgentArrived(agent.getName());
     a.onArrival(from, here);
}
clm.inc(agent.getJob());
thread.start();
```

4.3. Implementación del algoritmo newscast

El paquete `drm.core` implementa el algoritmo newscast en la red DRM. El algoritmo newscast se implementa mediante el concepto de colectivo, que es un grupo de entidades que trabajan conjuntamente para alcanzar un objetivo no necesariamente común. Cada entidad posee una vista parcial del colectivo, que incluye parte de una base de datos de contribuciones y comandos.

El rol de un miembro del colectivo puede ser de observador, contribuyente o controlador. Un observador puede leer las bases de datos de contribuciones y comandos, un contribuyente puede escribir sobre la base de datos de contribuciones

y un controlador puede escribir sobre la base de datos de comandos. Un miembro puede realizar varios roles simultáneamente.

Las vistas de cada nodo varían mediante una operación periódica en la que dos nodos construyen aleatoriamente dos nuevas vistas locales a partir de la unión de sus dos vistas previas.

El paquete también define el concepto de nodo, que es una base que participa del algoritmo newscast siendo un observador, un contribuyente y un controlador.

4.3.1. ContributionBox

Un `ContributionBox` es una estructura de datos que implementa `Serializable` con tres campos que implementa lo que en el algoritmo se suele llamar contribución.

`long: timestamp` - El momento de creación del `ContributionBox`.

`Address: contributor` - La dirección del creador del `ContributionBox`.

`Object: contribution` - Una estructura de datos disponible para la aplicación.

`ContributionBox` sólo implementa `Serializable`, pero sería conveniente que implementase también `Comparable`, como veremos más adelante.

4.3.2. Contributor

Esta interfaz define la funcionalidad básica para aportar información a un colectivo. Es importante destacar que es el colectivo y no el contribuyente quién decide cuándo se aporta la información.

`ContributionBox: getContribution()` - Esta función es llamada por el colectivo cuando sea preciso actualizar la contribución.

`String: getName()` - Los contribuyentes deben tener un nombre único.

4.3.3. Observer

Define la funcionalidad necesaria para leer la información presente en el colectivo.

`void: collectiveUpdated(ContributionBox peer)` - El colectivo llama esta función cada vez que sucede una operación de mezcla. El parámetro es el par con quien se ha realizado.

`Address[]: getPeerAddresses()` - Se llama cuando no se conocen pares. Un observador necesita estar conectado para tener sentido.

`IRequest: fireMessage(Address m, String type, Object o)` - El observador lo necesita para pedir información.

En opinión del autor no se entiende por qué `fireMessage()` forma parte de esta interfaz.

4.3.4. Controller

Define la funcionalidad necesaria para introducir comandos en el colectivo. Como siempre, es el colectivo el que le pide los comandos al controlador en el momento oportuno.

`Set: getCommands()` - Este método es llamado por el colectivo cuando es el momento de introducir comandos.

4.3.5. IDRM

Interfaz utilizada por los agentes para conseguir información del colectivo.

`ContributionBox: getContribution(String name)` - Devuelve la contribución indexada por `name`, si esta presente en la caché.

`ContributionBox: getNewestContribution()` - Devuelve la contribución del último par con el que se comunicó el nodo.

`ContributionBox: getContributions()` - Devuelve todas las contribuciones en la caché en una lista inmodificable.

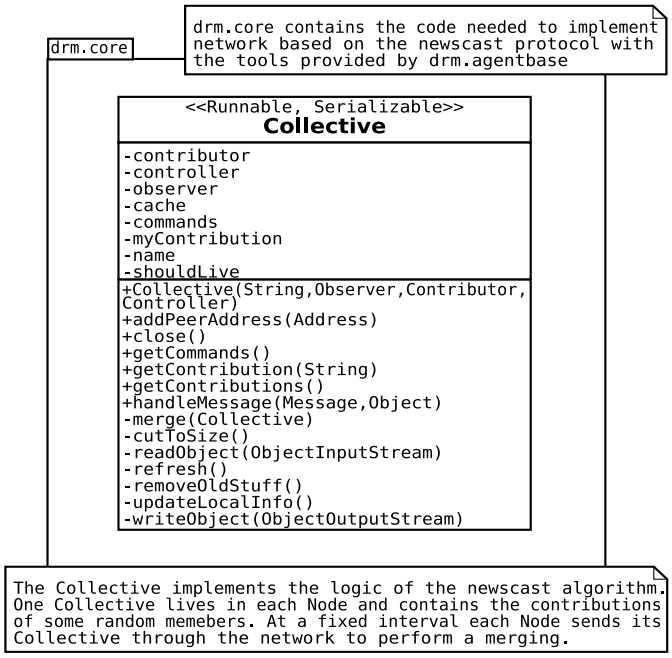

Figura 4.8: Diagrama parcial de clases relativo a `drm.core.Collective`

4.3.6. Collective

Esta clase implementa un proxy a un colectivo, es una vista parcial de la entidad compuesta por todos los contribuyentes, observadores y controladores. Los miembros concretos que se ven en esta vista local se determinan dinámicamente mediante el algoritmo newscast. El diagrama de clases relativo a `Collective` se expone en la figura 4.8.

Para unirse a un colectivo no es suficiente con construir un objecto `Collective` con `Collective c = new Collective("mycollective",this)`, también es necesario redirigir mensajes del tipo "`collectiveUpdate-mycollective`" implementando `handleMessage(Message,Object)`.

Estructuras de datos y variables.

`Map: cache` - Almacena las contribuciones conocidas. Los elementos son objetos `ContributionBox` y las claves los nombres de sus creadores. Nunca contiene una contribución del contribuyente local.

`ContributionBox: myContribution` - Almacena la contribución del contri-

buyente local.

`Map: commands` - Almacena los comandos activos. Estos comandos han de ser ejecutados por miembros del colectivo.

`Observer: observer` - El observador al que sirve el colectivo.

`Contributor: contributor` - El contribuyente al que sirve el colectivo. Sus contribuciones son propagadas por la red.

`Controller: controller` - El controlador al que sirve el colectivo. Sus comandos son distribuidos por la red e interpretados por los demás pares.

`boolean: shouldLive` - Cuando es puesto a falso el hilo de ejecución termina. Se controla el paralelismo en su acceso mediante la marca `volatile`.

`String: name` - El nombre del colectivo, también denominado `job` o `group` en ocasiones.

Constantes

`long: CONTRIBUTION_TIMEOUT` - Las contribuciones cuya edad en milisegundos sea mayor a esta constante son eliminadas.

`long: COMMAND_TIMEOUT` - Los comandos cuya edad en milisegundos sea mayor a esta constante son eliminados.

`long: MAX_CACHE_SIZE` - Número máximo de contribuciones que puede contener la caché.

`long: MAX_COMMANDS_SIZE` - Número máximo de comandos que puede contener `commands`.

`long: REFRESHRATE` - Intervalo de tiempo entre dos intentos de iniciar una comunicación de mezcla.

El algoritmo newscast

Esta sección muestra la implementación del algoritmo newscast, explicado con detalle en la sección 3. El algoritmo puede ser iniciado de dos modos: periódicamente desde el método `run()` que llama al método `refresh()`, o como respuesta a la iniciativa de otro nodo mediante el método `handleMessage(Message m, Object o)`. Los principales métodos involucrados se exponen a continuación:

`refresh()` - Realiza el bucle principal del algoritmo newscast.

Refresca las contribuciones y comandos locales con `updateLocalInfo()`.

Crea una permutación aleatoria de pares y trata de comunicarse con uno de ellos. Si lo consigue intercambian sus colectivos.

Si existen, elimina su propia contribución y la del par tanto de su propio colectivo como del recibido del par.

Convierte las direcciones locales en globales y añade la contribución del par a la caché recibida mediante `repairSenderAddress(Collective c, Address sender)`.

Mezcla los dos colectivos en uno sólo mediante `merge(Collective c)`.

Avisa a los observadores de que la caché ha cambiado mediante el método `observer.collectiveUpdated()`.

`updateLocalInfo()` - Pide al contribuyente que sirve a este colectivo la última versión de su contribución. A los controladores les pide sus comandos.

`repairSenderAddress(Collective c, Address sender)` - Añade la contribución del contribuyente del colectivo `c` a la caché también de `c` con la dirección de `sender`.

`merge(Collective c)` - Si existe en la caché `c` alguna entrada con el mismo contribuyente que una contribución en la caché local, y la contribución de la caché c es más reciente, sustituye la contribución antigua en la caché local por la contribución reciente de la caché `c`. Hace lo mismo con los comandos. Por último llama a `cutToSize()`.

