Universidad de Murcia

Departamento de Ingeniería de la Información y las Comunicaciones

Design and Implementation of a Wide-Area Middleware Infrastructure for the Development of Distributed Applications in Structured Peer-to-Peer Environments

PhD Thesis Dissertation

Presented by:
Carles Pairot Gavaldà

Directed by:
Dr. Pedro Antonio García López
Departament d'Enginyeria
Informàtica i Matemàtiques
Universitat Rovira i Virgili

Dr. Antonio F. Gómez Skarmeta
Departamento de Ingeniería de la
Información y las Comunicaciones
Universidad de Murcia

Murcia, November 2006

Dr. Antonio Fernando Gómez Skarmeta, Full Professor of Telematics Engineering in the Department of Communication and Information Engineering from the University of Murcia, AUTHORIZES:

The presentation of the thesis dissertation entitled *Design and Implementation of a Wide-Area Middleware Infrastructure for the Development of Distributed Applications in Structured Peer-to-Peer Environments*, written by Mr. Carles Pairot Gavaldà, under his supervision and that of Dr. Pedro Antonio García López, and submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy for the University of Murcia.

In Murcia, on ___ , ________ 2007
Dr. Antonio Fernando Gómez Skarmeta

Abstract

Distributed systems have evolved considerably in recent years. Depending on the scalability level required, several solutions can be found to develop distributed applications. If the number of users is relatively low, there are several centralized client-server models, with a rather simple subjacent architectural complexity, which provide an acceptable performance.

As soon as a distributed application needs to grow, its set of target users probably does so as well. It is at this point that centralized solutions start showing performance problems because of the well known *bottleneck* effect: that is to say, the central server is unable to efficiently cope with all of the users' needs. As a consequence of these scalability needs, the current trend is towards decentralization, thus minimizing this *bottleneck* effect by adding more servers that process logic. Therefore, not so many clients overwhelm the single central server.

These approaches are adequate for applications which do not need to be worldwide scalable because the number of potential users they are used by is relatively enclosed, and not excessively big. However, when worldwide scalability is required, decentralization is one of the best options, but it also creates many problems which have to be dealt with. Some of these problems include how to provide fault tolerance, how to deal with constant node joins and leaves, and many others.

This thesis proposes a set of middleware elements that help to develop distributed applications which are worldwide scalable, dynamic and fault tolerant. We have chosen a totally decentralized paradigm which guarantees the scalability of worldwide accessible applications. This middleware is built on three basic pillars. Firstly, we need an efficient *message routing layer*; secondly, we need a layer which provides *application-level multicast* services, so that one-to-many communication is efficient. Finally, a *decentralized persistence service* is required to store and restore data.

Using these three primitives, we build a wide-area middleware platform for the development of distributed applications, based on a **remote object layer** and a **distributed component layer**. The functionalities of these tiers' are as generic as possible so they do not depend on the underlying layers. Therefore, several underlying decentralized paradigms can be theoretically used without any variation in the middleware layer. Consequently, developers dispose of a higher level of abstraction when building wide-area distributed applications, since they can make good use of the services provided by the middleware layer.

In order to materialize this thesis, we implemented the proposed middleware framework on top of a relatively new paradigm of *peer-to-peer* networks, called *structured peer-to-peer networks*. These networks have interesting properties which make them highly suitable for the development of new distributed services.

The main contributions of our wide-area middleware proposal include the design and definition of a brand new set of invocation abstractions for our remote object model; a remote distributed interception algorithm; and a decentralized object location service. We have also defined two new load balancing algorithms based on the primitives of the remote object layer. As far as the reusable distributed component layer is concerned, our main contribution is the adoption of a lightweight container model which allows decentralized deployment and the activation of reusable components.

We also provide proof of the concept of our architecture with the implementation of SNAP: a decentralized, fault tolerant and scalable *Java Platform, Enterprise Edition (J2EE)* [46] web application deployment platform. The idea is to facilitate already existing J2EE applications in a worldwide interconnected network, in the easiest possible way, and by changing as little code as possible.

With the vision that we provide, we believe that developers will be able to produce newly decentralized applications by investing a minimal amount of learning time in a technology which surely still has much to show us.

Resumen

Los sistemas distribuidos han ido evolucionando ampliamente a lo largo de los últimos años. Dependiendo del nivel de escalabilidad requerido, nos encontramos con diferentes aproximaciones disponibles para el desarrollo de aplicaciones distribuidas. Para soportar un número de usuarios relativamente bajo, existen modelos centralizados cliente-servidor, que proporcionan un rendimiento aceptable, con una complejidad arquitectural subyacente bastante simple.

A medida que las necesidades de utilización de una aplicación distribuida crecen, probablemente su conjunto de usuarios destino también lo hace. Es en este punto en el cual las soluciones centralizadas empiezan a presentar problemas de rendimiento debido al conocido efecto de *cuello de botella*, donde el servidor central no es capaz de soportar de una forma eficiente las necesidades de los usuarios. Como consecuencia de las necesidades de escalabilidad, se tiende a descentralizar la lógica de proceso y de esta forma se intenta minimizar el efecto de *cuello de botella* añadiendo más servidores dedicados al procesamiento para que multitud de clientes no saturen a un único servidor central.

Estas aproximaciones son adecuadas para aplicaciones que no necesitan ser mundialmente escalables. Para tal efecto, el número de usuarios potenciales que las usan se mantiene relativamente limitado y no excesivamente grande. De todas formas, cuando se requiere conseguir una escalabilidad mundial, la descentralización se convierte en una de las mejores opciones, pero introduce muchos desafíos que deben ser tratados adecuadamente. Algunos de estos desafíos incluyen como proporcionar tolerancia a fallos, como tratar las constantes salidas y entradas de nodos, y muchos más.

El objeto de esta tesis es proponer un conjunto de elementos middleware para facilitar el desarrollo de aplicaciones distribuidas que sean escalables mundialmente, dinámicas y tolerantes a fallos. Para ello, se ha escogido como arquitectura subyacente un paradigma totalmente descentralizado que garantice la escalabilidad de las aplicaciones a nivel mundial. Dicho middleware se sustenta en tres pilares básicos necesarios para su correcto funcionamiento. Por una parte necesitamos disponer de una *capa de enrutamiento* de mensajes eficiente, por otra parte se requiere una capa que proporcione *servicios de multicast a nivel de aplicación*, para conseguir comunicaciones uno-a-muchos eficientes, y un *servicio de persistencia descentralizado* que permita almacenar y recuperar datos.

Utilizando estas tres primitivas, conseguimos construir una plataforma de middleware para aplicaciones distribuidas de área extensa basada en una **capa de objetos remotos** y otra de **componentes distribuidos reutilizables**. La idea es que las funcionalidades de estas dos capas sean lo más genéricas posibles de forma que no dependan de las capas inferiores. De esta forma, teóricamente se pueden utilizar diferentes paradigmas descentralizados por debajo sin que varíe para nada la capa de middleware. Por tanto, los desarrolladores disponen de un mayor nivel de abstracción a la hora de construir aplicaciones *wide-area*, ya que pueden utilizar los servicios que les proporciona la capa de middleware.

Para materializar la propuesta de esta tesis, hemos implementado la plataforma de middleware propuesta sobre un nuevo paradigma de redes *peer-to-peer* llamadas *redes peer-to-peer estructuradas*. Este tipo de redes ofrecen interesantes propiedades que las hacen muy adecuadas para el desarrollo de nuevas aplicaciones distribuidas.

Las principales aportaciones de la propuesta de middleware para aplicaciones de ámbito extenso de esta tesis incluyen el diseño y definición de un conjunto nuevo de abstracciones de invocación para nuestro modelo de objetos distribuido, un algoritmo de intercepción distribuida de objetos descentralizado, y un servicio de localización descentralizada de objetos. Además, se definen dos nuevos algoritmos de balanceo de carga basados en las primitivas ofrecidas por la capa de objetos remotos. Respecto a la capa de componentes distribuidos reutilizables, nuestra principal contribución es la adopción de un modelo de contenedor ligero que permite el despliegue y activación descentralizada de componentes.

Presentamos también una prueba de concepto de nuestra arquitectura propuesta con la implementación de SNAP: una plataforma de despliegue de aplicaciones web J2EE descentralizada, tolerante a fallos y escalable. La idea es que mediante SNAP los desarrolladores que provengan del mundo J2EE puedan desplegar sus ya existentes aplicaciones en una red mundial interconectada, de la forma más sencilla posible, y cambiando el mínimo código posible de sus aplicaciones.

Con la visión concreta que proporcionamos, creemos que el desarrollo de nuevas aplicaciones descentralizadas puede estar al alcance de la mano de la mayoría de los desarrolladores inviertiendo un tiempo de aprendizaje mínimo en una tecnología que en el futuro seguro que nos depara más sorpresas.

Agraïments

Suposo que gran part de la meva vida actual estigui condicionada a la informàtica depèn del fet que el meu pare em comprés, quan jo era molt petit, un ordinador Amstrad CPC6128, amb unitat de disc – que en aquells temps ja era un gran avanç. D'aquesta manera no vaig haver de sofrir les lentes càrregues des de la cinta de cassette, la qual cosa ja era un gran què. De bon principi em va entusiasmar tant aquest món que no podia parar d'aprendre'n cada cop més. Tant era així, que a casa em renyaven i em limitaven les hores que podia passar davant de la pantalla… De totes maneres no vaig escarmentar, i aquí estic, havent escrit la tesi doctoral i culminant la meva carrera en aquesta ciència que m'apassiona. Així doncs, el primer agraïment és per la meva família per haver-me donat tot el que estava a les seves mans per poder dedicar-me al que m'agrada i poder-m'hi guanyar la vida dignament. Pare, siguis on siguis, sé que estaries orgullós de mi.

Per descomptat, aquest esforç no hauria estat el mateix sense l'estimació i el suport constant de la Mireia, que ha vist com aquesta tesi ens robava massa hores, però no ha dubtat en donar-me ànims i acompanyar-me en molts dels viatges a congressos que he participat, encara que la temàtica no fos gaire del seu agrat.

Evidentment, també he d'agrair als membres del grup d'Arquitectura i Serveis Telemàtics de la Universitat Rovira i Virgili el seu suport, amistat, convivència… i perquè no, les farres, sopars i bon rotllo que hem viscut. Ha estat molt gratificant veure l'evolució del grup, des de la seva fundació (on vaig començar tot sol al Laboratori 135), fins a la seva consolidació actualment al Laboratori 144. Quin canvi… i per a millor! També voldria agrair especialment el suport del Rubén, treballador infatigable, el qual m'ha ajudat molt amb les seves aportacions, idees i treball. Una part d'aquesta tesi també és teva.

He d'agrair a l'Antonio el fet d'haver confiat en mi i també per les seves crítiques, sempre constructives, a l'hora de redactar els *papers*, així com aquesta tesi. També voldria agrair al Robert el seu constant suport i indicacions a l'hora de prendre certes decisions que ara que les veig en perspectiva, considero encertades.

Finalment, no podia deixar d'agrair al Pedro, director i amic, el seu suport incondicional al llarg de tots aquests anys. De fet, la seva empenta i optimisme van fer que decidís fer el doctorat. El camí ha estat llarg, i he passat moments d'estar a punt de llençar la tovallola… Pedro, moltes gràcies per haver confiat en mi i per ajudar-me en els moments en què les coses no acabaven de sortir. No sé si sense el teu constant suport aquesta tesi hauria estat posible.

Gràcies a tots sincerament.

Acknowledgements

I guess that a great deal of my life has been related to Computer Science because my father bought me an Amstrad CPC6128 computer with a floppy disk drive when I was a little boy. For this reason, I have never had to suffer incredibly long tape drive loading. From the very beginning I was so keen on this world that I could not stop learning more and more day after day. I was so enthusiastic that I was often told off by my parents, and the time I used to spend in front of the screen was regularly limited. However, I did not learn the lesson, and here I am: I have just written my PhD thesis and I am at the peak of my career in the science that I love. Therefore, the first acknowledgement goes to my family, for giving me everything that they could so that I could do what I liked, and I can now earn my living from such an enjoyable job. Dad, wherever you are, I know you would be proud of me.

Naturally, this effort would have not been the same without Mireia's love and encouragement. She has seen how this thesis has taken too many hours away from us. Yet she has never hesitated to cheer me up and come with me on many trips to the conferences I have participated in, although the topics were not much to her liking.

I would like to acknowledge the members of the research group Architecture and Telematic Services at the Rovira i Virgili Unversity for their support and friendship, and particularly the parties, the dinners and the good atmosphere. It has been very gratifying to see how the group has evolved from the very beginning (when I was all by myself in Laboratory 135), and has now consolidated in Laboratory 144. What a change, and for the better! I would also like to thank Ruben, an incredible, tireless worker, who has helped me a lot with his contributions, ideas and work. Part of this thesis is also yours.

My thanks also go to Antonio, for trusting me and always providing constructive criticism when I wrote papers and this thesis, and Robert for his constant support and guidance when I had to make decisions. In perspective, I seem to have made the right choice.

Finally, I cannot end without thanking Pedro – director and friend – for his unconditional support throughout these years. In fact, his optimism and encouragement made me decide to do my PhD. The path has been long, and at certain points I was tempted to give up. Pedro, thank you so much for trusting me and helping me at those moments when things were not going just right. I do not know if I would have made it without your constant support.

I thank you all sincerely.

Table of Contents

1 INTRODUCTION AND OBJECTIVES 25

1.1 THE WIDE-AREA MIDDLEWARE SCENARIO 25
1.2 REQUIREMENTS 26
1.3 OBJECTIVES 29
1.4 PROPOSED SOLUTION AND CONTRIBUTIONS 30
1.5 THESIS STRUCTURE 33

2 OVERVIEW AND BACKGROUND 35

2.1 WIDE-AREA APPLICATION REQUIREMENTS 36
2.2 PROPOSED ARCHITECTURE 39
2.3 BACKGROUND 41
2.3.1 Peer-to-Peer Wide-Area Routing Substrates 41
2.3.1.1 The First Generation: Central Index Location Scheme 44
2.3.1.2 The Second Generation: Unstructured Peer-to-Peer Networks 46
2.3.1.2.1 JxTA 49
2.3.1.3 The Third Generation: Structured Peer-to-Peer Networks 52
2.3.1.3.1 CAN 56
2.3.1.3.2 Chord 57
2.3.1.3.3 Pastry 59
2.3.1.3.4 Tapestry 62
2.3.1.3.5 Bamboo 63
2.3.1.3.6 Symphony 64
2.3.2 Wide-Area Application-Level Multicast 66
2.3.2.1 Reference Model 67
2.3.2.2 Event Systems Review 68
2.3.3 Wide-Area Persistence Systems Architectures 70
2.3.3.1 CFS 71
2.3.3.2 OceanStore 73
2.3.3.3 PAST 74
2.3.3.4 OpenDHT 75
2.3.3.5 Bunshin 77
2.4 RELATED WORK IN WIDE-AREA MIDDLEWARE SYSTEMS 79
2.4.1 Traditional Distributed Middleware Overview 79
2.4.2 Globe 80
2.4.3 Legion 81
2.4.4 JxtaJeri 82

2.4.5 The Peer-to-Peer Sockets Project *82*
2.4.6 The Common API for Structured Overlays *83*
2.4.7 Grid Computing *85*
2.4.8 Wide-Area Component-Based Architectures *88*
2.5 CONCLUSIONS 88

3 WIDE-AREA MIDDLEWARE PROPOSAL 92

3.1 INTRODUCTION 92
3.2 THE WIDE-AREA REMOTE OBJECT MIDDLEWARE LAYER 95
3.2.1 Innovative Services *96*
3.2.1.1 p2p Call Abstractions 96
3.2.1.2 Decentralized Object Location Service 100
3.2.1.3 Distributed Interception 103
3.2.1.4 Load Balancing Service 105
3.2.1.4.1 Decentralized load balancing management 105
3.2.1.4.2 Interceptor-based load balancing management 106
3.2.2 Additional Services *107*
3.2.3 Proof-of-Concept Implementation: Dermi *108*
3.2.3.1 Churn and Failure Handling 111
3.2.3.2 Validation 112
3.2.3.2.1 Experimental measurements 112
3.2.3.2.1.1 Direct Synchronous Calls 112
3.2.3.2.1.2 Synchronous Multicall 113
3.2.3.2.1.3 Anycall 113
3.2.3.2.1.4 Manycall 114
3.2.3.2.1.5 Decentralized load balancing scheme 114
3.2.3.2.2 Simulation Results 115
3.2.3.2.2.1 Distributed Interception 115
3.2.3.2.2.2 Interceptor-based Load Balancing Scheme 117
3.2.3.2.3 Discussion 118
3.3 THE WIDE-AREA DISTRIBUTED COMPONENT MIDDLEWARE LAYER 119
3.3.1 Architecture and Services *120*
3.3.1.1 Decentralized Component Location and Deployment 121
3.3.1.2 Decentralized Lightweight Container Model 122
3.3.1.2.1 Component Life Cycle Service 122
3.3.1.2.2 Component Persistence 124
3.3.1.2.3 Adaptive component activation 125
3.4 PROSPECTIVE USES OF OUR PROPOSED WIDE-AREA MIDDLEWARE 126
3.4.1 A Sample Distributed Computing Application Scenario *129*
3.5 SUMMARY 130

4 APPLICATIONS OF OUR MIDDLEWARE: SNAP 134

4.1 INTRODUCTION 134
4.2 RELATED WORK 135
4.3 ARCHITECTURE 137
4.3.1 Peer-to-Peer Routing Layer *137*
4.3.2 Dermi and p2pCM Layers *138*
4.3.3 Lightweight Web Server Layer and SNAP's Core Layer *138*
4.4 INNOVATIVE SERVICES 139
4.5 SNAP'S USE CASE SCENARIO 143
4.6 EMPIRICAL EVALUATION 146
4.7 PROSPECTIVE USES OF SNAP 149
4.8 SUMMARY 153