`cutToSize()` - Elimina entradas al azar de la caché y de los comandos hasta que su talla no sea mayor de `MAX_CACHE_SIZE` y `MAX_COMMANDS_SIZE` respectivamente.

writeObject y readObject

Las funciones de serialización del colectivo se han reescrito para poder realizar algunas comprobaciones y ajustes sobre el timestamp. Por ello las estructuras que deben ser serializadas están también marcadas como `transient`.

Métodos públicos

Otros métodos presentes en `Collective` son los siguientes:

`run()` - Periódicamente y mientras no se le pida que pare realiza las funciones del nodo.

Llama al *garbage collector* de Java.

Utiliza `removeOldStuff()` para eliminar contribuciones y comandos demasiado viejos.

Llama a `refresh()` para ejecutar el algoritmo newscast.

Si no conoce pares utiliza los medios a su alcance para descubrir alguno nuevo.

`void: close()` - Le da a `shouldLive` el valor de falso y termina la actividad del colectivo.

`void: addPeerAddress(Address a)` - Sirve para darle al colectivo la dirección de otra base. Se utiliza para conectarlo inicialmente a la red.

`boolean: handleMessage(Message m, Object o)` - Solo sabe interpretar mensajes de tipo "`collectiveUpdate`"+`name`. Estos mensajes son peticiones de mezcla.

`ContributionBox: getContribution(String name)` - Devuelve la contribución de un determinado actor si se encuentra presente en la caché.

`List: getContributions()` - Devuelve todas las contribuciones de la caché en una lista inmodificable.

`List: getCommands()` - Devuelve los comandos activos en una lista inmodificable.

Cuando un agente residente en un nodo acceda al colectivo, lo hace a través de una clase `Firewall` que implemente la interfaz `IDRM`.

4.3.7. Node

Un nodo es una implementación que extiende a `Base` y que incorpora un objeto `Collective`. El colectivo se ejecuta en su propio hilo y se comunica periódicamente con otros nodos para mantener activo el algoritmo newscast. El nodo accede al colectivo para tener un repositorio de direcciones a las que acceder y utilizar la red para sus propios objetivos. Su diagrama de clases se muestra en la figura 4.9.

`getContribution()` - Crea y devuelve un `NodeContribution`.

`getContribution(String name)` - Busca una contribución de `name` en la caché de la instancia local del colectivo.

`getContributions()` - Devuelve todas las contribuciones presentes en la caché de la instancia local del colectivo en una lista inmodificable.

`getNewestContribution()` - Devuelve la contribución de `lastPeer`.

`collectiveUpdated()` - Actualiza `lastPeer` y ejecuta localmente los comandos presentes en el colectivo mediante `invokeCommandLocally()`.

`fireMessage()` - Llama a `super.fireMessage()`.

`handleMessage()` - Reacciona a varios tipos de mensaje

`"collectiveUpdate"+getJob()` - Si recibe un mensaje de otro nodo del mismo colectivo que quiere mezclar cachés lo redirige a la instancia local del colectivo.

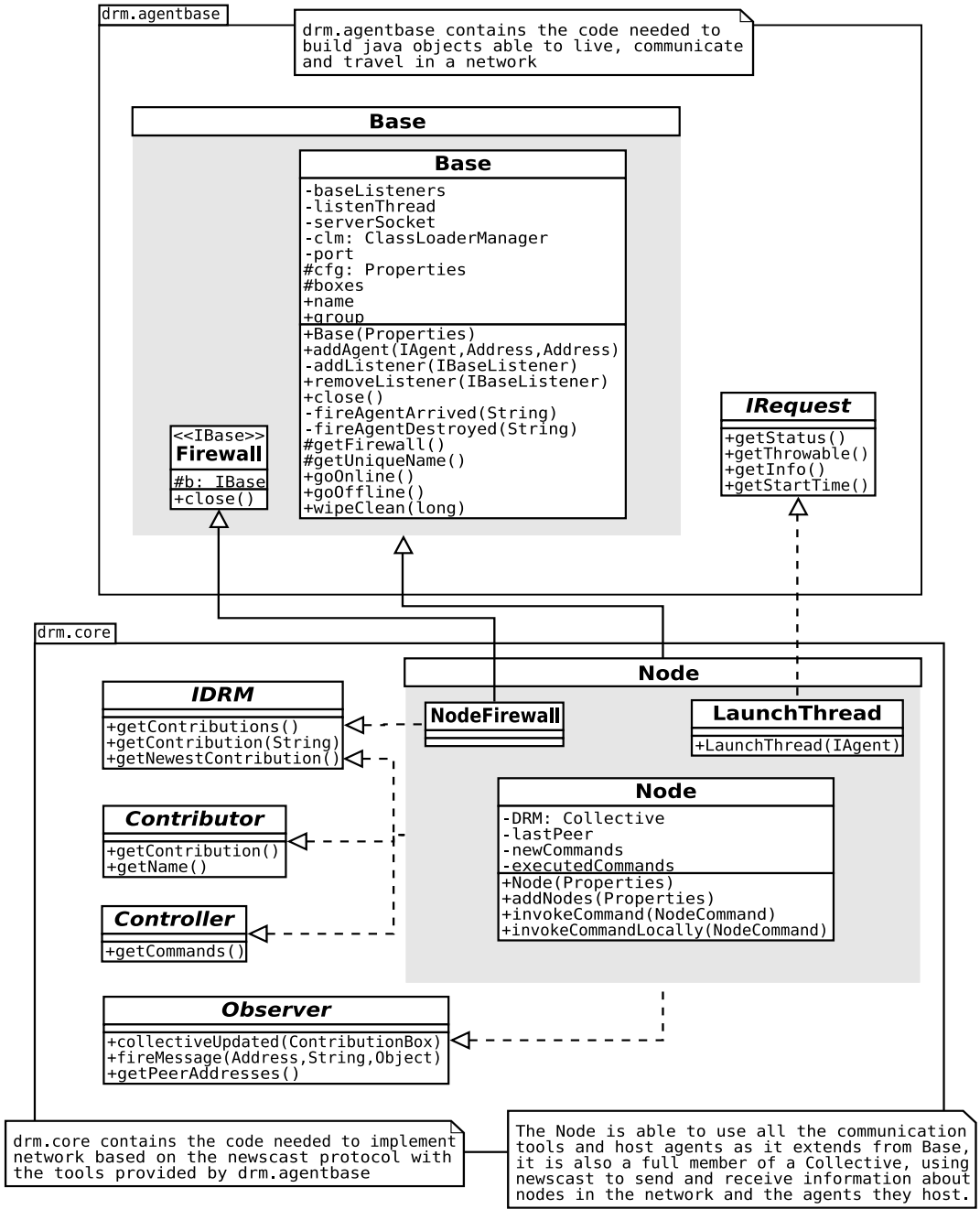

Figura 4.9: Diagrama parcial de clases relativo a `drm.core.Node`

"addCommand" - Añade un comando al colectivo.

"getInfo" - Devuelve al emisor algo de información de la máquina y el colectivo locales.

"getStatus" - Devuelve al emisor "running" si el nodo funciona.

`getCommands()` - Este método es invocado por el colectivo para recoger los comandos que el nodo ha ido almacenando en `newCommands`.

`invokeCommandLocally(NodeCommand command)` - Ejecuta un comando. Sólo reconoce el tipo de comando `CLEANALL`, que resetea el nodo mediante la función `super.wipeclean()`.

`invokeCommand(NodeCommand command)` - Incorpora un comando a la estructura `newCommands`. El colectivo imnporta estos comandos invocando el método `getCommands()` en el nodo.

`launch()` - Implementa el modo "RANDOM" para enviar un mensaje a un par aleatorio, si se emplea otro modo llama a `super.launch()`.

`onArrival()` - Crea la instancia local del colectivo y lo conecta a la red.

`addNodes(Properties cfg)` - Añade a la instancia local del colectivo las direcciones de los nodos que encuentre en `cfg`.

`getFirewall()` - Devuelve un firewall de tipo `NodeFirewall`, que extiende al `Firewall` de `Base` e implementa `IDRM`.

`close()` - Detiene y cierra el hilo de ejecución del colectivo y después el del nodo.

4.3.8. ContributorAgent

Un `ContributorAgent` es un agente que ejecuta el algoritmo newscast y posee una instancia de un `Collective`. Implementa las interfaces `Contributor` y `Observer`. El colectivo de este agente utiliza los servicios de la `Base` para enviar y recibir mensajes. Es interesante observar que si un `ContributorAgent` se ejecuta sobre un `Node` los colectivos de ambos son totalmente independientes.

4.4. Inicio de un nodo DRM

En el paquete drm.server se encuentran las clases que hay que invocar para iniciar una instancia de la Distributed Resource Machine, en esta sección se comentará el funcionamiento de `drm.server.NewNode`, desarrollado por el autor a partir de código previamente contenido en otras clases del paquete. Esta clase es invocada desde línea de comandos con `drmnode.new` e interpreta varios parámetros para iniciar un nodo DRM.

-p: Indica el puerto en el que escucha el nodo.

-g: Indica el nombre del colectivo al que pertenece el nodo.

-n: Indica una serie de nodos en el formato ip:puerto a los que conectarse.

-v: Indica el nivel de verbosidad del nodo.

-r: Ejecuta la clase indicada según el formato `fichero.jar\!paquete.clase`

-h: Muestra la ayuda.

La clase posee dos métodos respectivamente para iniciar un nodo y para iniciar un agente alojado en él.

`run()` - Este método lee los parametros de creación del nodo, con los que lo crea y lo conecta a la red. Si así se le indica llama a `startScript()` para crear un agente inicial.

`startScript(String scriptName, Node startOn, String[] args)` - Esta función se llama si se le indica al nodo mediante un argumento que después de iniciarse debe crear un agente definido por el usuario. Se obtienen de los argumentos el `JobClassLoader` y el nombre de la clase agente y después mediante reflexión se obtiene el constructor de la clase agente y se instancia.

Capítulo 5

Tutorial de uso de DRM

Es importante proporcionar la información necesaria para que el propio lector pueda desarrollar sus propias aplicaciones. Este tutorial ha sido compilado a partir de los ejemplos originales para DRM creados por Jelasity [18] y de tres ejemplos adicionales creados por el autor que explican detalles no cubiertos con anterioridad.

5.1. La primera aplicación

El primer programa que crearemos sera un agente DRM que tras iniciarse nos saluda con un "Hello World!" y desaparece. Este ejemplo es el primero del tutorial de Jelasity [18].

5.1.1. Crear un nodo DRM

Para ejecutar una aplicación DRM es necesario tener un nodo DRM en ejecución. Una red DRM es un conjunto de nodos conectados entre sí. Para ejecutar una primera aplicación crearemos una red DRM lo más sencilla posible, consistente en un único nodo. Esto se puede realizar mediante varios de los ejecutables proporcionados con DRM, en este tutorial hemos escogido `drmnode.new`, que inicia un nodo DRM mediante la clase `drm.server.NewNode`.

Cuando se desarrollan aplicaciones que pueden fallar, es importante evitar que éstas se extiendan a otros nodos no controlados por el usuario. Esto es fácil de hacer ya que DRM está diseñado para crear tantas redes separadas como sea necesario,

para ello basta con indicarle a `drmnode.new` que queremos crear un red propia a la que le daremos un nombre que sepamos que nadie más ha escogido.