5 CONCLUSIONS AND FUTURE WORK 156

5.1 CONCLUSIONS 156

5.2 FUTURE WORK 159
5.2.1 The p2pWeb Model 159
5.2.2 Peer-to-Peer Middleware for providing Autonomic Computing 160

6 PUBLICATIONS 163

6.1 DERMI REMOTE OBJECT MIDDLEWARE 163
6.2 P2PCM COMPONENT-BASED MIDDLEWARE 164
6.3 PLANETDR AND SNAP 165

7 REFERENCES 168

ANNEX A. DERMI'S INSIGHTS 179

A.1 APPLICATION PROGRAMMING INTERFACE 179
A.1.1 Core Classes 179
A.1.1.1 RemoteEventListener 179
A.1.1.2 ERemote and EInterceptor 179
A.1.1.3 EventServer 180
A.1.1.4 DermiConnection 180
A.1.1.5 Session 183
A.1.1.6 DermiRemoteObject 183
A.1.1.7 DermiRemoteInterceptorObject 184
A.1.1.8 Registry 184
A.1.1.9 DermiProxy 185
A.1.1.10 RemoteException 186
A.1.2 Programming Dermi 186
A.1.2.1 Defining a Remote Object 186
A.1.2.2 Defining *anycall* methods 188
A.1.2.3 Defining *manycall* methods 190
A.1.2.4 Working with the Decentralized Object Registry 190
A.1.2.5 Implementing Distributed Interception Objects 191
A.1.2.6 Defining Replicated Objects 192

ANNEX B. P2PCM'S INSIGHTS 195

B.1 APPLICATION PROGRAMMING INTERFACE 195
B.1.1 Core Classes 195
B.1.1.1 ComponentInterface 195
B.1.1.2 Component 195
B.1.1.3 ComponentControl 196
B.1.1.4 ComponentFactory 198
B.1.1.5 ComponentUtil 198
B.1.1.6 Exceptions 198
B.1.2 Programming p2pCM 199
B.1.2.1 Defining a Remote Component 199
B.1.2.2 Component Deployment / Undeployment 202
B.1.2.3 Component Instantiation 202
B.1.2.4 Stateful Components 204

ANNEX C. SNAP'S INSIGHTS 205

C.1 APPLICATION PROGRAMMING INTERFACE 205
C.1.1 Core Classes 205
C.1.1.1 IApplication 207
C.1.1.2 Application 208
C.1.1.3 Context 208
C.1.1.4 SnapAppStartup 208
C.1.1.5 StartupServlet 209
C.1.1.6 SnapDataSource 209
C.1.1.7 DBInterceptor 209

C.1.1.8 SnapDeployer 209
C.1.1.9 Exceptions 209
C.1.2 The Database Engine: HSQLDB-WAN 210
C.2 PROGRAMMING SNAP 212
C.2.1 Web Application Adaptation 212
C.2.1.1 Static Web Applications 212
C.2.1.2 J2EE Web Applications 213
C.2.2 Web Application Deployment 215
C.2.3 Accessing SNAP applications 217
GLOSSARY 221

List of Figures

Figure 2.1. Proposed Generic Wide-Area Middleware Model Architecture 40
Figure 2.2. Peer-to-Peer network architecture.. 43
Figure 2.3. Napster's Architecture 45
Figure 2.4. Unstructured Topology of a Gnutella Network 48
Figure 2.5. Resource location via flooding on an Unstructured Peer-to-Peer Network . 49
Figure 2.6. Distributed Hash Table abstraction 53
Figure 2.7. CAN Lookup Example 57
Figure 2.8. A Chord ring consisting of many nodes 59
Figure 2.9. State of a hypothetical Pastry node 60
Figure 2.10. Pastry State and Lookup 61
Figure 2.11. Tapestry routing example 62
Figure 2.12. Neighbours in Pastry and Bamboo 63
Figure 2.13. CFS replication algorithm 72
Figure 2.14. OpenDHT architecture 76
Figure 2.15. Common API Diagram 83

Figure 3.1. Our proposed Wide-Area Middleware Architecture................................. 93
Figure 3.2. Our Remote Object Middleware’s p2p call abstractions............................ 96
Figure 3.3. Multicall abstraction.. 98
Figure 3.4. Anycall example 99
Figure 3.5. Definition of anycall methods in a remote object's interface 100
Figure 3.6. Example of object handle insertion and location...................................... 102
Figure 3.7. Binding and looking up an object from the decentralized object location service 102
Figure 3.8. Distributed interception algorithm 104
Figure 3.9. Binding an interceptor to an already running object................................ 104
Figure 3.10. Example of a decentralized and interceptor-based load balancing scheme 107
Figure 3.11. Dermi architecture diagram and how it fits within the Common API for Structured Overlays.. 110
Figure 3.12. Distributed Interception Simulations 116
Figure 3.13. Interceptor-based load balancing simulation 117
Figure 3.14. Diagram of the Wide-Area Component Model’s architecture................. 124
Figure 3.15. Adaptive component activation scheme 126

Figure 3.16. Component diagram of the p2pCM's sample distributed computing application.. 130

Figure 4.1. SNAP's architecture diagram 138
Figure 4.2. SNAP Deployer splash screen 139
Figure 4.3. SNAP Deployer Wizard 140
Figure 4.4. SNAP's Decentralized Application Locator.. 141
Figure 4.5. Sample deployment descriptor file for any SNAP Application (*snap-war.xml*)... 144
Figure 4.6. SNAP Web application access times 147

Figure A.1. Dermi's Class Diagram 182
Figure A.2. Interface of a Dermi Remote Object 187
Figure A.3. Implementation of a Dermi Remote Object.. 188
Figure A.4. Definition of anycall methods in a remote object's interface 189
Figure A.5. Sample anycall Dermi application class diagram 189
Figure A.6. Definition of manycall methods in a remote object's interface................ 190
Figure A.7. Binding an object into the decentralized object location service............. 191
Figure A.8. Looking up an object from the decentralized object location service....... 191
Figure A.9. Implementing a distributed interceptor object 192
Figure A.10. Binding an interceptor to an already running object 193
Figure A.11. Object replication example 194

Figure B.1. p2pCM Class Diagram 197
Figure B.2. Implementation of a component's factory class....................................... 200
Figure B.3. A p2pCM component interface 201
Figure B.4. A p2pCM component implementation 202
Figure B.5. Deploying a p2pCM component 202
Figure B.6. A p2pCM component instantiation 203
Figure B.7. Sample p2pCM Application Class Diagram... 204

Figure C.1. SNAP's API. Interface *IApplication* 206
Figure C.2. SNAP's Introspection API usage 207
Figure C.3. SNAP's Class Diagram... 207
Figure C.4. HSQLDB-WAN Class Diagram 211
Figure C.5. SNAP Deployer option window.. 217
Figure C.6. SNAP's Home Page... 218

List of Tables

Table 2.1. Services present in JxTA 51
Table 2.2. Comparison of different p2p resource-location architectures 55
Table 2.3. Comparison of various structured overlay protocols 56
Table 2.4. Pastry's exposed API 61
Table 2.5. Comparison of event systems 69
Table 2.6. Comparison of the wide-area persistence systems analyzed 70
Table 2.7. CFS software layering 71
Table 2.8. PAST exposed API 75
Table 2.9. Common API's Tier 1 API 84
Table 2.10. Common API's Tier 0 API 85
Table 2.11. Comparison of the different Wide-Area Middleware approaches 89

Table 3.1. Performance of one-to-one direct synchronous calls 112
Table 3.2. Performance of one-to-many multicalls 114
Table 3.3. Performance of anycalls 114
Table 3.4. Performance of the decentralized load balancing scheme 115

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rick Cook, The Wizardry Compiled

If the automobile had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside.

Robert X. Cringely, InfoWorld magazine

Tended a ser un poco aprendices de todo, para vuestro bien, y maestros en algo, para bien de los demás.

Pere Puig Adam, Matemático y maestro de matemáticos

Chapter One

1 Introduction and Objectives

1.1 The Wide-Area Middleware Scenario

Over the years, the Internet has been growing steadily in its number of users and nowadays its ubiquitous nature is well-assumed by everybody. Network bandwidth has increased considerably and the emergence of many successful wide-area applications has made it more and more popular. Apart from network bandwidth itself, computers have greater overall capacity and their resource sharing capabilities are improving day by day.

When creating global-scale Internet-based distributed applications, developers repeatedly face the same implementation issues: object location, replication, mobility, caching, etc. Middleware plays a key role in addressing these challenges because it provides a common higher-level interface for application programmers and hides the complexity of myriad underlying networks and platforms. Middleware systems have a long tradition in centralized client–server environments, but there are very few globally scalable middleware solutions.

The distributed object-oriented middleware frameworks that get the most attention are those that model messaging as method calls. These systems are often called Remote Procedure Call (RPC) [113] systems. The major benefit of these systems is that they make remote procedure (or method) calls appear to be local procedure calls (LPCs). This is a powerful abstraction that considerably simplifies the development of remote applications. Mature examples of this middleware are the Common Object Request Broker Architecture (CORBA) [36], Java Remote Method Invocation (RMI) [47] or the Distributed Component Object Model (DCOM) [32], deprecated by Microsoft’s .NET Framework [54].

Message Oriented Middleware (MOM) has recently received considerable attention because of its decoupled nature, which nicely solves asynchronous one-to-many interactions and highly dynamic distributed environments. Unlike RPCs, MOMs do not model messages as method calls; instead, they model them as events in an event delivery system. Clients send (produce) and receive (consume) events, or *messages*, and producers and consumers do not explicitly know each other. All applications communicate directly with each other using the MOM. Messages generated by applications are meaningful only to other clients, because the MOM itself is only a message router.

Nevertheless, distributed object-oriented frameworks and MOMs are still almost isolated worlds that do not fully benefit from each other's unique advantages and concepts.

Architecturally, both middleware approaches are mostly built on top of centralized client/server models, and this is proven to work well in local-area or even metropolitan-area environments. However in wide-area settings, these platforms clearly suffer from scalability problems, although these can be solved by forming cluster topologies among servers. This option may not be economically viable in all cases. The current trend is to head towards decentralized models which benefit more from the computing at the edge paradigm, where resources available from any computer in the network can be used and are normally made available to their members.

Therefore, there is a need for a middleware platform that can be used to develop worldwide oriented distributed applications. This middleware must be scalable, provide fault tolerance, be able to adapt to continuous node joins and leaves in the network, provide high availablity guarantees, and be a good use maker for the computational resources available on the edges of the Internet. Do any of the available middleware systems comply with all these requirements? The answer is mainly no.

Existing middleware approaches for wide-area scale applications do not provide all of these services. Therefore, a great deal of time must be invested in providing such guarantees first. Sometimes, it is very difficult or practically impossible to comply with these requirements. Therefore, we propose that a middleware framework be created for developing distributed applications that run on top of very dynamic and changing environments. This approach fulfills our requirements, thus allowing developers to concentrate on the application itself, and not on the underlying common services.

1.2 Requirements

The creation of a wide-area middleware platform is a complex challenge, and requires a set of basic features, which means that a variety of problems must be overcome. We need to fulfil the following requirements:

- **Scalability**. Any wide-area middleware needs to be scalable, so that it can support applications which may require global-scale concurrent access and utilization. There are a wide variety of client-server middleware approaches. However, the platforms suffer from scalability problems, since the server itself becomes the bottleneck of the whole architecture. One solution to this came in the form of the *clustering* or *federation* of servers. Following a decentralization pattern, servers are made redundant so when one becomes unavailable, another one can take its place. There are several variants of this system, where requests can be redirected to one server or another, depending on their load, or simply in a sequential order. Nevertheless, this redundant server alternative is normally expensive to achieve and maintain.

Taking this decentralization trend to its logical conclusion involves a pure decentralized architecture. In this system, all members behave in the same way as *peers*. Therefore, each of these *peers* can perform the same functions as another *peer*. As a consequence, no single entity is more powerful than any other. This scheme provides the best scalability, but communication mechanisms must be efficient enough to make performance acceptable.

- **Fault tolerance**. We require our wide-area distributed applications to be as fault tolerant as possible. This feature allows graceful recovery from errors, and it is aimed to be provided by the middleware layer in a transparent way. Therefore, we need to assure that all possible resources can be located at all times. Our network routing layer should be aware of possible failures, and should be smart enough to avoid these failures by re-routing messages appropriately.

- **High availability**. This requirement is a consequence of fault tolerance. High availability refers to the fact that access to any resource must be guaranteed at all times. Therefore, we need to have a transparent mechanism that can guarantee access to resources at any time. The idea of high availability usually comes in the form of data replication. So if there is some redundancy, the guarantee of high availability improves.

  Therefore, we should provide some redundancy or replication of resources so as to permit high availability. In order for this redundancy to be kept synchronized, we need efficient state change notifications. Since propagating state from one object to each of its replicas is going to be a usual task, we require a group communication service. This service should use the ***multicast*** primitive, which allows notifications to be sent from one source to many targets.

  There are several ways of providing this service. The first is *IP Multicast* [51]. IP Multicast is a method by which a message can be sent simultaneously to several computers, not just one. In order to do this, the message is sent to a range of addresses reserved for multicast groups (224.x.x.x-239.x.x.x). Each computer must also decide whether or not it wishes to be part of a specific group.

  The main problem of IP Multicasting is that it is only supported by a few routers in the Internet. Therefore, this alternative is too impractical to be used. In order to provide the same functionality as IP Multicasting, *application-level multicast* solutions allow the same multicasting features at the application level. Events and messages can be relayed from origin to destination by an application specific component, called the *event bus*.

  Centralized event systems have proven to be a very useful group communication middleware in the design of distributed applications. The distributed information bus (*event bus*) is responsible for transmitting to subscribers events thrown by producers based on the information contained in these events.

  Unfortunately, the same scalability limitations apply to group communication middleware, so, when scaling up, client-server solutions are not suitable because the event server may become a bottleneck when dealing with millions of simultaneous notifications. Therefore, we sketch the need for such a service, not only for replica state propagation, but for any global scale group communication our middleware offers to higher level layers.

- **Load balancing**. Load balancing's main aim is to distribute processing and communication activity evenly across a computer network so that no single device is overwhelmed. This service is especially important for networks in which it is difficult to predict the number of requests that will be issued to a node, and this is the case of our scenario. Sometimes, a specific resource may be requested by many nodes, swamping the responsible node (*the Britney problem*). In order to avoid such a problem, load must be evenly distributed to other nodes holding replicas of the resource (linked to the high availability requirement).

  Even though an efficient group communication service is needed for our middleware if *one-to-many* notifications are to be propagated, this scenario can be further enriched, and related to the the requirement we are describing. We could think about benefiting from other primitives (if available), like ***anycast*** [53]. Anycast allows messages to be delivered to the *closest* member of a multicast group, where closeness is defined in terms of a particular metric (for example, network proximity).

  This primitive can be used to construct a load balancing service that our middleware can offer transparently to the upper layers. For example, requests can be directed to a group member and, if it is overwhelmed, re-routed to the *closest* member, by following an anycast pattern.

- **Dynamicity**. Our middleware must guarantee dynamicity. This feature accounts for the fact that the members of the network do not usually remain constant. The routing substrate must allow for this behaviour. Therefore, it must guarantee that resources are moved from nodes when they leave, and that new responsibilities are assigned to newcoming nodes. Since our middleware targets highly dynamic environments, the routing substrate must provide the necessary primitives to guarantee this feature without losing data.

- **Make good use of the computational resources on the edges of the Internet**. This requirement is closely related to scalability. We want our middleware to make the most of each of the nodes connected to the network. In traditional client-server architectures, the majority of resources for an application's execution are managed and hosted on the main server. Therefore, any persistent data, state change, or complex calculation is performed on the server. Clients interact only with the server to provide it with the necessary data, but little business logic is performed on them. If we imagine the huge number of clients throughout the Internet, it is easy to see that most of their resources are largely not utilized. These resources, the resources of the *edges of the Internet* could be made available to others in order to contribute to a larger network of peers.

  This idea is not new, since several already available applications have exploited this paradigm (for example, SETI@Home [41], United Devices Cancer Research Project [23], Folding@Home [16], or even the ubiquitous eMule [15]). However, we consider this requirement a must for a wide-area middleware, since we cannot rely on resources scattered on a single remote server. Resources must be contributed and managed by the whole community.

- **Usability and Programming Abstractions**. Middleware solutions should never forget their target users, and therefore never underestimate the importance of their own ease-of-use. It is not sufficient for a middleware to just implement all the functionality required for building applications; it should also be easy to use. Implementing applications should be as easy as possible, and both the middleware and the applications developed should be easy to deploy and manage. It is also important to distinguish which programming abstractions are available to the developer in order to work with the middleware. Categorization may include availability of remote objects or components, object location facilities, group communication primitives, etc.

Analyzing all these requirements, we propose three important core components on which our middleware proposal will be based: a **wide-area efficient communication substrate**, which allows inter-node communication, a **wide-area efficient application-level multicast service**, providing the necessary tools to propagate notifications from one source node to many, and a **wide-area persistence layer**, which permits persitent information to be stored and retrieved on top of a decentralized infrastructure. These three main pillars are crucial for fulfilling the above mentioned set of requirements.

1.3 Objectives

All the above information shows that it is not easy to develop wide-area applications on top of a wide-area routing network, since no middleware infrastructures are available. This gives developers the problem of re-implementing common services over and over again, thus wasting precious programming time which could be dedicated to other matters.

In this study, we aim to propose a developer framework suitable and flexible enough to allow wide-area application development and deployment on top of a worldwide scalable peer-to-peer network infrastructure.

Therefore, the objectives of this thesis are to facilitate the creation and deployment of wide-area scope distributed applications. To achieve this, we require a middleware approach which abstracts all common services needed by developers, so that implementing a distributed application on top of a peer-to-peer substrate is as easy as possible. Indeed, achieving this ambitious goal is a complex matter, so we have adopted a bottom-up strategy which starts from the lowest level and scales up. In this way, complexity is increasingly hidden from the developer. In this study, we plan to achieve the following goals:

- Define a layered architecture which makes it easy to modularize the development of wide-area applications. This architecture must simplify the development and deployment of applications and their inner components, and uniformize access to common services.

- Define a complete component and application development model, abstracting the underlying layer complexities, and even allowing these components / applications to be deployed in the proposed framework. This model must contain the complete development cycle of an application, and enable generic templates to be used, which simplify the development cycle.

- Implement the proposed generic model by means of two core layers that contain *remote objects* and a *reusable wide-area component framework*. Thus, we demonstrate the viability of our theoretical model.

- The proof of the concept of our proposed model comes in the form of a wide-area application deployment infrastructure, which makes use of all the services provided by the remote object and component layers. We demonstrate the genericity of our proposed framework and its applicability in heterogeneous environments.

1.4 Proposed Solution and Contributions

On the basis of the objectives listed above, we plan an approach to the problem which has the next phases:

- Definition of a generic architecture model made up of different layers which provides a set of generic common services. This model must be generic enough to be applied to a variety of software designs.