```
drmnode -g migrupo
```

5.1.2. Ejecutar la aplicación

El modo más conveniente de ejecutar una aplicación en una red DRM es encapsulándola en un fichero Java comprimido en formato jar. Para ejecutar el ejemplo 1 comprimimos los dos ficheros que lo componen en un archivo llamado `tutorial1.jar`, y ejecutaremos un nodo DRM indicándole la ruta hasta el archivo y dentro la ruta hasta la clase que inicia la aplicación.

```
drmnode -g migrupo -r tutorial1.jar\!hwjob.Launch
```

Se debería ver un texto inicial proveniente del nodo que se inicia y el texto "Hello World!" producido por el agente descrito en la sección 5.8.

5.1.3. Seguridad

Por defecto las aplicaciones tienen derechos de *sandbox*, no se les permite escribir a disco o iniciar conexiones a través de la red, por ejemplo. Nuestro experimento simple contenido en `tutorial1.jar` no lo necesita, pero en muchos casos sí será deseable.

Para darle a la aplicación derechos de escritura sin hacer cambios mayores en DRM basta con colocar el archivo .jar en el directorio desde el que se inicia el nodo, aunque esto no le dará permisos en ningún otro sitio.

5.1.4. La aplicación desde dentro

Nuestro trabajo se compone de dos clases comprimidas en un archivo jar del siguiente modo:

```
hwjob/HelloWorld.class
Launch.class
```

Aquí la clase `Launch`, presente en la sección 5.7, es la que inicia la ejecución, creando y lanzando un agente presente en la sección 5.8. El lanzador implementa el

constructor estándar y la interfaz Runnable, que son un requisito de todo lanzador DRM.

El método `run()` construye el agente y usa el método `launch()` del nodo para iniciarlo. El segundo parámetro con valor "`DIRECT`" indica que el agente será lanzado al nodo cuya dirección aparece en el tercer parámetro. Al ser éste último nulo el agente será lanzado en el nodo desde el que se invocó a `Launch`.

El único requerimiento de un agente es que debe implementar la interfaz `IAgent`. La clase `Agent` es una clase de conveniencia que proporciona implementaciones por defecto para la mayoría de los métodos de `IAgent`. En nuestro caso hemos implementado tan sólo el método `run()`, que escribe "Hello World!" por la salida estándar.

Para detenerse, un agente debe llamar explicitamente `suicide()`, que resultará en la eliminación del agente de su entorno. Si no se llama este método el agente permanece vivo, consumiendo recursos, aunque permanece pasivo. Estar vivo significa que el agente aún puede recibir y manejar mensajes.

5.2. Segunda aplicación: Movilidad

Ilustramos la movilidad de los agentes mediante una aplicación que consiste en un agente que salta entre diferentes nodos mientras cuenta el número de saltos.

Para este ejemplo necesitamos una red DRM de al menos dos nodos. Esto se consigue ejecutando la aplicación `drmnode.new` dos veces en terminales distintas, e indicando mediante el parámetro `-n` la dirección de uno cualquiera de ellos al otro. Si ambos nodos se inician en la misma computadora es necesario especificar con el parámetro `-p` puertos distintos de escucha para cada uno.

```
drmnode.new -g test -p 10101
drmnode.new -g test -p 10102 -n localhost:10101
```

Mediante este procedimiento es posible crear una red DRM de cualquier tamaño, basta con que todos los nodos formen una red conexa al iniciarse para que el algoritmo newscast construya una red como la descrita en el capítulo 3.

La clase lanzadora, expuesta en la sección 5.9, es casi idéntica a la del primer ejemplo. El código del agente, en la sección 5.10, utiliza la interfaz `IDRM` (sección

4.3.5) para conseguir información de otros nodos en la red. El agente le pide a su base que lo envíe a uno de estos nodos mediante `base.dispatchAgent(...)`. El agente posee un campo donde almacena el número de veces que ha saltado a un nuevo nodo, el valor de este campo se preserva de acuerdo a los principios de serialización de Java cuando el agente viaja a un nuevo nodo.

5.3. Tercera aplicación: Comunicación

Esta aplicación consiste en un agente que crea otro agente, le envía un mensaje e imprime la respuesta. La clase lanzadora, en la sección 5.11, tiene la misma estructura simple de los ejemplos anteriores. En la sección 5.12 se encuentra el código del agente. El primer agente en ser creado tiene el nombre `Talker.test<time>.1`, el segundo tiene el mismo nombre pero terminado en 2. La primera línea del método `run()` comprueba cuál es el agente en ejecución, puesto que el segundo es pasivo y no realiza acciones mediante este método, sino que sólo responde a mensajes. El primer agente lanza al segundo a un nodo vecino aleatorio mediante `base.launch("RANDOM",...)`.

El envío de mensajes se realiza mediante `fireMessage(...)` que entrega el mensaje a la base para que lo encamine hacia su destino final. El agente destino reescribe el método `handleMessage(...)` para reconocer el mensaje y responder por el mismo canal de comunicación con `m.setReply(...)`. No importa si el agente es activo o pasivo, siempre tiene la oportunidad de reaccionar a los mensajes que recibe mediante este método.

En este ejemplo se hace un completo uso de la interfaz `drm.agentbase.IRequest`. Los métodos de comunicación asíncronos utilizan esta interfaz, que puede proporcionar información del estado de la comunicación mediante `irequest.getStatus()`. Si se quiere saber con seguridad si una comunicación tuvo éxito se debe esperar a que termine y comprobar su estado, como se hace en este ejemplo.

Las operaciones asíncronas pueden devolver información que puede ser recuperada mediante el método `irequest.getInfo(clave)` si se dispone de la clave adecuada. En el caso de `base.launch("RANDOM",...)` se puede recuperar la dirección a la

que fue enviado el agente con `irequest.getInfo(“address”)`. En el caso del envío de mensajes se puede obtener una respuesta mediante `irequest.getInfo(“reply”)`.

5.4. Cuarta aplicación: Uso de direcciones

La manera de comunicarse con actores concretos de una red DRM se ilustra mediante el agente en la sección 5.14. Este ejemplo fue desarrollado por el autor de un modo similar al ejemplo anterior, en el que un agente inicia un segundo y se comunican entre sí. La diferencia estriba en que en este ejemplo se envía el segundo agente a un nodo concreto y ambos se comunican de una manera más compleja.

En esta ocasión nuestros agentes extienden `ContributorAgent` para aprovechar que este agente sabe mantener una dirección denominada `root` para estar conectado a su creador. El lanzador (sección 5.13) crea el agente raíz mediante `new Killer(“killjob”, “root”)`. Esto quiere decir que el agente creado se llamará `Killer.killjob.root`.

Este agente raíz crea un segundo agente mediante `new Killer(“killjob”, “child”, new Address(this.name, -1, null))`. Este segundo agente se llamará `Killer.killjob.child` y dispondrá de la dirección `new Address(this.name, -1, null)` para comunicarse con el agente raíz. Ésta es una dirección local, puesto que sólo se ha proporcionado el nombre de un agente y no su puerto ni su IP. Se transformará en una dirección global cuando el agente llegue su destino, como se especifica en el código de `ContributorAgent`.

Para lanzar a su destino al agente hijo, el agente raíz obtiene la dirección de un nodo vecino mediante `getDRM().getNewestContribution().contributor`. Este comando indica que se quiere acceder a la información del colectivo (`getDRM()`), recuperar la información de la última operación de mezcla (`getNewestContribution()`) y por último la dirección del vecino con el que se realizó (`contributor`). Esta dirección se utiliza para lanzar el agente con el modo `DIRECT` con `base.launch(“DIRECT”, child, target)`.

Ahora queremos mandarle al segundo agente un mensaje desde el primero que lo haga detener su ejecución. Cuando se desea mandar un mensaje a un agente en otra

base es necesario saber su dirección completa. Esta dirección esta compuesta por la IP y el puerto de la base que lo aloja más el nombre del agente, construido como `new Address(target.getHost(), target.port, child.name)` en este caso.

Estos agentes son capaces de interpretar dos tipos de mensajes, `INFOMESSAGE` y `SUICIDEMESSAGE`, como se ve en el método `handleMessage(...)`. El primero simplemente se muestra por pantalla al ser recibido, mientras que el segundo hace al agente que lo interpreta detener la ejecución y mandar una confirmación. Un mensaje de un tipo no soportado haría devolver falso a este método para que el agente que envió el mensaje fuera avisado del error.

Es muy importante diferenciar el nodo (que es iniciado desde línea de comandos con `drmnode.new`) del agente (que puede ser transferido a través de la red). Si se le envía un mensaje a un nodo o base con la esperanza de que se lo transfiera al agente pero sin indicarle el nombre concreto del mismo obtendremos un mensaje de error y el mensaje se perderá. En este ejemplo lo que se hace es guardar una copia local del agente que enviamos para poder saber su nombre, a diferencia de ejemplos anteriores.

5.5. Quinta aplicación: Uso del colectivo para direccionamiento

El siguiente ejemplo, desarrollado por el autor y expuesto en las secciones 5.15 y 5.16, muestra como crear una red de `ContributorAgent` que se extiende sobre todo un grupo de nodos. También muestra que en este caso existen dos redes newscast superpuestas, la de los nodos y la de los agentes. Por último muestra como es posible mandar objetos serializables Java de un agente a otro.

El agente raíz iniciado por el lanzador obtiene todas las contribuciones de la instancia local del colectivo presente en su nodo con `getDRM().getContributions()`. El resultado es una `List` de `ContributionBox`, donde cada uno de estos es un contenedor con la dirección del nodo que lo creó inicialmente, el momento en el que lo hizo y una lista con los agentes que acoge. Por ahora sólo nos interesan las direcciones de los nodos y les enviamos un agente a cada uno.

A continuación queremos que los agentes se manden información entre ellos, usando el colectivo que crean independientemente del colectivo de los nodos. Para ello esperamos un periodo prudencial de tiempo para darles tiempo a realizar unas cuantas operaciones de mezcla y creamos una serie de estructuras a enviar, en este caso tablas hash con números aleatorios, pero se puede enviar cualquier cosa que implemente `Serializable`.

Utilizamos el método `collective.getContributions()` para conseguir la lista de los agentes en el colectivo. Los datos que recibimos tienen el mismo formato que cuando hicimos `getDRM().getContributions()` pero se refieren al colectivo creado por los agentes.

Una vez que tenemos las direcciones y los datos, para hacer los envíos basta con invocar `fireMessage(address, type, data)`. El parámetro `type` es un `String` que sirve para identificar el mensaje en el destino.

El método `handleMessage(Message, Object)` contiene la lógica necesaria para reaccionar al mensaje. El parámetro `Object` contiene lo que se introdujo en el otro extremo como `data`, ya deserializado. En `Message` están el resto de los elementos, como las direcciones del emisor y el receptor, el tipo del mensaje o el objeto sin deserializar.

Es muy importante notar la diferencia entre invocar `getDRM()`, lo que nos permite acceder al colectivo y a la red del nodo, o acceder a la variable `collective`, que es el mismo tipo de objeto pero permite acceder al colectivo y a la red del agente, siendo ambas redes totalmente independientes.

5.6. Sexta aplicación: Difusión de información mediante el colectivo

Este ejemplo fue desarrollado por el autor para mostrar del modo más simple posible como utilizar el colectivo y difundir información mediante el algoritmo newscast. El código necesario se encuentra en las secciones 5.17 y 5.18.

Si se observa el método `run()` se ve que los agentes son pasivos, ya que el código de este método sólo afecta al agente raíz que lo utiliza para crear la red de

agentes. Y el método `handleMessage(...)` no se ha reescrito tampoco, así que el comportamiento de los agentes difiere bastante de los ejemplos anteriores.

Cada agente mantiene un elemento de información, su contribución. Esta contribución es requerida periódicamente por el colectivo con `getContribution(...)` y encapsulada en un `ContributionBox`. Este método devuelve la contribución del agente, que el colectivo toma y difunde por la red. El usuario debe sobreescribir este método para que devuelva la información que desee.

Cada vez quela instancia local del colectivo en el agente sufre una modificación se llama al método `collectiveUpdated(...)` en el agente, que puede aprovechar para reaccionar a las informaciones que otros agentes han contribuido al colectivo. Para recuperarlas basta con invocar a `collective.getContributions()`.

5.7. Primera aplicación: Launch