- Analysis of the routing substrates available which can provide worldwide application scalability for the model defined above. We also analyze the state of the art in the set of wide-area common services targeted to facilitate the design of global distributed applications.

- Design and construction of the proposed generic model by means of two complementary middleware approaches: *remote objects* and *distributed reusable components*. The *remote object layer* provides the foundations and most important innovative services for the component layer. This *component layer* allows the lightweight components to be defined and deployed. These components can later be reused to provide a higher level of abstraction to compose wide-area distributed applications.

  - Our **first contribution** is the definition of a new set of remote object invocation abstractions. We have defined a lower-level core of remote object functionalities (whose practical implementation is called **Dermi**). This provides the traditional object-to-object (one-to-one) remote method invocations. It also provides object-to-objects (one-to-many) calls by using a wide area application-level multicast communications bus. If this underlying information bus also provides us with network proximity-aware primitives such as *anycast*, we can also provide the *anycall* and *manycall* abstractions. Such remote method invocation techniques allow a method to be invoked on one of the *nearest* objects which complies with a parameterized condition.

Moreover, *hopped* calls allow for fault tolerance when invoking methods on dead objects: if another live replica of the object exists, it responds to the call. It is important to recall that such invocation abstractions aim to be generic in the sense that they are not closely coupled to a specific underlying information bus or routing substrate. Therefore, the underlying layers can be switched for others with the same functionalities, though the same interfaces must be respected.

- Our **second contribution** is the definition of a decentralized object location service. This service allows remote objects / components / applications to be located and inserted into our decentralized generic model. It is similar to any *naming service*, but provides a fault tolerant and scalable level of indirection. Any object data can be stored and located later by using simple *bind()* and *lookup()* primitive operations. One of the major advantages of this service is that it is inherent to the *Key-Based Routing substrate (KBR)* we use. Nevertheless, its definition is generic enough to allow other routing substrate algorithms to be used, and only requires that they follow the same *Application Programming Interface (API)* contract.

  This service is closely related to the decentralized persistent layer of our model. Any persistent data to be stored is recorded in a decentralized way, as are object or component handles. In order to support fault tolerance, data is replicated among a specified number of nodes. A set of algorithms is used to take care of bottlenecks and node overwhelming.

- Our **third contribution** is the distributed interception service. By means of the underlying information bus, we provide primitives that easily intercept remote object calls, like in *Aspect Oriented Programming (AOP)* techniques. Therefore, invocations to remote objects can be captured, analyzed, transformed, and even discarded. This service provides runtime interception with no need to change either the source or the target object's code. Type-compatible interceptors are therefore added or removed in runtime by calling our model's interception service. This approach, for example, is used for monitoring and providing load balancing to our objects or components.

- Our **fourth contribution** is the provision of wide-area load balancing through interceptors or the *anycall* abstraction. Two ways of providing load balancing in wide-area distributed objects and components are described. Both these alternatives fit gracefully into our proposed generic middleware framework for developing global distributed applications. Basically, these two load balancing techniques target different domain areas. For those scenarios in which each object is aware of its own load, the *anycall*-based scheme enables the target object to be selected by letting each target node decide. This approach is rather stateless and provides proximity aware support. The alternative scheme uses an interceptor to determine the state of each of the objects to load balance. Requests are directed to the interceptor which forwards the invocations to the less loaded object server (thus defining what *load* policy is to be taken into account).

  Both schemes are complementary and target different use cases, providing the load balancing requirement with enough genericity and flexibility.

  - Our **fifth contribution** is the adoption of a decentralized lightweight container model for our *component framework*. The transition from remote objects to distributed components is such that it provides a higher level of abstraction to application developers. Therefore, we define a component-based reusable layer, whose implementation is called **p2pCM,** which presents an alternative way of *holding* any component's life cycle routines. Distributed components are modelled as remote objects, including a life cycle service, and a decentralized deployment and location service. Instead of having a monolithic heavyweight container housing all the components, we opt to make each of the nodes in the worldwide network a lightweight container. Therefore, components are distributed throughout the network, and benefit from the underlying services provided by the remote object layer.

- Finally, we present a proof of concept implementation which directly benefits from the underlying framework services. This software application validates our whole generic model and shows that it is appropriate for designing wide-area scalable applications. We also introduce another prototype to demonstrate the viability and genericity of our model.

  - Our **final contribution** is the proposal of this wide-area application deployment service. As a proof of concept for our generic wide-area middleware model, we have developed an application which allows web applications to be securely deployed on top of a worldwide network. This platform (called **SNAP**) provides fault tolerance, persistence, interoperability via web services, clustering, and other services to application developers. The idea is that this platform enables new wide-area distributed applications to be developed and deployed in a trauma-free way. Applications can be developed with either Dermi or p2pCM primitives, or even following the traditional client-server guidelines. Once the application is ready to be deployed, it is automatically prepared to run on top of a worldwide network. For instance, fault tolerance is automatically managed by activating application replicas throughout the network. Therefore, should one application node become unavailable, the application does not.

1.5 Thesis Structure

The structure of this thesis is summarized as follows:

Chapter 2 presents the big picture of our proposed generic model, and tries to establish a reference model in which we expose the problem we are trying to solve. Therefore, we present the requirements that a wide-area middleware should fulfill, and we observe that the solution requires research into three main building blocks: available wide-area routing substrates, wide-area application-level multicast infrastructures for message dissemination, and wide-area persistence systems. First we explain and analyze the available wide-area routing substrates and their properties, and then we go on to describe decentralized solutions based on peer-to-peer models. Secondly, we describe event-based architectures suitable for worldwide scalable applications and, finally, we describe the available wide-area persistence systems which can be used to provide a persistence layer and common services for our middleware. After describing this background, we analyze previous related work in the field of wide-area middleware solutions, and observe that the requirements discussed at the beginning of the chapter canot be fulfilled. As a consequence, there is a need for a new wide-area middleware proposal, which is described in the next chapter.

Chapter 3 is central to this thesis and describes our proposed model's most important contributions: the architecture and innovative services of our wide-area remote object middleware, as well as our reusable component layer architecture, based on the previous object middleware. We outline the features and services these layers provide for application developers and demonstrate the viability of our proposal by presenting our practical implementation of the model in the form of **Dermi** and **p2pCM**. We also present an empirical evaluation of these implementations. Finally, we summarize the initial requirements and demonstrate that our middleware fulfills them all.

Chapter 4 introduces a proof-of-concept implementation which uses our generic model's features. This application is called SNAP and is a wide-area application deployment infrastructure. We introduce related work regarding this application, and describe how it benefits from Dermi and p2pCM's services. We finish the chapter by describing some other prospective uses for our wide-area middleware proposal.

Chapter 5 presents the conclusions derived from this work and a variety of possible future research lines.

Chapter Two

2 Overview and Background

Because of the complexity of the problem to be solved, our work has been influenced by highly heterogeneous research areas. As we shall explain below, we have been heavily influenced by wide-area routing techniques, application-level multicast services, decentralized persistence services, and software engineering theories applicable to distributed systems.

As we mentioned in the previous chapter, our objective is to design a distributed architecture oriented to the development of wide-area distributed applications. Moreover, our intention is that this architecture be quite generic, use reusable code, and provide easy access to commonly needed underlying services. These objectives are so ambitious that several essential research fields must be studied and analysed if the whole problem is to be understood.

This chapter is subdivided into four main blocks. First, we use current techniques to analyze the requirements of today's wide-area applications, and existent problems. Next, we give an overview of our generic model's proposed architecture. This approach is useful to show our architecture's *big picture*, and it will be the join point for all of the forthcoming blocks in this chapter and the whole thesis. This *big picture*, as well as the requirements section, serves as an introduction to the three blocks in which our middleware proposal lies. All these tiers act like pieces of a puzzle which fit together and provide a set of common services to the upper-level layers. Therefore, this block is essential if the proposed generic model is to be described, and the forthcoming sections and chapters of this research work are to be fully understood.

The third block shows that our architecture proposal needs a scalable and efficient routing layer, which acts as the core communication infrastructure for our proposal; an upper event-based application-level multicast layer that allows efficient one-to-many communication between different upper-level elements for our model; and finally, a wide-area persistence service which makes it possible to store and look up data efficiently by using the underlying routing layer.

After analyzing this background, the fourth block relates it closely to other work in the wide-area middleware field. Finally we show that no solutions comply with all our requirements.

2.1 Wide-Area Application Requirements

Wide-area applications are normally associated by the general public with those applications which are accessible throughout the Internet. Most people consider that any web page is a wide-area application. This is untrue. Even though the world wide web (WWW) itself is an application that can be found over the world, it mainly follows a centralized client-server architecture. This means that, in most cases, only one server is backing up the whole web page or application. This, in turn, means that whenever the server is down, access to the web page or application is impossible.

We are tired of navigating the web and finding the usual "*Server is not responding*" error message. We obviously try again after a few minutes and, if we are lucky, we can continue navigating. However, if we were in the middle of a transaction, say filling in a form, we unfortunately observe that our data has very probably vanished. Generally speaking, we observe that one of the most widely used applications of the Internet is not fault tolerant. Non-scalability is also one of WWW's problems, since servers may stop serving requests if they are overwhelmed with requests themselves or have exhausted their computing capacity.

It is true that the WWW is accessible to the whole world but, in a sense, if the whole world tries to use it at the same time, it is no longer accessible. Wide-area applications should be accessible efficiently at any time, anywhere, by a massive number of concurrent users.

It is expected that in a near future, ubiquitous network connections and interactions between devices, systems, services, people, and organizations will be the rule rather than the exception [59]. The realization of the full potential of all conceivable patterns of interaction and collaboration will require a sophisticated global infrastructure, on top of which service providers can develop their applications. However, today's infrastructure is rudimentary, which makes the development of new services both difficult and expensive. A new global infrastructure is needed to handle the sheer complexity of new and varied models of interaction and collaboration.

As we have previously stated in Chapter 1, decentralized approaches are efficient at providing such guarantees. As a matter of fact, the appearance of truly wide-area applications has fostered the popularity of peer-to-peer networks, which have therefore proven their stability and correctness as a wide-area substrate for global concurrent access to resources [79].

Broadly speaking, the p2p applications community has focused on three different application domains: computing, collaboration and file sharing. Computing, also known as *cycle-stealing* or *PC Distributed Computing* uses otherwise-idle cycles on desktop and laptop computers for large-scale computation. Condor [13], Entropia [67], United Devices [23] and Data Synapse [14] are all examples of such p2p computing applications. These systems have shown that for some problem domains, the volatility, security and data distribution issues can be resolved so that PC Grids can compete in performance with traditional cluster technology.

Peer-to-peer collaboration applications allow peers to construct and discover ad-hoc groups, join and leave groups, and perform shared operations as a member of a group. Groups may be long-lived, made up of friends, colleagues, and organizations or everybody. Groups may also be very short-lived and spontaneous when they are groups of people in the same place at the same time. Sun's JxTA [39], for example, builds group support into the infrastructure, enabling applications to leverage this group support. Groove Networks [24] is another example: they provide integrated solutions to enable users on desktops or laptops to share and jointly work on documents.

The third class of applications [110] is perhaps a special case of the previous domain: file sharing applications such as Napster [34], Gnutella [85], KaZaA [42], eMule [15], BitTorrent [102], among others, enable a large number of users to share content. In some cases, the content files are small (for example, music files or images), and the challenge is to quickly locate any instance of the content among the copies that are available at that instant; in other cases, the files are much larger and p2p technology is used to stream the content to the user from nearby resources.

By analyzing these three use case scenarios in depth, we can determine the requirements that are common to these kinds of applications in particular, and to wide-area applications in general:

- **They make good use of the resources available on the edges of the Internet**. To achieve their goals, applications should use resources that are available throughout the network. These resources can include CPU processing power, storage space, network bandwidth, etc. It is highly likely that a little power from millions of machines together will be stronger than a few powerful servers.

- **They promote collaboration among groups**. People tend to merge into groups of interest. Applications targeted to wide-area utilization must support such requirements, since collaboration is widespread in human-to-human and human-to-computer interaction.

- **They provide resource sharing capabilities**. This requirement is closely related to using the resources on the edges of the Internet. Applications share the resources of the whole community so that *my* resources can be used by other peers.

- **They provide fault tolerance, and make numerous resources available**. It is not enough for others to be able to share *my* resources. If I leave the network, these resources should still be reachable. Applications need this requirement if they are to continue working properly when members leave the network, or when there are resource spikes, meaning that some kind of load balancing strategy needs to be provided.

Even though these requirements may be enough for wide-area applications, we believe that other points that are often not considered need to be improved. These can be thought of as new requirements for the successful deployment of these kinds of applications:

- **New wide-area applications should be easy to develop**. At the moment no higher-level middleware solutions are available that make it easy. Developers must implement a wide-area application practically with their bare hands. Some lower-level middlewares exist, but their provided abstractions are limited to socket, or message programming, which makes it tremendously complicated to implement even the most trivial service. It is true that Groove Networks [24] provides a framework for developing p2p collaborative applications, even though it is based on a proprietary network infrastructure. However, Groove is not suitable for building generic wide-area applications, since it is more focused on enterprise collaboration matters, and cannot be considered as a global wide-area middleware solution.

- **Location awareness should be fostered**. Because collaboration is a requirement, we consider that wide-area applications should make it easy to collaborate with *close* members. This requirement involves the notion of *location awareness*, which is normally mapped to a proximity metric of network distance. For example, we may want to find people interested in *Bruce Springsteen* who are near us. Another example, involving file sharing applications: we would probably like to retrieve large files from peers who are *close-by* in terms of network distance to maximize efficiency.

- **Connectivity should be maximized**. The peers in p2p systems are typically desktop computers running in a complex network environment. In home networks, Internet service providers may block certain types of network traffic; network address translators (NATs) and firewalls may hide resources behind a common name; and even the network address of a peer may change quite frequently. Corporate networks are also becoming quite complex and the issues described for home networks are also challenges in companies. Wide-area applications should try to overcome these problems in order to reach the maximum number of peers without problems, thus maximizing resource sharing and utilization.

- **Security should be improved**. Security in p2p systems normally focuses on anonymity and user privacy. In order to achieve additional security, p2p systems have developed such alternative mechanisms as community-based trust (user ratings), and replication and verification. However, trying to provide strong user identities, and trusted user proofs is a challenge, which can be desirable in some wide-area applications. We mention this requirement, but it is left out of the scope of this thesis.

By discussing all the requirements that a wide-area application should fulfill, we conclude that they are similar to the requirements of a wide-area middleware (see chapter above). Therefore, we introduce the need for three important core components on which our middleware proposal will be based: a **wide-area efficient communication substrate**, which allows inter-node communication, a **wide-area efficient application-level multicast service**, which provides the necessary tools to propagate notifications from one source node to many, and a **wide-area persistence**

layer, which permits the storage and retrieval of persitent information on top of a decentralized infrastructure.

2.2 Proposed Architecture

As we have pointed out above, the aim of this research work is to develop a distributed architecture oriented to the development of wide-area distributed applications. We want to foster wide-area applications with a middleware proposal that eases this development process. Our intention is that this architecture be quite generic, use reusable code, and provide easy access to commonly needed underlying services.

This approach cannot be based totally on a centralized solution, since these systems can be overwhelmed and produce important bottlenecks when accessed simultaneously by millions of clients. Naturally, such client-server approaches can be made more scalable if supported by expensive load-balancing clustering systems. However, our aim is to provide an infrastructure that is generic enough to achieve the same goal at a much lower cost. Therefore, *decentralizing* and distributing the tasks that a heavyweight centralized server would perform among *n* nodes has proven to be a low-cost efficient solution [12, 15, 43].

Bearing in mind the requirements described in the section above and that the target platform needs to be decentralized for the reasons described above, we identified three minimal core requirements for achieving such an ambitious goal:

- The **routing substrate,** which serves as our whole system's communication core layer, needs to be scalable, efficient and fault tolerant. This layer is responsible for routing messages between network nodes in an efficient and fault tolerant way. It is important that this routing substrate be as autonomous as possible so that it can transparently handle node failures, arrivals, departures and other exceptional events in the upper layers. Therefore, the routing substrate should have the desirable features of self-organization, self-healing, flexibility, etc.

- An **application-level multicast service** is required to efficiently propagate messages to many nodes at the same time. This multicast service must be scalable and efficiently deliver messages to many clients, thus allowing one-to-many communication. This approach is intuitively more efficient than sending *n* notifications throughout the network. Moreover, as specified in the requirements, this service can provide *anycast* primitives which enable messages to be sent to *any* members. If this *anycast* service is network proximity aware, we can even get messages delivered to the sender's *closest* members in terms of network latency, for example. This service is crucial for our wide-area middleware proposal, and we will describe it in greater detail in the next chapter.

- Any wide-area application will need to store its data somewhere. A **persistence service layer** is obviously needed. If all application users (which could be millions) tried to store or look up data from a centralized persistence service, the bottleneck problem would arise again. Therefore, a decentralized persistence layer is needed if applications are to be able to store data in a highly available service, which allows for fault tolerance, efficiency and scalability concerns.

These three building blocks are the basic elements that a wide-area generic model should provide the upper levels. Nevertheless, it is impractical to build new applications on top of this bare core. We need to go one step further in terms of abstraction. This is why we have built *remote object* (Dermi) and *component-based* (p2pCM) additional tiers which allow easier access to these common services than the application programmer does. Figure 2.1 shows the complete picture of our overall proposal.