```
import drm.core.Node;
import hwjob.HelloWorld;

public class Launch implements Runnable {

    private final Node node;

    public Launch( Node node ) {this.node = node;}

    public void run() {
        node.launch(
            "DIRECT",
            new HelloWorld("test+"+System.currentTimeMillis(),"1"),
            null );
    }
}
```

5.8. Primera aplicación: hwjob.HelloWorld

```
package hwjob;

import drm.agents.Agent;

public class HelloWorld extends Agent {

    /** calls super constructor */
    public HelloWorld( String job, String name ) {
        super( "Helloworld", job, name );
    }

    /** prints "Hello World" and exits after waiting 5s*/
    public void run() {
        System.out.println("Hello world!");
        try { Thread.currentThread().sleep(5000); }
        catch( Exception e ) {}
        suicide();
    }
}
```

5.9. Segunda aplicación: Launch

```
import drm.core.Node;
import jumperjob.Jumper;

public class Launch implements Runnable {

    private final Node node;

    public Launch( Node node ) {this.node = node;}

    public void run() {
        node.launch(
            "DIRECT",
            new Jumper("test+"+System.currentTimeMillis(),"1"),
            null );
    }
}
```

5.10. Segunda aplicación: jumperjob.Jumper

```
package jumperjob;

import drm.agents.Agent;
import drm.core.*;

public class Jumper extends Agent {

    /** jump counter. Its value is serialized so it is
    * preserved while travelling to other nodes. */
    private int jumps = 0;

    /** calls super constructor */
    public Jumper( String job, String name ) {
        super( "Jumper", job, name );
    }

    /** Jumps to another random node 3 times. The waiting periods are
    * not necessary, they are included only to slow it down so it can
    * be followed by a human.*/
    public void run() {

        ContributionBox cb = getDRM().getNewestContribution();

        if( cb == null ){
            System.err.println("No nodes to jump to");
            try { Thread.currentThread().sleep(1000); }
            catch( Exception e ) {}
            suicide();
        }
        else if( jumps++ < 3 ){
```

```
            try { Thread.currentThread().sleep(1000); }
            catch( Exception e ) {}
            System.err.println("Jumping to "+cb.contributor);
            base.dispatchAgent(name,cb.contributor);
        }
        else{
            System.err.println("Got tired of jumping around...");
            try { Thread.currentThread().sleep(1000); }
            catch( Exception e ) {}
            suicide();
        }
    }
}
```

5.11. Tercera aplicación: Launch

```
import drm.core.Node;
import talkjob.Talker;

public class Launch implements Runnable {

    private final Node node;

    public Launch( Node node ) {this.node = node;}

    public void run() {
        node.launch(
            "DIRECT",
            new Talker("test+"+System.currentTimeMillis(),"1"),
            null );
    }
}
```

5.12. Tercera aplicación: talkjob.Talker

```
package talkjob;

import drm.agents.Agent;
import drm.agentbase.*;

public class Talker extends Agent {

    /** calls super constructor */
    public Talker( String job, String name ) {
        super( "Talker", job, name );
    }

    /** Launches another agent, sends it a message and prints the
    * reply. Does not suicide to allow some manual testing.*/
    public void run() {

        if( name.endsWith("2") ) return; // this agent is passive

        IRequest r = base.launch(
            "RANDOM",new Talker( job, "2" ), null );
        while( r.getStatus() == IRequest.WAITING ){
            try { Thread.currentThread().sleep(100); }
            catch( Exception e ) {}
        }
        if( r.getStatus() != IRequest.DONE ) return;

        Address a = (Address)r.getInfo("address");
        r = fireMessage( a, "test", "How are you?" );
        while( r.getStatus() == IRequest.WAITING ){
            try { Thread.currentThread().sleep(100); }
```

```
            catch( Exception e ) {}
        }
        if( r.getStatus() == IRequest.DONE )
            System.out.println( "Answer: "+r.getInfo("reply") );
    }

    /** Handles message type "test" answering always with the String
    * object "Fine thanks."*/
    public boolean handleMessage( Message m, Object object ) {
        if( super.handleMessage( m, object ) ) return true;

        if( m.getType().equals("test") ){
            System.out.println("Received: "+object);
            m.setReply("Fine thanks.");
            return true;
        }

        return false;
    }
}
```

5.13. Cuarta aplicación: Launch

```
import killjob.Killer;
import drm.core.Node;

public class Launch implements Runnable{

    private final Node node;

    public Launch(Node node){this.node = node;}

    public void run(){
        node.launch(
            "DIRECT",
            new Killer("killjob","root"),
            null
        );
    }
}
```

5.14. Cuarta aplicación: killjob.Killer

```
package killjob;

import drm.agents.*;
import drm.agentbase.*;

/** By extending ContributorAgent we enable the agent to
 * automatically know a root agent to communicate with. Also
 * it will belong to a collective, not used in the example.
 * This example illustrates messaging to particular agents. */
public class Killer extends ContributorAgent {
    public static final long serialVersionUID = 7894561344352L;

    public static final String INFOMESSAGE = "info";
    public static final String SUICIDEMESSAGE = "suicide";

    public Killer(String job, String name, Address root){
        super("Killer", job, name, root);
    }

    public Killer(String job, String name){
        super("Killer", job, name, null);
    } // root address is null, thus this is a root agent

    /** The root agent will spawn one agent to some node, which
     * will be ordered to suicide afterwards. */
    public void run(){
        if(name.endsWith("child")){
            System.out.println("Child initializing");
            fireMessage(root,INFOMESSAGE,"I'm alive!");
            return;
```

```
}

/* We choose one node from the DRM net, the last node
 * which communicated with us. */
Address target =
    getDRM().getNewestContribution().contributor;
/* We create the agent locally to get access to its name,
 * root IP Address (null) and port will be automatically
 * corrected later upon arrival. */
Killer child = new Killer(job, "child",
            new Address(null,-1,this.name));
IRequest r = base.launch("DIRECT", child, target);

while(r.getStatus() == IRequest.WAITING){
    try{Thread.sleep(1000);}
    catch(Exception e){System.err.println("Exception: " + e);}
}
if(r.getStatus() != IRequest.DONE)
    System.err.println("There was an error " +
        "sending an agent to " + target.name);

System.out.println("Die!");
/* The address to which we must post messages is the one
 * created with the host and port from the recipient node,
 * plus the name of the launched agent (There can be many
 * agents in each node). */
r = fireMessage(
        new Address(target.getHost(),target.port,child.name),
        SUICIDEMESSAGE, "Die!");

while(r.getStatus() == IRequest.WAITING){
```

```
            try{Thread.sleep(100);}
            catch(Exception e){System.err.println("Exception: " + e);}
        }

        if(r.getStatus() == IRequest.DONE)
        System.out.println("Request obeyed");
        suicide();
    }

    /** Prints each message received. If the type of the message
     *  is "suicide", the agent suicides. If the message is not
     *  recognized by this agent, this function must return false.*/
    public boolean handleMessage(Message m, Object o){
        if(!super.handleMessage(m, o)){
            if(m.getType().equals(INFOMESSAGE)){
                System.out.println("Received: " + o);
            }
            else if(m.getType().equals(SUICIDEMESSAGE)){
                System.out.println("Received: " + o);
                fireMessage(m.getSender(),INFOMESSAGE,"I'll be back.");
                suicide();
            }
            else return false;
        }
        return true;
    }
}
```

5.15. Quinta aplicación: Launch

```
import sendjob.Sender;
import drm.core.Node;

public class Launch implements Runnable{

    private final Node node;

    public Launch(Node node){
        this.node = node;
    }

    public void run(){
        node.launch("DIRECT", new Sender("sendjob","root"),null);
    }
}
```

5.16. Quinta aplicación: sendjob.Sender

```
package sendjob;

import drm.agents.*;
import drm.agentbase.*;
import drm.core.*;

import java.util.*;

/** This example illustrates how serializable objects can be
 * sent through messaging. */
public class Sender extends ContributorAgent {
    public static final long serialVersionUID = 78945624352L;

    public static final String DATAMESSAGE = "data";

    public Sender(String job, String name, Address root){
        super("Sender", job, name, root);
    }

    public Sender(String job, String name){
        super("Sender", job, name, null);
    }

    /** Children will be sent by root to every other node in
     * the DRM network. Any agent will create a hashtable with
     * random numbers and sent it to every other node in the
     * collective.
     * The DRM network is composed by the nodes started previously
     * without any job, and the collective is formed by the agents
     * of a kind hosted in that DRM network.*/
```

```
public void run(){
    Address target;
    IRequest request;
    Iterator<ContributionBox> contributions;

    // Get all nodes in the DRM and send them "Sender" agents.
    if(root == null){
        contributions = getDRM().getContributions().iterator();
        while(contributions.hasNext()){
            target = contributions.next().contributor;
            Sender child = new Sender("Sender",null,
                    new Address(null,-1,this.name));
            request = base.launch("DIRECT", child, target);
            while(request.getStatus() == IRequest.WAITING){
                try{Thread.sleep(1000);}
                catch(Exception e){
                    System.err.println("Exception: " + e);
                }
            }
            if(request.getStatus() != IRequest.DONE)
                System.err.println("There was an error " +
                    "sending an agent to " + target.name);
        }
    }

    /* We must wait a prudential time so the agents arrive
     * and connects between them to create the collective. */
    System.out.println("Waiting 30s to send objects...");
    try{Thread.sleep(30000);}
    catch(Exception e){System.err.println("Exception: " + e);}
```

```
System.out.println("Creating object to send...");
Hashtable<String,String> data =
    new Hashtable<String,String>();
for(int i=0; i<5; i++) data.put(
        "" + Math.random(),"" + Math.random());
Iterator<String> keys = data.keySet().iterator();
while(keys.hasNext()){
    String key = keys.next();
    System.out.println(key + " - " + data.get(key));
}

/* We can access to the agents we sent using the collective
 * field. The data returned is the same type than that we get
 * with getDRM(), but they refer to different things.*/
contributions = collective.getContributions().iterator();
while(contributions.hasNext()){
    target = contributions.next().contributor;
    System.out.println("Sending data to " + target.name);
    request = fireMessage(target, DATAMESSAGE, data);
    while(request.getStatus() == IRequest.WAITING){
        try{Thread.sleep(1000);}
        catch(Exception e){
            System.err.println("Exception: " + e);
        }
    }
    if(request.getStatus() != IRequest.DONE)
        System.err.println("There was an error " +
            "sending the data to " + target.name);
}