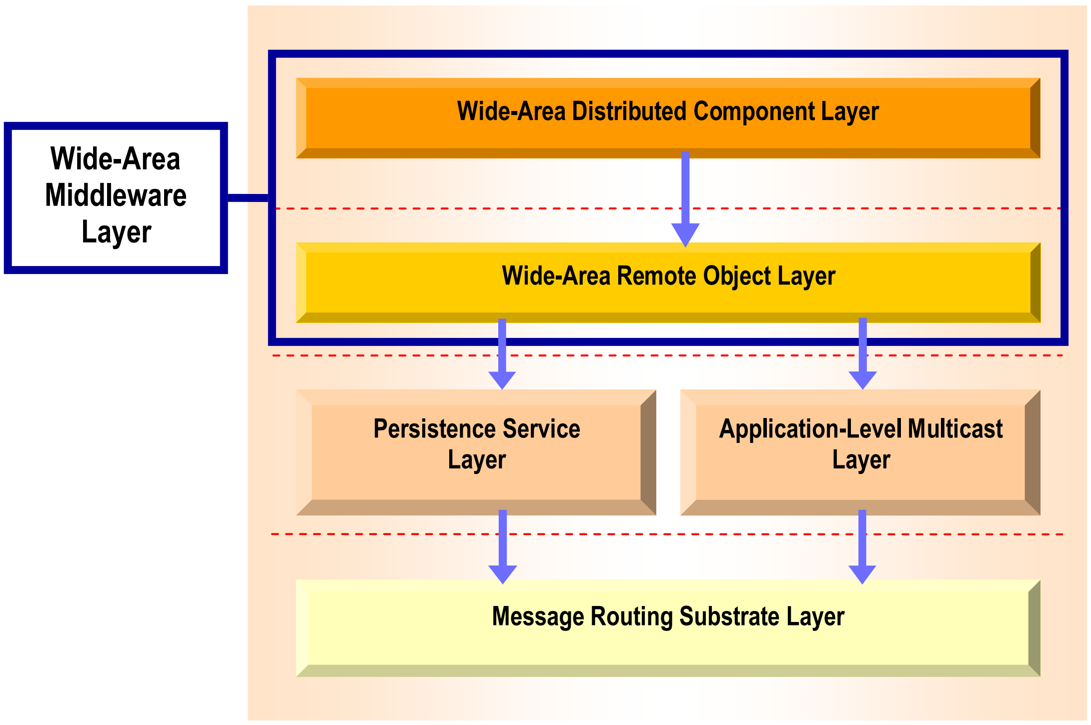

Wide-Area Middleware Layer
Wide-Area Distributed Component Layer
Wide-Area Remote Object Layer
Persistence Service Layer
Application-Level Multicast Layer
Message Routing Substrate Layer

Figure 2.1. Proposed Generic Wide-Area Middleware Model Architecture

Having described our proposed model’s core architecture, we will now go on to describe the background to the three main layers of which it consists. We shall start by analyzing the wide-area routing substrate alternatives that are available, move on to describe suitable application-level multicast services and finally focus on globally scalable, efficient persistence solutions.

2.3 Background

2.3.1 Peer-to-Peer Wide-Area Routing Substrates

Any communication substrate which is intended to be used for wide-area routing needs to be able to route messages between network nodes in an efficient and fault tolerant way. It is important that this routing substrate be as autonomous as possible so that it can transparently handle node failures, arrivals, departures and other exceptional events in the upper layers. Therefore, the routing substrate should have the desirable features of self-organization, self-healing, flexibility, etc.

Nowadays, Internet applications tend to be organized in a relatively small number of powerful servers which service many client nodes. In fact, this is the standard way in which the World Wide Web (WWW) operates: a client-server architecture. Nevertheless, although HyperText Transfer Protocol (HTTP) is such a lightweight protocol, this model suffers from scalability problems. Once the application's hosting server is overwhelmed with requests from many clients, it clearly becomes a bottleneck. Moreover, if the server crashes, the application becomes unusable (unless redundant clusters solve the problem). Therefore, the client-server architecture does not seem to be suitable for low-cost wide-area fault tolerant massive application accessibility.

Many successful wide-area applications that support high numbers of concurrent connected users have advocated the use of peer-to-peer (p2p) technologies to solve the scalability problem. These applications use the decentralization paradigm to avoid bottlenecks. Consequently, not only is there a single server that holds all the application data, but also a bunch of nodes which work together to support the application. If a service node goes down, another one can take its place and continue serving requests. The same happens when trying to load balance requests: if there is more than one server to serve these requests, the load can be balanced over all the available servers. This approach taken to the extreme is the peer-to-peer philosophy, which has no clients or servers: all nodes are treated as equal *peers*.

Another paradigm which follows this same decentralization line is **Grid computing** [62]. The popularity of both Grid and p2p has led to a number of (often contradictory) definitions. We assume that Grids are sharing environments implemented via the deployment of a persistent, standards-based service infrastructure that supports the creation of, and resource sharing within, distributed communities. Resources can be computers, storage space, sensors, software applications and data, all connected through the Internet and a middleware software layer that provides basic services for security, monitoring, resource management, and so forth. Resources owned by various administrative organizations are shared under locally defined policies that specify what is shared, who is allowed to share, and under what conditions.

Peer-to-peer (p2p) is defined as a class of applications that takes advantage of the resources —storage, cycles, content, human presence—at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, p2p design requirements commonly include independence from *Domain Name Systems (DNS)* and significant or total autonomy from central servers. Their implementations frequently involve the creation of overlay networks [107, 111, 116] with a structure that is independent of that of the underlying Internet. We prefer this definition to the alternative *decentralized, self-organizing distributed systems, in which all or most communication is symmetric* [97] because it encompasses large-scale deployed (albeit centralized) “p2p” systems (such as Napster [34] and SETI@home [41]) where much experience has been gained.

Consequently, the general idea seems to split both the Grid and the p2p worlds into those cases in which infrastructure is used to allow seamless access to supercomputers and their datasets (Grids), and those which enable ad hoc communities of low-end clients to advertise and access the files on communal computers (p2p).

To add more confusion, the concept of p2p Grids has also been devised [62]. A p2p Grid contains a set of services that includes those of Grids and p2p networks and naturally supports environments that have features of both limiting cases. In p2p Grid architectures, Web services play a very important role. There is also an event service which links these Web services and other resources together. In p2p Grids, everything is a resource, and they are exposed directly to users and to other services.

After this introduction, we go on to analyze the evolution and state of the art in p2p network routing substrates. We chose this kind of substrate because Grids do not gracefully support highly dynamic environments where nodes join and leave frequently, as one of our requirements states. Moreover, Grids do not typically focus on using the resources of the edges of the Internet,: rather they focus on large-scale computing applications.

As defined in [94], the term “peer-to-peer” refers to *a class of systems and applications that employ distributed resources to perform a critical function in a decentralized manner. The resources encompass computing power, data (storage and content), network bandwidth, and presence (computers, human, and other resources). The critical function can be distributed computing, data/content sharing, communication and collaboration, or platform services. Decentralization may apply to algorithms, data, and meta-data, or to all of them. This does not preclude retaining centralization in some parts of the systems and applications if it meets their requirements*.

Peer-to-peer architecture embodies one of the key technical concepts of the Internet, described in the first Internet Request for Comments, *RFC 1, Host Software* [49] dated 7 April 1969. More recently, the concept has achieved recognition from the general public because of the absence of central indexing servers in architectures used for exchanging multimedia files. In this context, p2p computing has deployed a vast number of applications mainly oriented to people communication and collaboration, as well as distributed computing. One of their main features is high availability, thanks to the multiple *peers* that make up a group of interest. This characteristic aims to guarantee that almost any of the group members can satisfy any user’s request.

This philosophy remains in stark contrast to that of traditional computing models, where high availability is the result of complex load balancing mechanisms. Examples of popular peer-to-peer applications include **Napster** [34], **SETI@home** [41], **KaZaA** [42], **eMule** [15], or the most recent **BitTorrent** [12], or **BOINC** [11], to name a few. However, the high availability of these networks is not the panacea, since it can be easily compromised: there is no guarantee that peers will behave correctly, or that they will contribute resources to the community. The so-called **tragedy of the commons** phenomenon often occurs, where only a few peers contribute to the network, and others (often called *leechers*) try to take everything out without putting anything in. This particular issue is explained in [97] - *... nobody has to think of being nice to the next guy or put in even a tiny bit of extra effort. We've heard plenty about the tragedy of the commons. In the 1968 essay that popularized the concept, "The Tragedy of the Commons," Garrett Hardin wrote: "Therein is the tragedy. Each man is locked into a system that compels him to increase his herd without limit - in a world that is limited. Ruin is the destination toward which all men rush, each pursuing his own best interest in a society that believes in the freedom of the commons. Freedom in a commons brings ruin to all."*.

Peer-to-peer architectures tend to be non-reliable. This non-reliability comes from the fact that there can be constant joins and leaves, and that resources have to be relocated on the fly. The wide diversity of node capacities, operating systems and system architectures which conform the network give p2p this heterogeneity factor. In order to support these particular features, p2p networks must be self organizing and self repairing, as well as fault tolerant. Their typical objective is for all nodes to make good use of the shared distributed resources (i.e. CPU time, bandwidth, storage capacity, etc.).

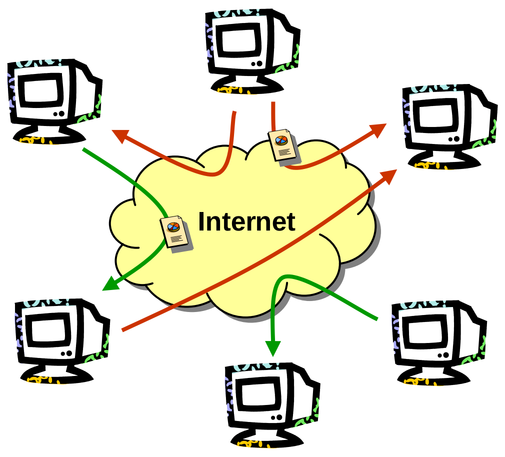


Figure 2.2. Peer-to-Peer network architecture

Peer-to-peer networks can be classified in a wide variety of ways. However, we are interested in classifying the various p2p architectures to date by the algorithms they use when trying to locate resources. One of the principal challenges of such systems is how to locate a particular resource. Since this can be a highly complex problem, several approaches have been taken to overcome it. In chronological order, these are the **central index** location scheme, the **unstructured location** scheme (mainly based on unstructured p2p networks) and the **distributed hash table** scheme (based on structured p2p networks).

2.3.1.1 The First Generation: Central Index Location Scheme

The first attempt to locate all resources in a p2p network was made by **Napster** [34]:, the **central index location scheme**. Napster is an online music service which was originally a file sharing service created by Shawn Fanning. Napster was the first widely-used p2p music sharing service, and it had a major impact on how people, especially university students, used the Internet. Its technology allowed music fans to easily share *MPEG-1 Layer 3 (MP3)* format song files with each other, thus leading to the music industry's accusations of massive copyright violations. Although the original service was shut down by court order, it paved the way for decentralized p2p file-sharing programs such as Kazaa [42], Limewire [31], and Bearshare [10], which have been much harder to control. Napster continues to live on with pay services today. However, the popularity of the first Napster has made it a legendary icon in the computer and entertainment fields

Napster's architecture consisted of a central index server where all users logged in and uploaded metadata about which resources they were sharing. Content searches were made on the index server, and resource transfers were made between peers themselves.

More specifically, the central directory server maintained a metadata index of all the files shared in the network. These metadata include file names, creation dates, file sizes, and copyright information. The server also maintained a table of user connection information including the user's IP address and line speed. First, a file query was sent to the server. A query consisted of a list of desired words. When the server received a query, it searched for matches in its index. The query results, including a list of users who held the file, were sent back to the user who initiated the query. The user then opened a direct connection with the peer who had the requested file for downloading. This process can be seen in Figure 2.3.

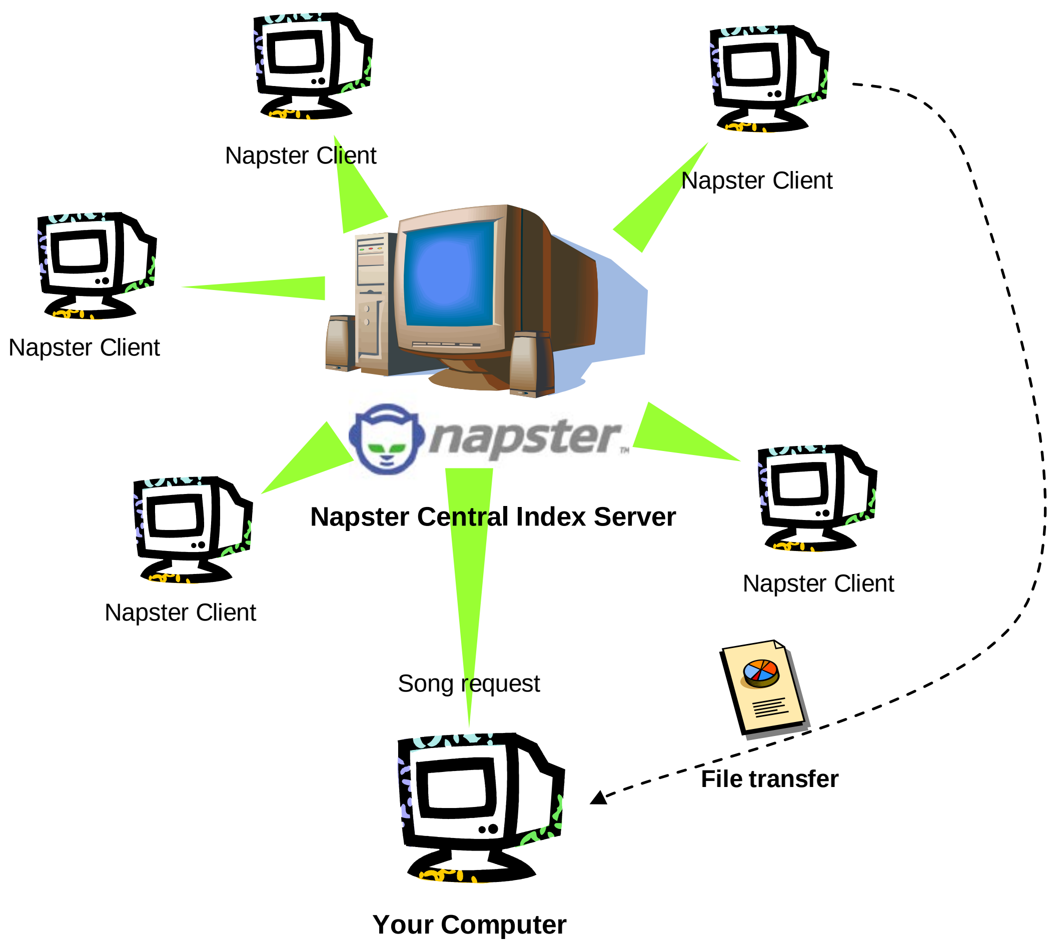


Figure 2.3. Napster's Architecture

This early Napster model had some problems, not all of which were related to the central index location scheme:

- **Peer inaccessibility**: not all peers were accessible because of such factors as *firewalls*, clients coming and going (the so-called *churn*), excessive round trip times, slow upload speeds due to the asymmetric nature of ADSL connections, etc.

- **Focalized attacks**: denial of service issues were relatively easy to perform, since it they involved attacking core index servers. Moreover, clients could lie about their shared content (*eg*: *serve Frank Sinatra in response to download Eminem*). Another form of attack was hacking the Napster client programs, in order to run the protocol in various disruptive ways. The discographic industry also tried to figure out who was serving files so that they could be sued.

These problems made Napster evolve to a more robust solution, thus providing enhanced directory servers which probed clients, tracking their health. It automatically reported download problems to trim bad sources from the list. Incentives were also provided: data sources were ranked so that clients who had been up for a long time, seemed to have fast connections and appeared to be *close* to the client doing the download (using the notion of *locality* in terms of Internet distance) had preference.

Finally, they implemented a parallel downloading mechanism in an attempt to leverage asymmetric download/upload speeds.

Nevertheless, one of fundamental problems of this centralized approach was clearly scalability, since the central index server became a bottleneck for the whole network. It quickly became overwhelmed with a vast number of user queries (logins, logouts, searches, etc.). Moreover, this approach was not considered to be a *pure* peer-to-peer solution, since content indexing was centralized. This centralization issue also made it easier for organizations like the *Recording Industry Association of America (RIAA)* to close down some of these networks (Napster [34], AudioGalaxy [9], etc.) which illegally exchanged copyrighted audio and video. Since the whole system relied upon these indexes, it was only a matter of time before they were located and closed down, thus leading these applications to a predictable demise.

2.3.1.2 The Second Generation: Unstructured Peer-to-Peer Networks

In order to palliate all the above mentioned problems, the next solution to the problem of efficiently locating resources in a peer-to-peer network adopted a pure decentralized unstructured p2p network architecture. In these systems, peers have the same capability and responsibility. Communication between peers is symmetric: there is no central directory index server where the files' metadata is stored. This metadata is stored locally among all peers. Examples of such systems include **Gnutella** [21], **Freenet** [18]**,** or **MojoNation** [33]. A wide variety of techniques for resource location were spawn in this period in time. Some of the most popular are:

- *Flooding:* The query is sent to the node's neighbours, and spread from neighbour to neighbour through a maximum number of hops. This technique is explained in greater detail below.

- *Epidemic algorithms* [83]: Epidemic algorithms follow the paradigm of nature by applying simple rules to spread information by just having a local view of the environment. These algorithms are easy to implement and guarantee message propagation in heterogeneous environments that are not always coherent.

  Each epidemic algorithm contains a population consisting of a set of interactive, communicating units. These units use a ruleset that defines how to spread specific information that might be of interest to other units.

  This ruleset is considerably affected by the design of the algorithm and can be freely chosen. The only requirement is that at a specific time *t* a unit must have one of the following states regarding specific information:

  - *Susceptible*: the unit does not know anything about the specific information but it can get it.
  - *Infective*: the unit knows the specific information and uses the ruleset to spread it.
  - *Removed (Recovered)*: the unit knows the specific information but does not spread it.

- *Random Walk* [91]: Random walk is a well-known technique which forwards a query message to a randomly chosen neighbor at each step until the object is found. This message is called a *walker*. When standard random walk is used (with one walker), it reduces message overhead significantly, by an order of magnitude compared to *flooding* across the network topologies.

  However, this efficiency increases user-perceived delay in successful searches by an order of magnitude. To decrease the delay, the number of *walkers* is increased. That is, instead of just sending out one query message, a requesting node sends *k* query messages, and each query message takes its own random walk. The expectation is that *k* walkers after *T* steps should reach roughly the same number of nodes as one walker after *kT* steps.

Perhaps one of the best known of these unstructured systems is the **Gnutella** file sharing protocol [21]. In Gnutella, clients (downloaders) are servers as well (called *servents*), and they may join or leave the network at any time, making Gnutella highly fault tolerant. But there is a cost: information is slowly discovered. Searches are done within the virtual network while actual downloads are done offline (through HTTP). The core of the protocol consists of 5 **descriptors** *(PING, PONG, QUERY, QUERYHIT and PUSH).*

The strategy adopted in Gnutella was to utilize the **flooding** algorithm. It consisted of a network of unstructured, anarchically connected nodes, which did not depend upon a centralized index server when joining, leaving or searching content. Throughout all of this anarchy, the search mechanism worked quite well. The idea was that if a node wished to start a search, it issued a search message to all of its connected neighbours looking for a resource. When a neighbor received the message it looked to see if it could satisfy the query (i.e. whether the resource was found locally). If it could, then it routed a message back to the sender saying that it had been found there. If it could not, the message was routed on to the neighbour's connected nodes and so on. This process was repeated until the message had traveled a maximum number of hops (*time-to-live (TTL)* parameter), and it was then considered to have expired. This mechanism prevented the network from being overwhelmed with a bunch of messages, and avoided a general broadcast for each request, which would result in rapid degradation of the network's performance.