// Let's wait some for the messages that could arrive yet.
```

```
        System.out.println("Waiting 30s to exit...");
        try{Thread.sleep(30000);}
        catch(Exception e){System.err.println("Exception: " + e);}

        suicide();
    }

    /** If the message is recognizable as a data message, print it.*/
    public boolean handleMessage(Message message, Object object){
        if(!super.handleMessage(message, object)){
            if(message.getType().equals(DATAMESSAGE)){
                System.out.println("Data message arrived: ");
                 // Take care of downcasting when messaging objects
                if(!(object instanceof Hashtable)){
                    System.err.println("Data must be sent in " +
                            "Hashtable format");
                    return false;
                }
                Hashtable<String,String> data =
                    (Hashtable<String,String>)object;
                Iterator<String> keys = data.keySet().iterator();
                while(keys.hasNext()){
                    String key = keys.next();
                    System.out.println(key + " - " + data.get(key));
                }
            }
            else return false;
        }
        return true;
    }
}
```

5.17. Sexta aplicación: Launch

```
import contributionjob.Contributor;
import drm.core.Node;

public class Launch implements Runnable{

    private final Node node;

    public Launch(Node node){
        this.node = node;
    }

    public void run(){
        node.launch(
            "DIRECT",
            new Contributor("contributionjob","root"),
            null
        );
    }
}
```

5.18. Sexta aplicación: contributionjob.Contributor

```
package contributionjob;
import java.util.*;

import drm.agents.*;
import drm.agentbase.*;
import drm.core.*;

/** The purpose of this very simple example is to show how
 * some agents can pass information through a collective. */
public class Contributor extends ContributorAgent {
    public static final long serialVersionUID = 78944524352L;

    public Contributor(String job, String name, Address root){
        super("Contributor", job, name, root);
    }

    public Contributor(String job, String name){
        super("Contributor", job, name, null);
    }

    /** This function is called by the collective to update
     * the contributions. So this is the point where you can
     * contribute to the collective :)*/
    public Object getContribution(){
        return "It's " + System.currentTimeMillis() +
                " here at " + name;
    }

    /** This one gets called each time a collective message
     * arrives or the refresh() method is called. Here we only
```

```
     * print the contributions, but much more complex things can
     * be done. You can access to the contributions at any point
     * you have access to the collective, not only here.*/
    public void collectiveUpdated(ContributionBox peer){
        Iterator<ContributionBox> contributions =
            collective.getContributions().iterator();
        while(contributions.hasNext()){
            ContributionBox cb = contributions.next();
            System.out.println("Contribution from "
                    + cb.contributor.getHost().toString() + "\n"
                    + cb.contribution + "\n");
        }
    }

    public void run(){
        /* Child nodes only become active with the getContribution()
         * and collectiveUpdated() methods. */
        if(root != null) return;

        // We send an agent to any available node
        Iterator<ContributionBox> contributions =
                getDRM().getContributions().iterator();
        while(contributions.hasNext()){
            Address target = contributions.next().contributor;
            Contributor child = new Contributor(job,"" +
                    Math.random(),new Address(null,-1,this.name));
            IRequest r = base.launch("DIRECT", child, target);
            while(r.getStatus() == IRequest.WAITING){
                try{Thread.sleep(1000);}
                catch(Exception e){
                    System.err.println("Exception: " + e);
```