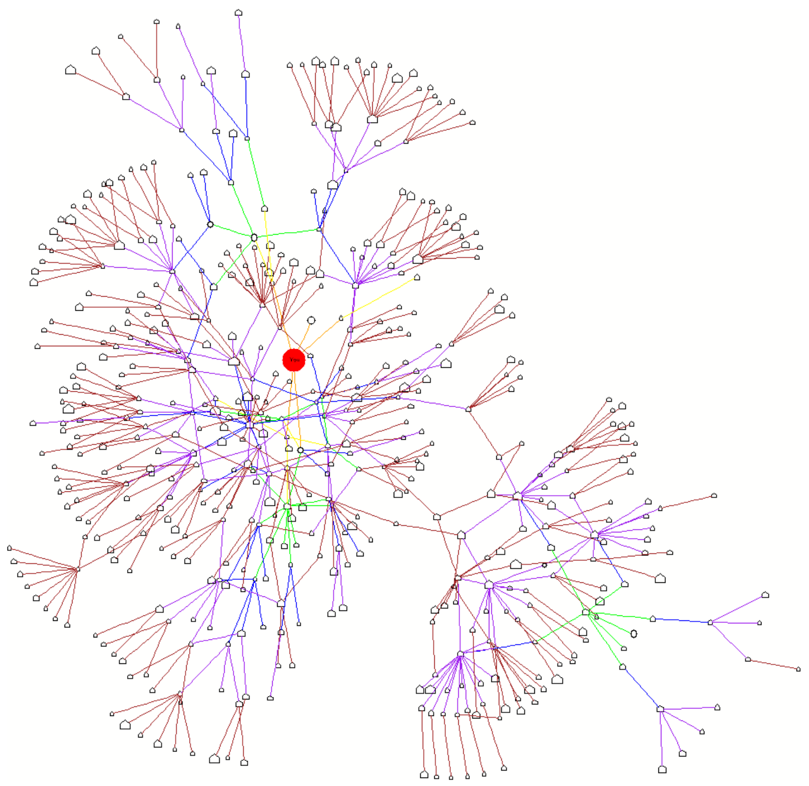


Figure 2.4. Unstructured Topology of a Gnutella Network
A snapshot from a local Gnutella peer network in any neighbourhood using the mapping functions of the Gnucleus [20] client.

Gnutella's *Breadth First Search* (BFS) algorithm always finds the optimal path, and the performance is the same under random and target failure. However, the search bandwidth used by queries proportionally increases with the number of network nodes. Therefore, the Gnutella communication overhead is relatively high [56], resulting in approximately 63% of messages being of type *PING/PONG*, and 37% *QUERY/QUERYHIT*. Moreover, the problem of **free riding** is definitely an important issue, meaning that 70% of Gnutella users share no files, and nearly 50% of all responses were returned by the top 1% of sharing hosts.

Despite these issues, the Gnutella protocol works quite well in terms of returned results and network scalability. However, it is not deterministic in the way resources are found. The problem is that if a resource is too far away from the requester, it is not found in the default number of search message hops. Therefore, it is reported that the resource does not exist, whereas it does. As a consequence, with the flooding mechanism, resource location is highly probable but not guaranteed, since if ***r*** of ***N*** nodes have a copy of the resource, the expected search cost is at least ***N/r***, i.e. ***O(N)***: if the number of copies (***r***) increases, the probability of finding the resource also improves. Keeping many copies of any resource is needed to keep overhead small.

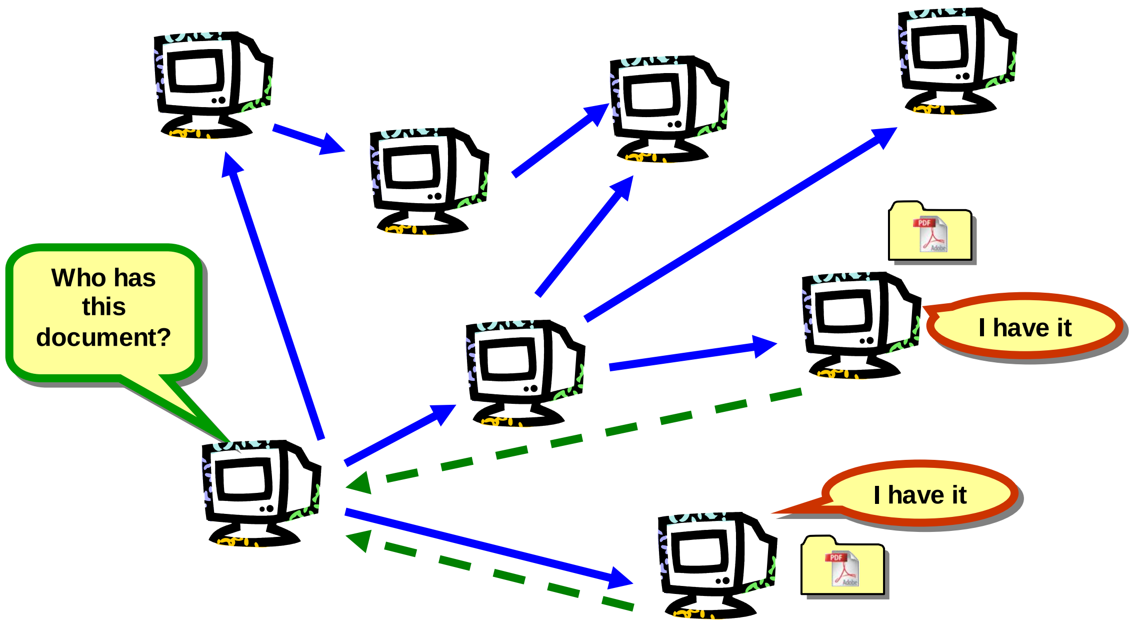


Figure 2.5. Resource location via flooding on an Unstructured Peer-to-Peer Network
A node asks its neighbours for a document, and they keep propagating the request to their respective neighbours (up to a maximum number of hops). Once the document has been found, the hosting nodes answer directly to the request initiator.

Within the second generation of peer-to-peer location architectures we can also find some hybrid approaches, which use some nodes as super-peers which act as central indexes for their children nodes. These solutions are called *partial centralized indexing systems*, and they use a central server which registers users to the system and facilitates the peer discovery process. After a peer authenticates on the server, the server provides it with the IP address and port of one or more super-peers (or *supernodes*) to which the peer then connects. Local super-peers index the files shared by their connected peers and proxy search requests on behalf of these peers. Therefore, these nodes act as query redirectors to other super-peers and help locate content among them. Searches within children nodes are made using flooding or random walking techniques. Examples of such systems include **KaZaA** [42], **Morpheus** [45] or the **eDonkey network** [15], and even **Skype** [43].

In the case of the Morpheus [45] file sharing system, peers are automatically elected to become *supernodes* if they have sufficient bandwidth and processing power (a configuration parameter allows users to opt out of running their peer in this mode). Once a Morpheus peer receives its list of *supernodes* from the central server, little communication with the server is required.

2.3.1.2.1 JxTA

Since we are analyzing state of the art in wide-area routing substrates, and since we are describing unstructured p2p networks, we must talk about **JxTA**, which is precisely an unstructured p2p routing substrate.

JxTA [39] is an open source peer-to-peer platform created by Sun Microsystems in 2001. It is defined as a set of *eXtensible Markup Language (XML)* based protocols that allow any device connected to a network to exchange messages and collaborate whatever the network topology. JxTA is one of the most mature p2p frameworks currently available and was designed to allow a wide range of devices - PCs, mainframes, cell phones, PDAs - to communicate in a decentralized manner.

JxTA defines two main categories of peers: *edge peers* and *super-peers*. Super-peers can be further divided into *rendezvous peers* and *relay peers*. Each peer has a well defined role in the JxTA peer-to-peer model. *Edge peers* are usually defined as peers that have transient, low bandwidth network connectivity. They usually reside on the edges of the Internet, hidden behind corporate firewalls or accessing the network through non-dedicated connections. A *rendezvous peer* is a special purpose peer that is in charge of coordinating the peers in the JxTA network and provides the necessary scope to message propagation. A *relay peer* allows the peers that are behind firewalls or NAT systems to take part in the JxTA network. This is done by using a protocol that can traverse firewalls (for example, HTTP).

The components that make up a JxTA system are exactly the same as those that can be identified in many p2p network implementations, and include *peers and peer groups, services, pipes, messages, and advertisements*.

The logical partitioning of the physical network creates working sets of peers called *peer groups*. Peer group memberships can overlap with no restriction; in other words, any peer can belong to as many peer groups as necessary. The JxTA specification does not dictate or recommend an appropriate way of forming peer groups. In a JxTA network, a peer group is a collection of peers that share resources and services. Consistent with JxTA's design philosophy, a peer group is specified to be as unconstrained and as generic as possible.The existence of these peer groups mandates some means of maintaining membership. Again, the JxTA specification states only the minimal need for maintaining group membership, without dictating how this should be done. This membership service is a part of the core JxTA services, but it can take many forms – it can be either a database or directory service, for instance, and based on either a centralized or a distributed implementation.

JxTA services are available for shared use by peers within a peer group. In fact, a peer may join a group primarily to use the services available within that group. A set of services, called core services, is essential to the basic operation of a JxTA network. We have already seen one instance of a core service: the membership service.

Table 2.1 shows the core services included in version 1.0 of the JxTA specification.

Service name	Description
Pipe	The main means of communications between peers; provides an abstraction for a one-way, asynchronous conduit for information transfer.
Membership	Determines which peers belong to a peer group; handles arrival and departure of peers within a peer group.
Access	Security service for controlling access to services and resources within a peer group; a sort of security manager for the peer group.
Discovery	A way peers can discover each other, the existence of other peer groups, pipes, services, and the like.
Resolver	Allows peers to refer to each other, peer groups, pipes, or services indirectly through a reference (called an *advertisement*); the resolver binds the reference to an implementation at run time.

Table 2.1. Services present in JxTA

One way to transfer data, files, information, code, or multimedia content between peers is through logical **pipes**, as defined by the JxTA specification. JxTA pipes are used to send messages (with arbitrary content) between peers.

A pipe instance is, logically speaking, a resource within a peer group. It is typically implemented through the pipe service. Unlike conventional (UNIX-like) systems, JxTA pipes are unidirectional and asynchronous. Two peers requiring two-way communications will have to create two independent pipe instances.

The blocks of information carried though pipes are referred to as JxTA **messages**. JxTA messages are data bundles that are passed from one peer to another through pipes. The JxTA specification is again as generic as possible here, so as not to inadvertently introduce any implementation-dependent policies into the definition of a message. A message is defined as an arbitrarily sized bundle, consisting of an envelope and a body. To provide for a standard, easy-to-parse, universal encoding mechanism, JxTA messages are currently XML documents.

Advertisements are the less obvious cousins of messages. JxTA advertisements are also XML documents. The content of an advertisement describes the properties of a JxTA component instance, such as a peer, a peer group, a pipe, or a service. For example, a peer with access to an advertisement of another peer can try to connect directly to that other peer. A peer with access to an advertisement of a peer group can use the advertisement to join that group. The current Internet analogue to an advertisement is the domain name and DNS record of a Web site. The JxTA specification does not dictate how advertisements are created, circulated, or destroyed.

Because of the nondeterministic nature of the JxTA world, a specific resource request may not return for minutes, hours, or even days; in fact, it may never return at all. In addition, people from different parts of the world requesting the same resource are likely to get different copies of the resource from completely different servers. In an attempt to palliate the first problem, a recent implementation of a Distributed Hash Table on top of JxTA is currently being researched: GISP [19].

2.3.1.3 The Third Generation: Structured Peer-to-Peer Networks

The next generation of p2p networks tries to solve the non-determinism problem of resource location. The idea is that if a specific resource is on the network, it should be found. To find it, the philosophy changes, and these networks start becoming structured node groupings. Nodes are arranged in a structured fashion, typically following tree or ring formations. The objective is to assign particular nodes to store particular content. When a node wishes to look for a resource, it must be redirected to the node which is supposed to hold it.

The challenges of these structured peer-to-peer networks are:

- to avoid bottlenecks in particular nodes, thus distributing responsibilities *evenly* among the existing peers.

- to adapt to nodes joining or leaving (or failing). As a consequence, it is logical to give new responsibilities to joining nodes, and redistribute responsibilities from leaving nodes.

These challenges perfectly match the idea of a **hash table**, in which each data item is associated with a key. The key is hashed to find its corresponding *bucket* in the hash table. Each bucket is expected to hold *#items/#buckets* items. In order to map this data structure to our problem, it is considered that nodes are the buckets in our global **Distributed Hash Table** (DHT). Therefore, the key is hashed to find the resource's responsible peer node, obtaining data and load balancing across nodes.

As a consequence, we can define **Distributed Hash Tables** as a class of decentralized distributed systems that partition ownership of a set of keys among participating nodes, and can efficiently route messages to the unique owner of any given key. Each node is analogous to a bucket in a hash table. DHTs are typically designed to scale to large numbers of nodes and to handle continual node arrivals and failures. This infrastructure can be used to build more complex services, such as distributed file systems, p2p file sharing systems, cooperative web caching, multicast, anycast, and domain name services.

Even though this approach seems to solve the problems caused by both central index and unstructured p2p network schemes, it also raises several issues that must be taken care of:

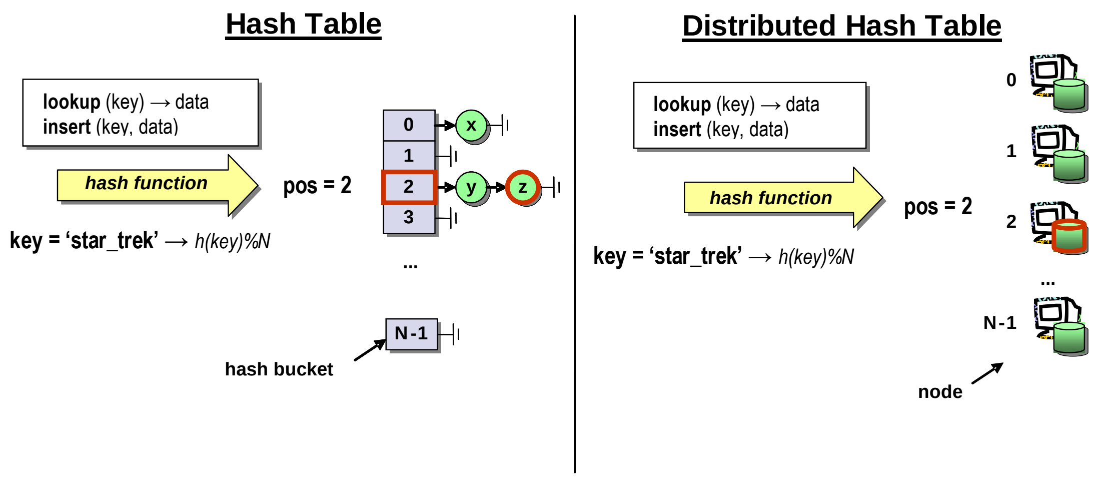


Figure 2.6. Distributed Hash Table abstraction
In a normal hash table, hash buckets are stored in local memory. However, in a DHT, hash buckets correspond to network physical nodes, and (key,value) pairs are stored on them.

- **Dinamicity**: if we use a hash function *modulus* ***N*** (where ***N*** is the approximate number of nodes in the network), virtually every key will change its location whenever a node is added or removed, since its hash function will return different results according to this formula:

$$\boxed{h(k) \bmod m \neq h(k) \bmod (m+1) \neq h(k) \bmod (m-1)}$$

  In order to solve this problem, a method called **consistent hashing** [86], adopted by the Chord [111] routing algorithm, is currently used by the major DHT designers. Consistent hashing implies defining a fixed hash space in which all hash values fall, and they do not depend on the number of peers. As a consequence, each key falls into the peer closest to its ID in the hash space, according to some proximity metric. This concept is described in greater detail when the Chord routing algorithm is explained in Section 2.3.1.3.2.

- **Size**: do we need a connection to each node in the network? This approach works well with small, static server populations. Nevertheless, when talking about wide-scale p2p networks, it is impossible to assume that every single node can be connected to all others, since the maintenance overhead would kill the entire network. The only possible solution is to allow each peer to know only a few neighbours. Messages are therefore routed through neighbours via multiple hops, using an **overlay routing** scheme.

In an efficient DHT, hosts are configured into a structured network so that mapping table lookups require a small number of hops. Designing a practical scheme along these lines is challenging because of the following desiderata:

- **Scalability**: the protocol should work for a range of networks of arbitrary size.

- **Stability**: the protocol should work for hosts with arbitrary arrival and departure times, typically with small lifetimes. This means that nodes joining and leaving should be gracefully handled, which implies repartitioning the affected keys over existing nodes, reorganizing the neighbour sets, and providing bootstrap mechanisms to connect new nodes into the DHT.

- **Performance**: the protocol should provide low latency for hash lookups and low maintenance cost in the presence of frequent joins and leaves.

- **Small diameter**: a consequence of the previous property. The node(s) responsible for each object should be reachable via a *short* path. In fact, the existing DHT models fundamentally differ only in the routing approach.

- **Flexibility**: the protocol should impose few restrictions on the remainder of the system. It should allow for smooth trade-offs between performance and state management complexity.

- **Small degree**: a consequence of the *flexibility* property. There should be a *reasonable* number of neighbours for each node.

- **Decentralized routing**: DHT routing mechanisms should be decentralized, thus avoiding any single point of failure or bottleneck.

- **Low stretch**: necessary if our DHT wishes to perform well, minimizing the ratio of DHT routing versus unicast latency.

- **Simplicity**: the protocol should be easy to understand, code, debug and deploy.

The DHT abstraction provides a minimal access interface, which is mainly data-centric. It naturally supports a wide range of applications, because it imposes very few restrictions: keys have no semantic meaning, and values are application dependent. Therefore, DHTs can be used as a decentralized data insertion and location facility. It is important to note that DHTs are not meant for storing data: they provide the means to insert it and locate it in a decentralized fashion. However, data storage logic can be built on top of the DHTs, by using its principal programming interface: ***put (key, data)*** and ***get (key)*** $\rightarrow$ ***data***.

Many systems have adopted this scheme, starting with CAN [103] and Chord [111], which were the first to appear, followed by Tapestry [116], Pastry [107], Kademlia [93], Symphony [92] and Bamboo [104]. This kind of structured peer-to-peer overlay networks are often called *Key Based Routing* ***(KBR)*** substrates, since message routing depends upon node identifiers.

In order to point out the significance of structured p2p overlay network benefits, Table 2.2 shows a comparative study of the advantages and disadvantages of the existing p2p network approaches described throughout this section.

	Centralized	Decentralized Unstructured	Partially Centralized	Decentralized Structured
Advantages	Resources are deterministically located quickly and efficiently Searches are as comprehensive as possible All users are registered on the network	Scales very well Bottlenecks are removed Fault tolerant	Scales better than the centralized approach Reduces discovery time in comparison with purely decentralized unstructured indexing systems Reduces the workload on central servers in comparison with fully centralized indexing systems	Scales very well Searches are deterministic and results are found in a number of bounded hops Fault tolerant, self-organizing, self-healing State information per node is bounded
Disadvantages	Vulnerable to censorship and technical failure Popular data become less accessible because of the load of the requests on the central server Central index may be out of data because the central server's database is only refreshed periodically Has scalability problems	Slow information discovery More query traffic on the network Searches are probabilistic	Supernodes can be surgically attacked making the network unusable Scales worse than pure decentralized approaches	Churn must be properly handled to prevent data loss

Table 2.2. Comparison of different p2p resource-location architectures

The relatively new structured p2p protocols that have emerged in recent years seem to provide a solid enough base for supporting many p2p future developments. This is the reason why we consider structured peer-to-peer key-based routing substrates to be a very interesting alternative for being the basis for our proposed generic model. These substrates provide such neat features as self-organization, self-healing, fault tolerance, efficient message routing, and many others, thus fulfilling some of the requirements we had in mind: *scalability, dynamicity, fault tolerance,* etc Table 2.3 compares the various KBR protocols and their main features and performance. These protocols are described in the following sections.

	# Neighbours	Worst Case Routing Latency	Proximity-aware	Implementation
CAN	$O(d)$	$O(dn^{1/d})$	Yes	None known
Chord	$O(\log_2 n)$	$O(\log_2 n)$	N/A in the original design. Proximity-aware Chord is described in [71]	C/C++, not official yet
Pastry	$O((2^b \log_2 n)/b)$	$O((\log_2 n)/b)$	Yes	Java/.NET
Tapestry	$O((2^b \log_2 n)/b)$	$O((\log_2 n)/b)$	Yes	Java/C++
Bamboo	$O((2^b \log_2 n)/b)$	$O((\log_2 n)/b)$	Yes	Java
Symphony	$O(2k)$	$O((\log^2 n)/k)$	No	None known

Table 2.3. Comparison of various structured overlay protocols
d refers to the *# dimensions*, *n* to the *# nodes*, *b* to the *# digit bits*, and *k* to the *# long links*.

2.3.1.3.1 CAN

Content Addressable Network (CAN) [103] is a distributed hash table structured throughout a virtual *d*-dimensional Cartesian coordinate space on a *d*-torus. This coordinate space is completely logical and bears no relation to any physical coordinate system. At any point in time, the *entire* coordinate space is dynamically partitioned among all the nodes in the system such that every node “owns” its individual, distinct zone within the overall space.

This virtual coordinate space is used to store *key,value* pairs as follows: to store a pair (K_1,V_1), key K_1 is deterministically mapped onto a point *P* in the coordinate space using a uniform hash function. The corresponding *key,value* pair is then stored at the node that owns the zone within which point *P* lies. To retrieve an entry corresponding to key K_1, any node can apply the same deterministic hash function to map K_1 onto point *P* and then retrieve the corresponding value from it. If point *P* is not owned by the requesting node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone *P* lies. Efficient routing is therefore a critical aspect of a CAN.

Nodes in the CAN self-organize into an overlay network that represents this virtual coordinate space. A node learns and maintains the IP addresses of those nodes that hold coordinate zones adjoining its own zone. This set of immediate neighbors in the coordinate space serves as a coordinate routing table that enables routing between arbitrary points in this space.

Intuitively, routing in a Content Addressable Network works by following the straight line path through the Cartesian space from source to destination coordinates. A CAN node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its immediate neighbors in the coordinate space. In a *d*-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along *d-1* dimensions and abut along one dimension. This purely local neighbour state is sufficient to route between two arbitrary points in the space: a CAN message includes the destination coordinates. Using its neighbour coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbour with coordinates closest to the destination coordinates. For a *d*-dimensional space partitioned into *n* equal zones, the average routing path length is $(d/4)(n^{1/d})$ hops and individual nodes maintain $2d$ neighbours. These scaling results mean that for a d-dimensional space, we can increase the number of nodes (and hence zones) without increasing per node state while the average path length grows as $O(n^{1/d})$.

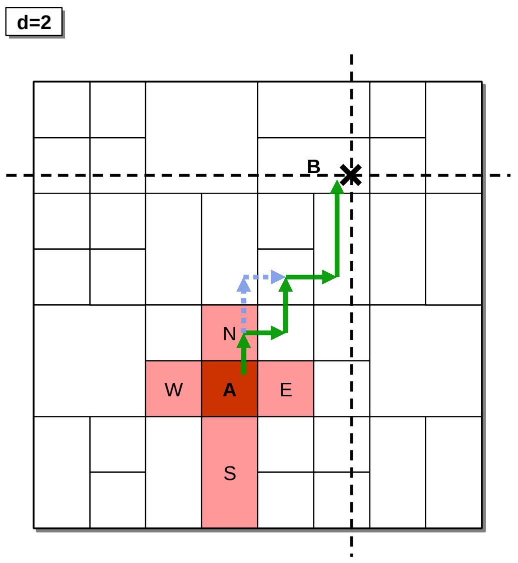


Figure 2.7. CAN Lookup Example
There are numerous path choices, so messages can be routed around failures.

Note that there are many different paths between two points in the space so, even if one or more of a node's neighbours were to crash, a node can automatically route along the next best available path.

CAN was the first distributed hash table to appear, and it was quickly overcome by other algorithms which were more efficient and required less neighbourhood state maintenance per node.

2.3.1.3.2 Chord

Chord [111] is another distributed hash table approach, contemporary to CAN. The Chord protocol specifies how to find the locations of keys, how new nodes join the system, and how to recover from the failure (or planned departure) of existing nodes. At its heart, Chord provides fast distributed computation of a hash function, and mapping keys to nodes responsible for them. It uses *consistent hashing* [86] to assign *key*, *value* pairs to their hash buckets, which are physical nodes.

With high probability the hash function balances the load (all nodes receive roughly the same number of keys). Also with high probability, when an N^{th} node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location: this is clearly the minimum requirement for maintaining a balanced load. Chord improves the scalability of consistent hashing by avoiding the requirement that every node knows about every other node. A Chord node needs only a small amount of *routing* information about other nodes. Because this information is distributed, a node resolves the hash function by communicating with a few other nodes. In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages. Chord must update the routing information when a node joins or leaves the network; a join or leave requires $O(\log^2 N)$ messages.

The *consistent hash* function uses a base hash function such as *Secure Hash Algorithm 1 (SHA-1)* to assign each node and key an m-bit *identifier*. A node's identifier is chosen by hashing the node's IP address, while a key identifier is produced by hashing the key. The identifier m must be long enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Consistent hashing assigns keys to nodes as follows. Identifiers are ordered in an *identifier circle* modulo 2^m. Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space. This node is called the *successor node* of key k, denoted by *successor(k)*. If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then *successor(k)* is the first node clockwise from k. Consistent hashing is designed to let nodes enter and leave the network with minimal disruption. To maintain the consistent hashing mapping when a node n joins the network, some keys that were previously assigned to n's successor are now assigned to n. When node n leaves the network, all of its assigned keys are reassigned to n's successor. No other changes in the assignment of keys to nodes need occur.

Each node stores information about only a small subset of the nodes in the system. in its routing table, called a *finger table*. The search for a node moves progressively closer to identifying the successor with each step. A search for the successor of f initiated at node r begins by determining if f is between r and the immediate successor of r. If so, the search terminates and the successor of r is returned. Otherwise, r forwards the search request to the largest node in its finger table that precedes f; call this node s. The same procedure is repeated by s until the search terminates.

Chord includes a simple stabilization protocol which allows it to be fault resilient, self-organizing and self-healing, and to perform acceptably even in the face of concurrent node arrivals and departures. Nevertheless, this simplicity is also one of the protocol's biggest problems, since it involves too much communication between nodes.

The authors of Chord proposed an extension to support network proximity for lower latency and higher throughput [71].

Although Chord has not been officially released, an experimental version is available for download on its website. Chord is the basis for the Cooperative File System (CFS) [70], a wide-area peer-to-peer storage system.

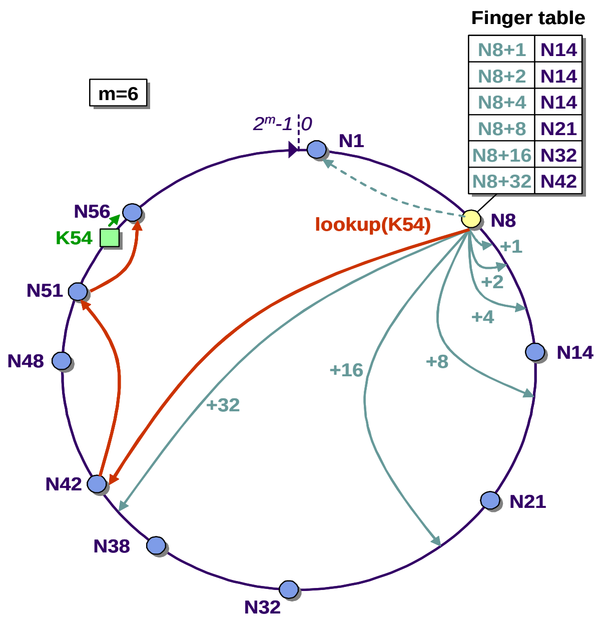


Figure 2.8. A Chord ring consisting of many nodes
Notice how the finger table is organized and how K54 is looked up following Chord's algorithm.

2.3.1.3.3 Pastry

Pastry [107] is a structured peer-to-peer network routing substrate which improved some of the limitations of the Chord protocol. A Pastry system is defined as a self-organizing overlay network of nodes, in which each node routes client requests and interacts with local instances of one or more applications. Any computer that is connected to the Internet and runs the Pastry node software can act as a Pastry node, subject only to application-specific security policies.

Each node in the Pastry p2p overlay network is assigned a 128-bit node identifier (*nodeId*). The *nodeId* is used to indicate a node's position in a circular *nodeId* space, which ranges from 0 to $2^{128} - 1$. The *nodeId* is assigned randomly when a node joins the system. It is assumed that *nodeId*s are generated such that the resulting set of *nodeId*s is uniformly distributed in the 128-bit *nodeId* space. For instance, *nodeId*s could be generated by computing a cryptographic hash of the node's public key or its IP address. As a result of this random assignment of *nodeId*s, with high probability, nodes with adjacent *nodeId*s are diverse in geography, ownership, jurisdiction, network attachment, etc.

Assuming a network consisting of N nodes, Pastry can route a given key to the numerically closest node in less than log_2bN steps under normal operation (b is a configuration parameter with a typical value of 4). Despite concurrent node failures, eventual delivery is guaranteed unless $|L|/2$ nodes with *adjacent nodeId*s fail simultaneously ($|L|$ is a configuration parameter with a typical value of 16 or 32). Therefore, Pastry routes to any node in the overlay network in $O(\log N)$ steps in the absence of recent node failures, and it maintains routing tables with $O(\log N)$ entries.

For the purpose of routing, *nodeId*s and keys are thought of as a sequence of digits with base 2^b. Pastry routes messages to the node whose *nodeId* is numerically closest to the given key. This is done in the following way. In each routing step, a node normally forwards the message to a node whose *nodeId* shares a prefix with the key that is at least one digit (or *b* bits) longer than the prefix that the key shares with the present node's ID. If no such node is known, the message is forwarded to a node whose *nodeId* shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's ID. To support this routing procedure, each node maintains a *routing table*, a *neighborhood set* and a *leaf set*.

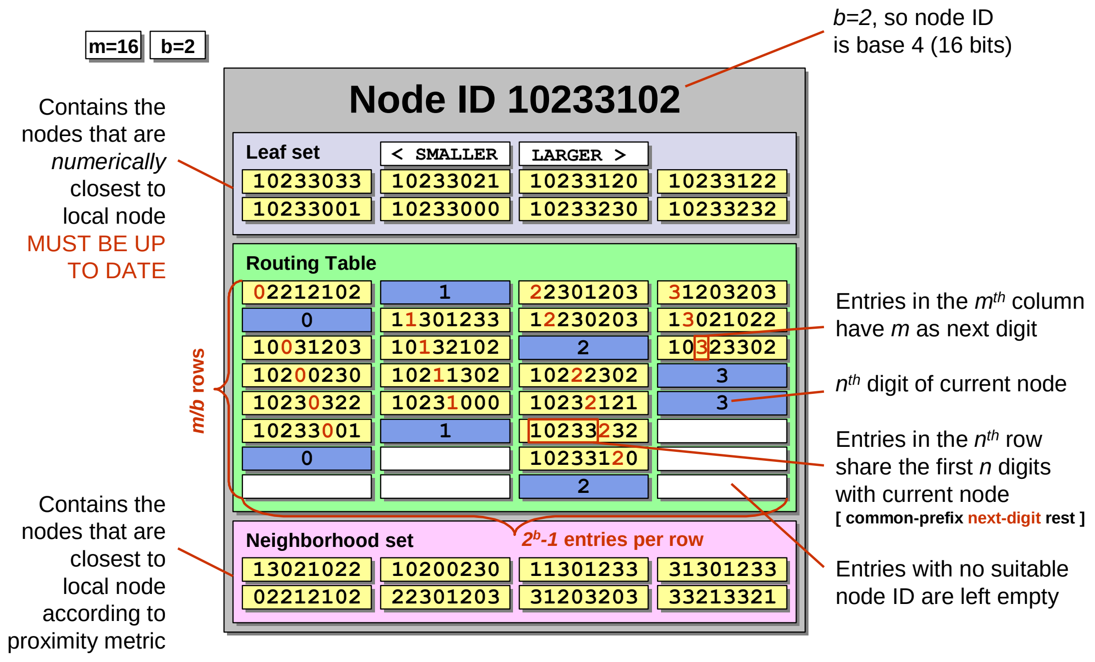


Figure 2.9. State of a hypothetical Pastry node
With nodeId 10233102, b = 2. All numbers are in base 4. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102* - *next digit* - *rest of nodeId*. The associated IP addresses are not shown.

The original version of Pastry was shipped with a minimal API which allowed programming of several applications like PAST [73] and Scribe [66]. This API was in the future extended to support the Common API for Structured Overlay Networks [72].

Pastry exports the following operations:

`nodeId = pastryInit(Credentials, Application)`

Causes the local node to join an existing Pastry network (or start a new one), initialize all relevant states, and return the local node's nodeId. The application-specific credentials contain information needed to authenticate the local node. The application argument is a handle to the application object that provides the Pastry node with the procedures to invoke when certain events happen (e.g. a message arrival).

`route(msg,key)`

Causes Pastry to route the given message to the node whose nodeId is numerically closest to the key, of all the live Pastry nodes.

Applications layered on top of Pastry must implement the following operations:

`deliver(msg,key)`
Called by Pastry when a message is received and the local node's nodeId is numerically closest to the key of all the live nodes.

`forward(msg,key,nextId)`
Called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL terminates the message at the local node.

`update(leafSet)`
Called by Pastry whenever there is a change in the local node's leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

Table 2.4. Pastry's exposed API

One important feature about Pastry is its **locality awareness**. This feature guarantees that the route chosen for a message is likely to be "good" with respect to the proximity metric. Pastry's notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance. It is assumed that the application provides a function that allows each Pastry node to determine the *distance* of a node with a given IP address from itself. A node with a lower distance value is assumed to be more desirable. An application is expected to implement this function depending on its choice of proximity metric, using network services like traceroute or Internet subnet maps, and appropriate caching and approximation techniques to minimize overhead.

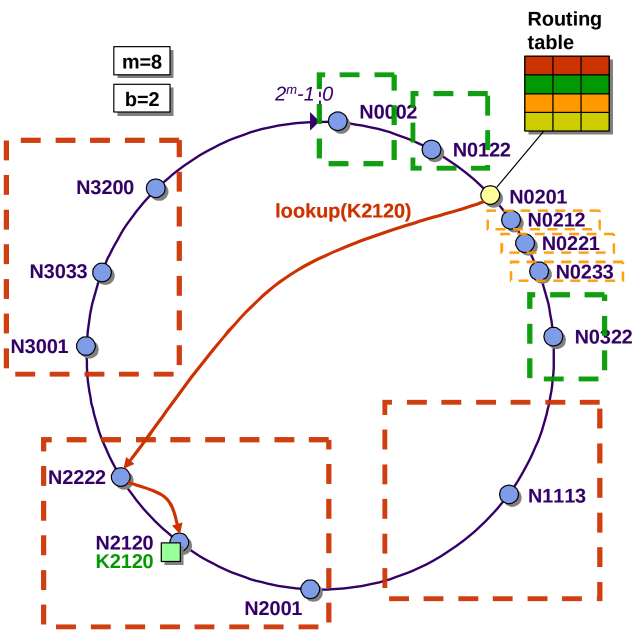


Figure 2.10. Pastry State and Lookup
For each prefix, a node knows some other node (if any) with the same prefix and different next digit. When multiple nodes are available, the topologically-closest is chosen, thus maintaining good locality properties.

2.3.1.3.4 Tapestry

Tapestry [117] is another p2p overlay network scheme that provides a routing architecture: a self-organizing, scalable, robust wide-area infrastructure that efficiently routes requests to content, in the presence of heavy load, and network and node faults. Tapestry has an explicit notion of locality, and provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and with no centralized services. Paradoxically, Tapestry uses randomness to achieve both load distribution *and* routing locality. It has its roots in the Plaxton distributed search technique [101], augmented with additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. The routing and directory information within this infrastructure is purely soft state and easily repaired. Tapestry is self administrating, fault-tolerant, and resilient under load.