```
                }
            }
            if(r.getStatus() == IRequest.DONE)
                System.out.println("Contributor sent to "
                        + target.getHost().toString());
            else
                System.err.println("Contributor couldn't arrive to "
                        + target.getHost().toString());
        }
    }
}
```

Capítulo 6

Conclusiones

Los objetivos cubiertos por este trabajo se resumirán a modo de conclusión en este capítulo. También se incluyen los pensamientos del autor sobre el posible futuro de esta biblioteca.

6.1. Sobre esta tesis

En el capítulo 2 se hizo un breve repaso de las tecnologías implicadas en este trabajo, estableciendo un camino conceptual importante desde los sistemas complejos, pasando por los sistemas de redes de pares y terminando en el proyecto DR-EA-M. Es fácil subestimar a los sistemas de redes de pares y considerar que su única utilidad es permitir la piratería de obras protegidas por derechos de autor. La teoría de sistemas complejos nos ayuda a desterrar este prejuicio y a reconocer el potencial de este paradigma computacional. Los avances conseguidos en caracterizar sistemas complejos deben ser aplicados a las redes de pares, que de este modo podrían ser el paradigma de computación distribuida dominante en un futuro no muy lejano.

Las aportaciones del capítulo 3 se encaminan en esta dirección. El autor ha definido detalladamente el algoritmo newscast para facilitar el trabajo de otros investigadores. Con una definición clara, el siguiente paso ha sido examinar con atención el comportamiento individual, local y global de los participantes en el algoritmo. El resultado ha sido obtener la topología de la red creada, con características nunca antes descritas. Se espera que estos resultados, las herramientas de análisis y los

razonamientos empleados sean de utilidad posterior.

En el capítulo 4 se expuso la implementación creada por Jelasity de un sistema multiagente basado en el algoritmo newscast que cumpliese los requerimientos del proyecto DR-EA-M. Se ha hecho un gran esfuerzo en definir las clases principales que lo componen, así como los procesos que se ejecutan en su interior. Se han ilustrado no pocos conceptos que quedaban oscuros o no se mencionaban en la documentación original, dando una visión mucho más completa de la biblioteca DRM.

El capítulo 5 es una continuación del capítulo 4 que comparte el mismo espíritu didáctico. Los ejemplos aportados originalmente no explicaban suficientemente el concepto del colectivo, resultando complicada su asimilación y uso. Los tres ejemplos añadidos por el autor se suman a los anteriores para formar un tutorial completo que guía al usuario en el aprendizaje de la biblioteca DRM.

6.2. Sobre DRM

Soy el segundo investigador que abandona esta herramienta después de invertir una cantidad significativa de tiempo en ella. No sé cuáles fueron las razones exactas de Mark para pasar a otro proyecto, pero es posible que coincidamos en el sentimiento de que es demasiado pronto para DRM. Esta biblioteca y su algoritmo subyacente están diseñados para redes de millones de elementos sobre los que apenas se ejerce control alguno.

En cierta medida es necesario ser un visionario para intuir, ya que no es posible probar, que una red tan gigantesca de agentes puede tener unos efectos emergentes que supongan avances enormes para la ciencia. No existen pruebas que garanticen que experimentos de ese tamaño no fueran una total perdida de tiempo, y esa gigantesca capacidad de computación se puede utilizar en problemas más inmediatos. Los componentes del proyecto DR-EA-M sí tuvieron esta visión, pero aún es necesario que más personas e instituciones la compartan.

Este documento garantiza de alguna manera que esta herramienta no caiga en el olvido y que otra persona pueda tomar el relevo, dando tal vez el último paso necesario para que se reconozca su utilidad.

6.3. Palabras finales

En este trabajo se ha dado la descripción más extensa hasta la fecha sobre la biblioteca DRM. Uno de mis deseos cuando empecé a trabajar sobre ella en el 2005 fue que existiera una documentación que me ayudara a comprenderla. Creo que he cumplido ese deseo, si no para que me ayude a mí, sí para que ayude al siguiente compañero en esta lucha por el progreso.

Valencia, Octubre de 2007

Bibliografía

[1] S. Androutsellis-Theotokis y D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys* **36-4**, 335–371 (Diciembre 2004).

[2] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. Merelo, B. Paechter, M. Preuss y M. Schoenauer. A framework for distributed evolutionary algorithms. En "Lecture Notes in Computer Science", tomo 2439, páginas 665–675 (2002).

[3] O. Babaoglu. Complex adaptive systems course, Bologna University. http://www.cs.unibo.it/~babaoglu/courses/cas/ (Octubre 2007).

[4] I. Clarke. Freenet software. http://www.freenetproject.org (Octubre 2007).

[5] B. Cohen. Bittorrent software. http://www.bittorrent.com (Octubre 2007).

[6] Microsoft Corporation. Msn messenger. http://es.msn.com (Octubre 2007).

[7] A. Cuesta-Cañada. DRM newscast testing. Informe t'ecnico, Instituto Tecnológico de Informática, Valencia, España (Octubre 2006).

[8] A. Cuesta-Cañada. A formal model for the newscast algorithm. Informe t'ecnico, Instituto Tecnológico de Informática, Valencia, España (Noviembre 2006).

[9] A. Cuesta-Cañada, E. Alfaro-Cid, K. Sharman y A. Esparcia-Alcázar. Análisis de la topología de una red newscast. En "Por publicar en las actas del CEDI07" (2007).

[10] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart y D. Terry. Epidemic algorithms for replicated database maintenance. En ACM, editor, "Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)", páginas 1–12, Vancouver (Agosto 1987).

[11] R. Diestel. "Graph theory". Springer Verlag, New York (2005).

[12] P. Druschel y A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. En "Proceedings of the 8th Workshop on Hot Topics in Operating Systems" (2001).

[13] P. Erdös y A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* **7**, 17–61 (1960).

[14] S. Fanning. Napster software. http://www.napster.com (Octubre 2007).

[15] XMPP Standards Foundation. Jabber software. http://www.jabberes.org (Octubre 2007).

[16] Future y emerging technologies european union programmme. Universal information ecosystems. http://cordis.europa.eu/ist/fet/uie.htm (Septiembre 2007).

[17] J. H. Holland. "Emergence, from chaos to order". Oxford University Press, UK (2000).

[18] M. Jelasity. Drm programming tutorial. Disponible en el paquete dr-ea-m.

[19] M. Jelasity, W Kowalczyk y M. van Steen. Newscast computing. Informe t'ecnico, Vrije Universiteit Amsterdam, Department of Computer Science (Noviembre 2003).

[20] M. Jelasity y M. van Steen. Large-scale newscast computing on the internet. Informe t'ecnico, Vrije Universiteit Amsterdam, Department of Computer Science (Octubre 2002).

[21] Q. Lv, P. Cao, E. Cohen, K. Li y S. Shenker. Search and replication in unstructured peer-to-peer networks. En "Proceedings of the 16th ACM International Conference on Supercomputing", Nueva York, EE.UU. (2002).

[22] N. Lynch y M. Tuttle. Hierarchical correctness proofs for distributed algorithms. En "Proceedings of the 6th Symposium on the Principles of Distributed Computing", páginas 137–151, Nueva York, EE.UU. (1987). ACM.

[23] M. Preuss M. Jelasity y B. Paechter. Maintaining connectivity in a scaleable and robust distributed environment. En "Proceedings of the IEEE International Symposium on Cluster Computing and the Grid", páginas 389–394, Berlin (Mayo 2002). IEEE Press.

[24] M. Preuss M. Jelasity y B. Paechter. A scalable and robust framework for distributed applications. En "Proceedings of the Congress on Evolutionary Computation", páginas 1540–1545. IEEE Press (2002).

[25] Mirabilis. Icq software. http://www.icq.com (Octubre 2007).

[26] G.Ñicolis y C. Rouvas-Nicolis. Complex systems. http://www.scholarpedia.org/article/Complex_Systems (Octubre 2007).

[27] Nullsoft. Gnutella software. http://www.gnutella.com (Octubre 2007).

[28] MIT Theory of Distributed Systems Group. Input-output automata. http://groups.csail.mit.edu/tds/i-o-automata.html (Septiembre 2007).

[29] A. M. Kermarrec P. T. Eugster, R. Guerraoui y L. Massoulie. From epidemics to distributed computing. *IEEE Computer* **37**, 60–67 (Mayo 2003). http://agva.informatik.uni-kl.de/pgc/papers/epidemic-eugster.pdf.

[30] B. Paechter, T. Back, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo y T. C. Fogarty. A distributed resource evolutionary algorithm machine (dream). En "Proceedings of the Congress on Evolutionary Computation (CEC 2000)", páginas 951–958 (2000).

[31] DR-EA-M Project. Dr-ea-m homepage. http://sourceforge.net/projects/dr-ea-m (Octubre 2007).

[32] S. Ratnasamy, P. Francis, M. Handley y R. Karp. A scalable content-addressable network. En "Proceedings of SIGCOMM 2001" (2001).

[33] M. Jelasity S. Voulgaris y M. van Steen. A robust and scalable peer-to-peer gossiping protocol. En "Proceedings of the AP2PC 2003", tomo 2872, páginas 47–58. Springer (2004).

[34] A. A. Sapozhenko. "Random graphs". Springer-Verlag (2002). http://eom.springer.de/G/g045000.htm.

[35] C. Shirky. What is p2p... and what isn't (Septiembre 2007).

[36] I. Stoica, R. Morris, D. Karger, M. Kaashoek y H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. En "Proceedings of SIGCOMM 2001" (2001).

[37] I. J. Taylor. "From P2P to web services and grids: peers in a client/server world". Springer-Verlag (2005).

[38] Berkeley University. The oceanstore project. http://oceanstore.cs.berkeley.edu (Octubre 2007).

[39] Berkeley University. Seti@home software. http://setiathome.berkeley.edu (Octubre 2007).

[40] M. Waldman. Publius software. http://cs.nyu.edu/w̃aldman/publius (Octubre 2007).

[41] X. Wang y G. Chen. Small-world, scale-free and beyond. *IEEE Circuits and Systems Magazine* **3-2**, 6–20 (2003).

[42] D .J. Watts y S. H. Strogatz. Collective dynamics of 'small world' networks. *Nature* **393**, 440–442 (Junio 1998).

[43] I. Witten, A. Moffat y T. Bell. "Managing gigabytes: compressing and indexing documents and images". Morgan-Kauffman, segunda edición edición (1999).

[44] N. Zennström y J. Friis. Kazaa software. http://www.kazaa.com (Octubre 2007).

[45] N. Zennström y J. Friis. Skype software. http://www.skype.com (Octubre 2007).

[46] B. Zhao, J. Kubiatowicz y A. Joseph. Tapestry: An infraestructure for fault-tolerant wide-area location and routing. Informe t'ecnico, University of California, Computer Science Division, Berkeley, 94720 (Abril 2001).