Tapestry uses local routing maps at each node, called *neighbour maps*, to incrementally route overlay messages to the *destination ID* digit by digit. A node N has a neighbour map with multiple levels, in which each level represents a matching suffix up to a digit position in the ID. By definition, the *n*th node a message reaches shares a suffix of at least length *n* with the *destination ID*. To find the next router, we look at its *n + 1*th level map, and look up the entry matching the value of the next digit in the *destination ID*. Assuming consistent neighbour maps, this routing method guarantees that any existing unique node in the system will be found within at most log_bN logical hops, in a system with an *N* size namespace using IDs of base *b*.

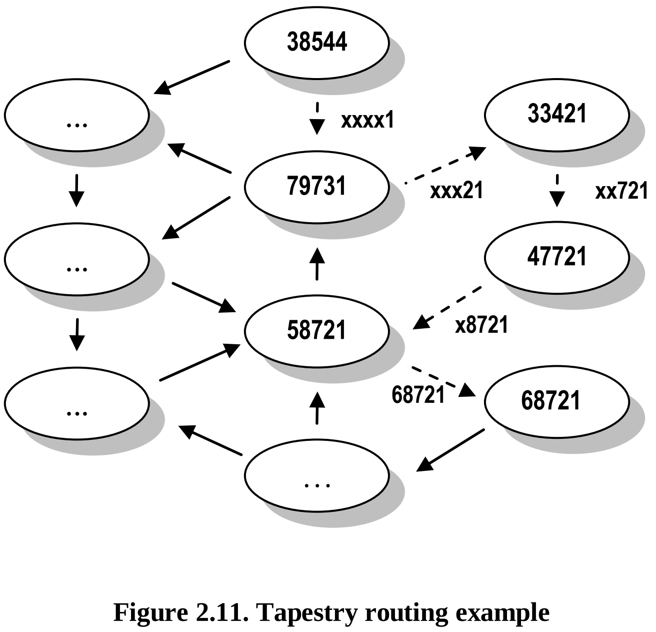


Figure 2.11. Tapestry routing example
Here we see the path taken by a message originating from node 38544 destined for node 68721 in a Plaxton mesh [101] using hexadecimal digits of length 5.

At present there is a Java implementation of Tapestry, and some interesting wide-area services have been built on top of this substrate. These include application level multicast services, like Bayeux [118], and the OceanStore Project [89], a global-scale persistent storage system.

2.3.1.3.5 Bamboo

Bamboo [104] is a DHT that is specially designed to handle the problem of *churn*, which is defined as *the continuous process of node arrival and departure*. Bamboo's design allows it to function effectively at churn rates at or higher than those observed in p2p file-sharing applications, while the maintenance bandwidth is lower than that of other DHT implementations.

The geometry and routing of Bamboo's internal design are identical to Pastry's. The difference lies in how Bamboo maintains the geometry as nodes join and leave the network and the network conditions vary. Using this Pastry-based design, Bamboo performs lookups in *O*(log *N*) hops, while the leaf set allows forward progress if the routing table is incomplete.

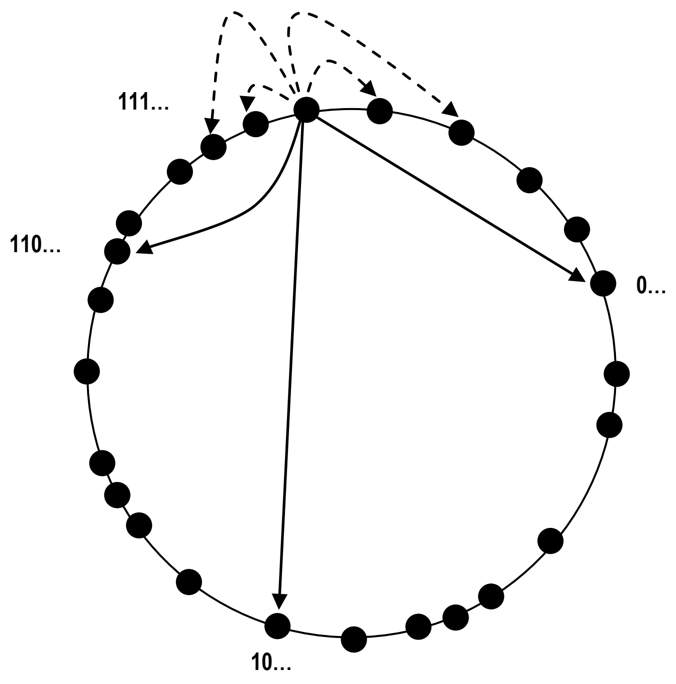


Figure 2.12. Neighbours in Pastry and Bamboo
A node's neighbours are divided into its leaf set, shown as dashed arrows, and its routing table, shown as solid arrows.

Bamboo's designers demonstrate that DHTs can handle high churn rates, and identify and explore several factors that affect the behaviour of DHTs under churn. These factors include *how DHTs recover from failures, how message timeouts are calculated during lookups, and how to choose nearby over distant neighbours*.

Therefore, they analyze the use of a *reactive recovery* failure strategy, whereby a DHT node tries to find a replacement neighbour as soon as it notices that an existing neighbour has failed. This strategy is in stark contrast to *proactive recovery*, where a node periodically shares its leaf set with every member of that set, each of which responds in kind with its own leaf set. Proactive recovery is the mechanism Bamboo currently uses.

The way in which *message timeouts* are calculated during lookups can also greatly affect performance under churn. If a node performing a lookup sends a message to a node that has left the network, it must eventually timeout the request and try another neighbour. Such timeouts are a significant component of lookup latency under churn, and they analyze several methods of computing good timeout values, including virtual coordinate schemes as used in Chord.

Finally, Bamboo's designers consider *proximity neighbour selection (PNS)*, where a DHT node with a choice of neighbours tries to select those that are nearest to itself in terms of network latency.

Bamboo can be configured to use any of the design choices that are appropriate for each factor, and the authors analyze the impact of each design choice applied to churn. A Java implementation of Bamboo is available and fully downloadable at http://www.bamboo-dht.org.

2.3.1.3.6 Symphony

The Symphony KBR protocol [92] places all hosts on a ring and equips each node with a few *long distance links*. Symphony is inspired by Kleinberg's Small World construction [88]. Kleinberg's result is extended by showing that with $k = O(1)$ links per node, it is possible to route hash lookups with an average latency of $O\left(\frac{1}{k}\log^2 n\right)$ hops. Among the advantages that Symphony has over existing DHT protocols are the following:

- *Low state maintenance*: Symphony provides low average hash lookup latency with fewer TCP connections per node than other protocols. Low degree networks reduce the number of open connections and ambient traffic corresponding to pings, keep-alives and control information. Moreover, sets of nodes that participate in locking and coordination for distributing state update are smaller sized.

- *Fault tolerance*: Symphony requires f additional links per node to tolerate the failure of f nodes before a portion of the hash table is lost. Unlike other protocols, Symphony does not maintain backup links for each long distance contact.

- *Degree vs Latency tradeoff*: Symphony provides a smooth tradeoff between the number of links per node and average lookup latency. It appears to be the only protocol that provides this tuning knob even at run time. Symphony does not

> dictate that the number of links be identical for all nodes. Neither is the number stipulated to be a function of current network size nor is it fixed at the outset. These features of Symphony provide support for heterogeneous nodes, incremental scalability, and flexibility.

Every node maintains $k \geq 1$ *long distance links*. For each such link, a node first draws a random number $x \in I$ from a probability distribution function. Then it contacts the manager of the point *x* away from itself in the clockwise direction by following Symphony's routing protocol. Finally, it attempts to establish a link with the manager of *x*. Symphony uses **bidirectional routing** as well to improve overall average latency.

The number of incoming links per node is bounded by placing an upper limit of *2k* incoming links per node. Once the limit is reached, all subsequent requests to establish a link with this node are rejected. The requesting node then makes another attempt by re-sampling from its probability distribution function. It is also ensured that a node does not establish multiple links with another node.

2.3.2 Wide-Area Application-Level Multicast

The second layer of our proposed generic model facilitates efficient communication from one node to multiple nodes. Applications often need to distribute or propagate state changes to their different views. In a distributed application, these views are usually remote clients which need to be updated whenever any changes occur due to the application logic. Therefore, it is inefficient for the sender node to send *n* messages to each of the associated clients. It is far more efficient to send *one* message which is efficiently and transparently propagated to the application's subscribed clients.

When thinking about propagating messages from one node to many, we quickly think about *IP Multicasting* [51]. IP Multicast is a method whereby a message can be sent simultaneously to several computers, instead of to one single computer. In order to do this, the message is sent to a range of addresses reserved for multicast groups (224.x.x.x-239.x.x.x) - each computer must also decide whether or not it wishes to be part of a specific group. (A computer can subscribe to the same group more than once - in such a case, each subscribing application receives a separate copy of each message received on the group IP address).

However, IP Multicasting is only supported by very few routers in the Internet. Therefore, this alternative cannot be used as a wide-area multicasting solution. In order to provide the same functionality as IP Multicasting, **application-level multicast** solutions allow such multicasting features at the application level. This way, events and messages are relayed from origin to destination by an application specific component, called the *event bus*, and its implementation, the *event system*.

It is important to make clear at this point that we refer indistinctively to *application-level multicast* and *publish/subscribe event systems* throughout this thesis. Formally, both approaches differ in the fact that publish/subscribe event systems have features that traditional application level multicast approaches do not (for example, durable messaging, or event ordering). For our work, however, since very few wide-area infrastructures of this kind exist, we group them into the same category.

Event systems have proven to be very useful middleware for distributed applications. The event bus is responsible for transmitting to subscribers events thrown by producers based on the information contained in these events.

There are many client-server and federation-based event systems which suffer from the same scalability problems found in these architectural approaches. In this section we analyze and compare state of the art in existing wide-area application-level multicast solutions.

2.3.2.1 Reference Model

We propose a reference model in which we compare some existing application-level multicast middleware platforms and see to what extent they could be used in a wide-area environment. We consider the following features:

- **Architecture**. The architecture is assumed to be implemented on top of a lower-level network infrastructure. We can classify existing event systems by their architectural model: *unstructured p2p, structured p2p, and hybrid.*

- **Usability**. It is important that the abstraction given by the middleware integrates cleanly with the application programming language so that it is easy to use. We will analyze API's complexity, multilanguage implementations and integration with remote object models.

- **Expressiveness**. Application-level multicast solutions typically follow a *publish/subscribe* approach. Therefore, *subscribers* (or *consumers*) express their interest in a specified content by subscribing to this content. From the moment of the subscription, they will start receiving events from *publishers* (or *producers*) on the content. A *publish/subscribe* event system can be classified by the different ways of specifying how to subscribe to and publish particular content:

  - *Topic-based publish/subscribe*: Participants publish notifications and subscribe to topics, which are represented by *keywords*.

  - *Content-based publish/subscribe*: A subscription scheme based on the properties of the notifications is used. In other words, events are not classified according to some pre-defined external criterion (e.g., topic name), but according to properties of the events themselves.

  - *Type-based publish/subscribe*: The name-based topic classification scheme is replaced by other filtering events according to their type. This enables the language and the middleware to be more closely integrated.

- **Security**. Several security models have been adopted by event-based middleware systems. It would be desirable for only authorized subscribers to receive notifications and to prevent unauthorized parties who are eavesdropping on the network to catch relevant information contained by the event data. Many approaches have led to server-authenticated solutions, where the publisher or the subscriber must first authenticate to a server to be able to access the system; or decoupled schemes, where security is managed in a distributed way by storing some kind of key as a field within event data.

- **Event Dispatching**. Many event-based platforms are performance oriented, meaning that message-delivery speed is highly optimized. This feature is normally interesting in real time systems which care not about reliability but performance.

- **Durable Messaging**. On the other hand, there are several event systems which are more likely to offer reliability mechanisms instead of performance. These platforms are specially important in enterprise systems which require a high degree of trust in message delivery. They may provide some kind of relaying service which can persistently store undelivered messages and deliver them when the destination is reachable again.

2.3.2.2 Event Systems Review

Now that our reference model has been described, we take into account the various aspects to analyze several event-based systems: Siena, Hermes, Narada, Bayeux, and Scribe. Table 2.5 shows the results.

- **Siena** [64]. Siena is an event-based, content-based system with a pattern expressiveness scheme and it features a hybrid architecture consisting of hierarchical client/server and unstructured peer-to-peer variations. It provides a C++ and Java API and currently it has neither security nor durable messaging mechanisms. Simulations have demonstrated that in low densities of clients which subscribe very frequently, the hierarchical client/server approach performs better, but the unstructured peer-to-peer model is more suitable when the total cost of communication is dominated by notifications.

- **Hermes** [100]. Hermes is an event-based middleware which follows a structured p2p architecture based on an overlay network (Pastry). Its expressiveness model follows the so called *type- and attribute-based* system, which is a combination of the topic- and content-based systems, providing better integration with the type model of an object-oriented programming language. A Java implementation of its API is provided and communicates through XML-defined messages, which makes it fully interoperable. Its main aim is to provide event dispatching rather than persistent events, which will be incorporated in the future. At the moment there are no access control mechanisms.

- **Narada** [68]. Narada was explicitly designed to be effective only when the multicast group size is small. It supports unstructured p2p and centralized models and, because it is *Java Message Service (JMS)* compatible, it is also a topic-based system. Its security model is designed to use distributed key management centers to achieve end-to-end integrity while ensuring that only authorized entities can publish, subscribe and decrypt messages sent to a topic. Durable messaging is supported, storing events marked as persistent to databases. Currently, its API is Java-based.

- **Bayeux** [118]. The Bayeux wide-area event dissemination system is built on top of the Tapestry [116] overlay network. Therefore, it supports very large multicast groups and follows a topic-based expressiveness model. It is unsecure unless the underlying network substrate provides a secure routing primitive. Persistent messaging is not implemented and neither is access control. The existing implementation is bundled with Tapestry with Java language. At the beginning of this thesis, there was no downloadable implementation of Bayeux.

- **Scribe** [66]. Scribe offers an overlay multicast substrate on top of the Pastry routing protocol. It introduces the concept of a topic (group identifier) to which nodes can subscribe to. Once subscribed, the node receives all event notifications that fire on that topic. Each group has a unique group identifier (*groupId*). The Scribe node with an identifier (*nodeId*) closest to the *groupId* acts as the *rendez-vous point* for the associated group. This rendez-vous point is the root of the multicast tree created for the group. Group membership is managed by creating a reverse path forwarding multicast tree rooted at the rendez-vous point. In addition to the basic multicast functionality, Scribe maintains the tree structure in the face of high levels of node failures. This is imperative if the system is going to be robust. Scribe is therefore designed for very large multicast groups and provides very good dispatching performance. It has no mechanisms for durable messaging and its security model is implemented through the secure routing functionalities of the underlying overlay network substrate (if any). No access control mechanisms have yet been implemented. Existing implementations include Java and C# languages.

	Architecture	Usability	Expressiveness	Security	Durable Msg	Event Dispatching
Siena	Hybrid / Unstructured p2p	C++ / Java API	content-based w/ patterns	None	None	Good performance
Hermes	Structured p2p	Java API + XML msgs	type- and attribute-based	None	None	Better advertisement dissemination than in Siena
Narada	Hybrid / Unstructured p2p	Java API	topic-based	Distributed key mgmt centers	Supported	Adequate for small groups
Bayeux	Structured p2p	Java API	topic-based	If secure routing is available	None	Adequate for very large groups
Scribe	Structured p2p	Java / C# API	topic-based	If secure routing is available	None	Adequate for very large groups

Table 2.5. Comparison of event systems

2.3.3 Wide-Area Persistence Systems Architectures

Another of the pillars of our proposed wide-area middleware solution is the need for a persistence service. This service should allow data to be eficiently stored and looked up in a wide-area environment.

There has been a long history of research in the area of distributed file systems and storage. Existing systems based on the client-server architecture such as AFS [84], NFS [50], xFS [57], Sprite LFS [106] and Coda [87] do not meet our goals of scalability, availability, and network performance.

We also rejected such approaches as a clustered farm of dedicated persistence servers (or even the usage of semi-static p2p approaches, like in Plethora [74]), since we want to provide a low-cost, easibly maintainable solution that makes good use of the resources on the edges of the Internet. Therefore, following the same line, we focused on wide-area persistence systems based on decentralized approaches. We studied CFS [70], OceanStore [89], PAST [73] and OpenDHT [105]. At a basic level, they provided the functionalities we required. We also reviewed a DHT that we had developed in our research group —Bunshin [95]— which we are currently using in our prototypes as our decentralized persistence engine. These wide-area persistence systems are shown in Table 2.6.

	File Size Limit	Caching / Replication	Implementation	Extra Features
CFS	None specified, even though the implementation has some problems storing files > 2.6 MB	Blocks are cached along lookup route / Blocks are replicated to *k* CFS servers after the successor	C/C++	Root block signed using private key. No explicit delete operation. Locality awareness. Quota management. Load Balancing.
OceanStore	None specified	To achieve caching and replication, data is replicated on or near the client machines where the data is accessed	Java	Replicas cooperate to share data and disseminate updates securely and efficiently.
PAST	None specified	Cache copies are left on nodes traversed by lookup or insert operations / Blocks are replicated to *k* closest neighbours	Java/.NET	Reclaim implementation. Load balancing by Pastry's locality properties.
OpenDHT	1024 bytes	Data is replicated on or near the client machines where the data is accessed	Language independent	Access by Sun RPC or XML-RPC interface.
Bunshin	None specified	Cache copies are left on nodes traversed by lookup or insert operations / Blocks are replicated to *k* closest neighbours	Java	Multifield and Multicontext features. Keyword Search. Key links. Link notifications.

Table 2.6. Comparison of the wide-area persistence systems analyzed

2.3.3.1 CFS

The Cooperative File System (CFS) [70] is a peer-to-peer read-only storage system that provides provable guarantees for the efficiency, robustness, and load-balance of file storage and retrieval. CFS does this with a completely decentralized architecture that can scale to large systems. CFS servers provide a distributed hash table (which they refer to as *DHash*) for block storage. CFS clients interpret *DHash* blocks as a file system. *DHash* distributes and caches blocks at a fine granularity to achieve load balance, uses replication for robustness, and decreases latency with server selection. *DHash* finds blocks using the Chord [111] location protocol, which operates in time logarithmic in the number of servers.

A CFS file system exists as a set of blocks distributed over the CFS servers available. CFS client software interprets the stored blocks as file system data and meta-data and presents an ordinary read-only file-system interface to applications. The core of the CFS software consists of two layers, *DHash* and Chord. The *DHash* layer performs block fetches for the client, distributes the blocks among the servers, and maintains cached and replicated copies.

Layer	**Responsibility**
`FS`	Interprets blocks as files; presents a file system interface to applications
`DHash`	Stores unstructured data blocks reliably
`Chord`	Maintains routing tables used to find blocks

Table 2.7. CFS software layering

DHash provides load balance for popular large files by arranging to spread the blocks of each file over many servers. To balance the load imposed by popular small files, *DHash* caches each block at servers likely to be consulted by future Chord lookups for that block. *DHash* supports pre-fetching to decrease download latency, and replicates each block at a small number of servers, to provide fault tolerance. *DHash* enforces weak quotas on the amount of data each server can inject, to deter abuse. Finally, it also enables the number of *virtual servers* per server to be controlled, which in turn controls how much data a server must store on behalf of others.

CFS provides consistency and integrity of file systems by adopting the SFSRO [78] file system format. This protocol is the base of a fast and secure distributed read-only file system. CFS extends SFSRO by providing the following desirable properties:

- **Decentralized control.** CFS servers need have no administrative relationship with publishers. CFS servers can be ordinary Internet hosts whose owners volunteer spare storage and network resources.
- **Scalability.** CFS lookup operations logarithmically depend on the number of network servers.
- **Availability.** A client can always retrieve data as long as it is not trapped in a small partition of the underlying network, and as long as one of the data's replicas is reachable using the underlying substrate. This is true even if servers

are constantly joining and leaving the CFS system. CFS places replicas on servers likely to be at unrelated network locations to ensure independent failure.

- **Load balance.** CFS ensures that the burden of storing and serving data is divided among the servers in rough proportion to their capacity. It maintains load balance even if some data are far more popular than others, through a combination of caching and spreading each file's data over many servers.

- **Persistence.** Once CFS commits to storing data, it keeps it available for at least an agreed period.

- **Quotas.** CFS limits the amount of data that any particular IP address can insert into the system. This provides a degree of protection against malicious attempts to exhaust the system's storage.

- **Efficiency.** Clients can fetch CFS data with a delay that is similar to that of FTP because of CFS' use of efficient lookup algorithms, caching, pre-fetching, and server selection.

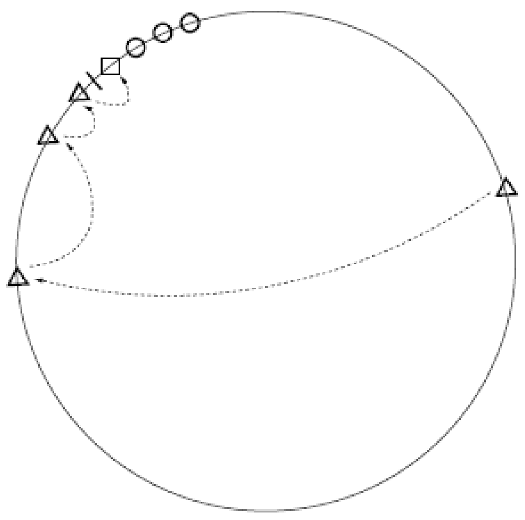


Figure 2.13. CFS replication algorithm
The placement of an example block's replicas and cached copies around the Chord identifier ring. The block's ID is shown with a tick mark. The block is stored at the successor of its ID, the server denoted with the square. The block is replicated at the successor's immediate successors (the circles). The hops of a typical lookup path for the block are shown with arrows; the block will be cached at the servers along the lookup path (the triangles).

To sum up, and with reference to the properties we are interested in for our review, we observe that CFS does not explicitly fix a file size limit. However, the implementation (in C/C++) has some problems storing files that are larger than 2.6 MB. CFS provides caching features, by caching blocks along the lookup route, and also block replication among *k* CFS servers after the successor. It provides some extra features that include root block signing using private key, no explicit delete operation, locality awareness, quota management and a load balancing algorithm.

2.3.3.2 OceanStore

OceanStore [89] is another infrastructure that allows wide-area access to persistent information through a peer-to-peer network. Since this kind of networks are comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data can be cached anywhere, anytime. Additionally, monitoring of usage patterns allows adaptation to regional outages and denial of service attacks; monitoring also enhances performance through proactive movement of data. OceanStore is built on top of the Tapestry [116] peer-to-peer substrate, and its prototype implementation is called Pond.

The OceanStore system has two design goals that differentiate it from similar systems: the ability to be constructed from an *untrusted infrastructure* and the support of *nomadic data*.

- **Untrusted Infrastructure:** OceanStore assumes that the infrastructure is fundamentally *untrusted*. Servers may crash without warning or leak information to third parties. This lack of trust is inherent in the utility model and is different from other cryptographic systems. Only clients can be trusted with cleartext—all information that enters the infrastructure must be encrypted. However, rather than assuming that servers are passive repositories of information (such as in CFS), in the case of OceanStore servers are able to participate in protocols for distributed consistency management. To this end, it is assumed that most of the servers work correctly most of the time, and that there is one class of servers that can be trusted to carry out protocols on the client's behalf (but not trusted with the content of a client's data). This *responsible party* is financially responsible for the integrity of client data.

- **Nomadic Data:** In a system like OceanStore, locality is of extreme importance. Thus, data can be cached anywhere, anytime. This policy is called *promiscuous caching*. Data that is allowed to flow freely is called *nomadic data*. Note that nomadic data is an extreme consequence of separating information from its physical location. Although promiscuous caching complicates data coherence and location, there is greater flexibility so that locality is more easily optimized and consistency traded off for availability. To exploit this flexibility, continuous *introspective* monitoring is used to discover tacit relationships between objects. The resulting meta-information is used for locality management.

The fundamental unit in OceanStore is the *persistent object*. Each object is named by a *globally unique identifier*, or GUID. Objects are replicated and stored on multiple servers. This replication provides availability in the presence of network partitions and durability against failure and attack. A given replica is independent of the server on which it resides at any one time; these are referred as *floating replicas*.

A replica for an object is located through one of two mechanisms. First, a fast, probabilistic algorithm attempts to find the object near the requesting machine. If the probabilistic algorithm fails, location is left to a slower, deterministic algorithm. Objects in the OceanStore are modified through *updates*. Updates contain information about what changes to make to an object and the assumed state of the object under which these changes are made. In principle, every update to an OceanStore object creates a new version. While more expensive to implement than update-in-place consistency, consistency based on versioning provides for cleaner recovery in the face of system failures.

OceanStore objects exist in both *active* and *archival* forms. An active form of an object is the latest version of its data together with a handle for update. An archival form represents a permanent, read-only version of the object. Archival versions of objects are encoded with an erasure code and spread over hundreds or thousands of servers; since data can be reconstructed from *any* sufficiently large subset of fragments, the result is that nothing short of a global disaster could ever destroy information.

For our review, there is no file size limit specified in OceanStore, caching and replication is achieved by replicating data on or near the client machines where the data is accessed, and the extra features included are a Java implementation of OceanStore, replica cooperation in data sharing and secure and efficient dissemination of updates.

2.3.3.3 PAST

The PAST [108] system is composed of nodes connected to the Internet, where each node is capable of initiating and routing client requests to insert or retrieve files. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing overlay network. Inserted files are replicated on multiple nodes to ensure persistence and availability. With high probability, the set of nodes over which a file is replicated is diverse in terms of geographic location, ownership, administration, network connectivity, rule of law, etc. Additional copies of popular files may be cached in any PAST node to balance query load.

While PAST offers persistent storage services, its access semantics differ from those of a conventional filesystem. Files stored in PAST are associated with a quasi-unique *fileId* that is generated at the time of the file's insertion into PAST. Therefore, files stored in PAST are *immutable* since a file cannot be inserted multiple times with the same *fileId*. Files can be shared at the owner's discretion by distributing the *fileId* (potentially anonymously) and, if necessary, a decryption key. PAST does not support a *delete* operation. Instead, the owner of a file may *reclaim* the storage associated with a file, which does not guarantee that the file is no longer available. These weaker semantics avoid agreement protocols among the nodes storing the file. PAST is built upon Pastry, thus ensuring that client requests are reliably routed to the appropriate nodes. Client requests to *retrieve* a file are routed to a node that is *close in the network* (network proximity is based on a scalar metric, such as the number of IP hops, geographic distance, or a combination of these and other factors) to the client that issued the request, among all live nodes that store the requested file. The number of PAST nodes traversed while routing a client request is at most logarithmic in the total number of PAST nodes in the system under normal operation.

PAST maintains the invariant that *k* copies of each inserted file are maintained on different nodes within a leaf set. Therefore, storage nodes and files in PAST are all assigned uniformly distributed identifiers, and replicas of a file are stored at the *k* nodes whose *nodeIds* are numerically closest to the file's *fileId*.

The PAST system exports the following set of operations to its clients:

`fileId = insert(name, owner-credentials, k, file)`

stores a file at a user-specified number *k* of diverse nodes within the PAST network. The operation produces a 160-bit identifier (*fileId*) that can be used subsequently to identify the file. The *fileId* is computed as the secure hash (SHA-1) of the file's name, the owner's public key, and a randomly chosen salt. This choice ensures (with very high probability) that *fileId*s are unique. Rare *fileId* collisions are detected and lead to the rejection of the file inserted last.

`file = lookup(fileId)`

reliably retrieves a copy of the file identified by *fileId* if it exists in PAST and if one of the *k* nodes that store the file is reachable via the Internet. The file is normally retrieved from a live node "near" the PAST node issuing the lookup (in terms of the proximity metric), among the nodes that store the file.

`reclaim(fileId, owner-credentials)`

reclaims the storage occupied by the *k* copies of the file identified by *fileId*. Once the operation completes, PAST no longer guarantees that a lookup operation will produce the file. Unlike a delete operation, reclaim does not guarantee that the file is no longer available after it is reclaimed. These weaker semantics avoid complex agreement protocols among the nodes storing the file.

Table 2.8. PAST exposed API

To sum up, PAST does not specify a file size limit, there are Java and .NET implementations of PAST, cache copies are left on nodes traversed by lookup or insert operations, and replication is achieved by replicating blocks to *k*'s closest neighbours. As extra features we can get a *reclaim* implementation, and load balancing by Pastry's locality properties.

2.3.3.4 OpenDHT

OpenDHT [105] is a publicly accessible distributed hash table (DHT) *service*. Unlike the usual DHT model, clients of OpenDHT do not need to run a DHT node in order to use the service. Instead, they can issue *put* and *get* operations to any DHT node, which processes the operations on their behalf. No credentials or accounts are required to use the service, and the available storage is fairly shared amongst all active clients. This *service model* of DHT usage greatly simplifies deploying client applications. By using OpenDHT as a highly-available naming and storage service, clients can ignore the complexities of deploying and maintaining a DHT and instead concentrate on developing more sophisticated distributed applications. OpenDHT's simple put-get interface is accessible over both Sun RPC and XML RPC. As such, the service is easy to access from virtually every programming language and from behind almost all NATs and firewalls. OpenDHT is built on prior efforts. The Bamboo overlay network implementation is used as its routing layer, and a soft-state storage layer is implemented on top.

The main difference between OpenDHT and other persistent DHT implementations is that OpenDHT is currently *alive and kicking*. It is deployed on the PlanetLab [69] network and, therefore, is accessible worldwide. Specifically, its goal is to provide a free, public DHT service that runs on PlanetLab *today*. In the longer-term, it is envisioned that this free service could evolve into a competitive commercial market in

DHT service. Because OpenDHT operates on a set of infrastructure nodes, applications need not concern themselves with DHT deployment, or run application-specific code on these infrastructure nodes. This is quite different from most other uses of DHTs, in which the DHT code is invoked as a library on each of the nodes running the application. The library approach is very flexible, as application-specific functionality can be put on each of the DHT nodes, but each application must deploy its own DHT. The service approach adopted by OpenDHT offers the opposite tradeoff: less flexibility in return for less deployment burden. OpenDHT provides a home for applications that are more suited to this compromise.

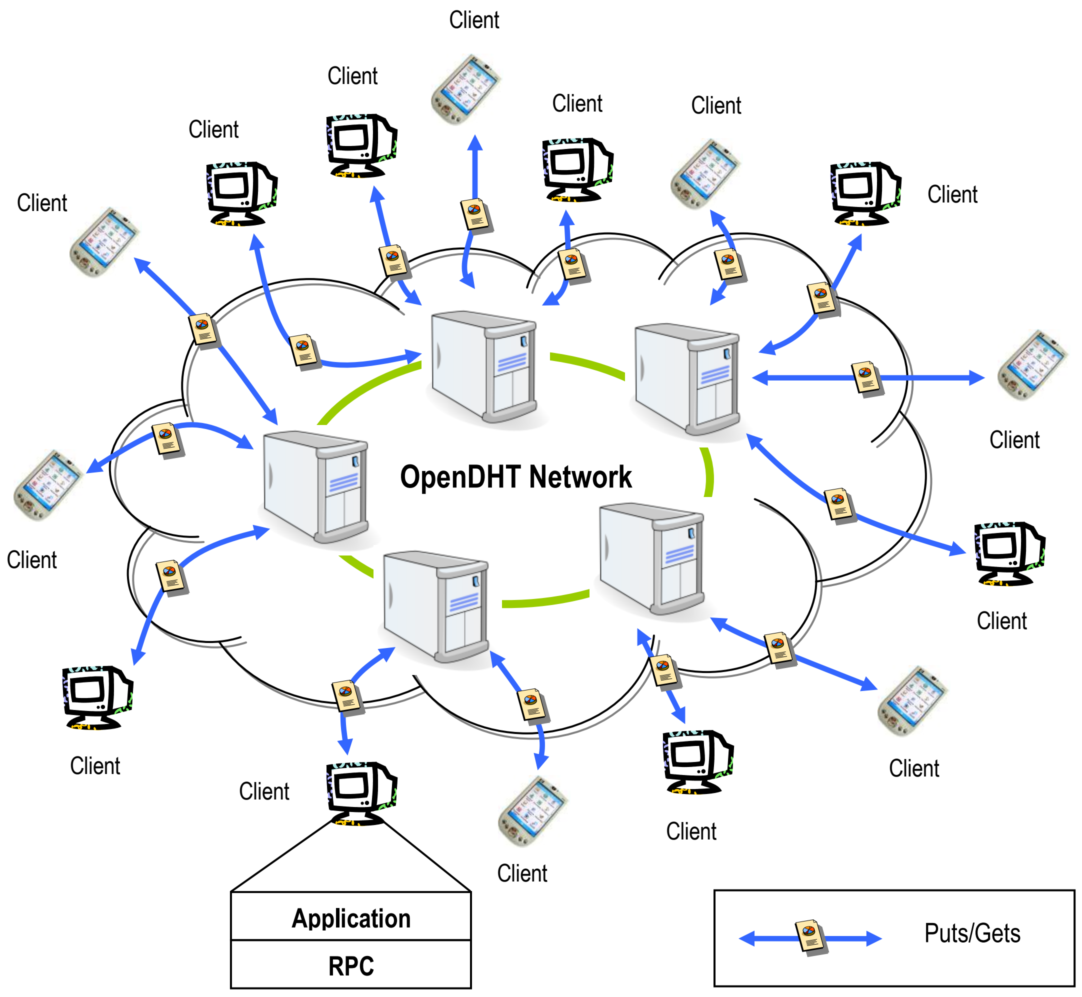


Figure 2.14. OpenDHT architecture

However, OpenDHT is limited to storing 1024-byte limit values. If larger values are to be stored, they have to be split into blocks (*à la* CFS).

Summarizing, OpenDHT is limited to 1024 byte files (file size can be increased by concatenating 1024 byte blocks on the DHT), data is replicated on or near the client machines where the data is accessed. It is language independent, since OpenDHT is accessed via Sun RPC or an XML-RPC interface.

2.3.3.5 Bunshin

Bunshin [95] is another DHT persistent system implemented in our research group [7]. It runs on top of the Pastry overlay substrate, and it came about because we had previously experimented with such practical DHT implementations as PAST which, however, did not guarantee a solution to the problem of key movement, when nodes joined or left the network. Sometimes, keys were not found or they simply returned a null value. Bunshin solves this problem by using an active replication scheme, as well as introducing new features which enrich this DHT layer.

Bunshin provides the standard *put()* and *get()* methods of a DHT, replicating data into *REPLICA_FACTOR* replicas. Possible node replicas are obtained via the underlying layer *replicaSet()* method. It also notifies of node joins / leaves by the *update()* method. However, to avoid event losses and, consequently, *update()* misses, each node periodically checks:

- whether all keys currently stored by the current node already belong to it. If not, the key/value pairs affected are inserted into the newly corresponding node.
- whether all replicas of the current node's key/value pairs are still alive, and their number is equal to *REPLICA_FACTOR*. If they are not, a new set of replicas must be chosen and updated accordingly.
- whether the owner of all the replicas assigned to the current node is still alive. If it is not, we may find that the new owner knows nothing about our key. Here we can choose between a variety of policies, but the main idea is that nodes notify the new owner and follow a versioning control to choose the newest one.

Bunshin is not a conventional DHT in the key/value sense of structuring, but it makes it possible to deal with data as if they were a hash table too: that is to say, that for a specified key we can hold as many values as we want, identified with a subkey that we call ***field***. The other possibility, at a higher level of abstraction, is that a node can not only be seen as the container of one bucket, but of as many of them as we need. Therefore, a bucket lies within a ***context***. If the application needs to have different buckets it will not be forced to instantiate a new Bunshin application for each one of them, but to indicate the context in which it will work. In summary, N buckets are arranged with values of M fields, depending on the particular application needs. These functionalities are called **multifield** and **multicontext**, respectively.

Moreover, Bunshin also provides an upper **search engine** layer, which gives the following services:

- *Keyword service* – allows keywords associated to a determinate key-value pair to be inserted. Querying these keywords returns a list of the contents associated to them.
- *Key link service* – allows maintenance of input and output link lists to be maintained. This way, it is easy to relate the keys of the inserted metadata. Therefore, we can easily find out how many links are pointing to / from a specified key.

- *Link notification service* – every time a link is added or removed, applications are notified by this service.

Bunshin's persistence engine stores values in a node's memory, and in a node's file system.

To sum up, Bunshin does not impose file size limitations, cache copies are left on nodes traversed by lookup or insert operations, blocks are replicated to *k* closest neighbours, and there is a Java implementation of this DHT. The extra features included are multifield and multicontext, keyword search, key links and link notifications.

2.4 Related Work in Wide-Area Middleware Systems

In the previous section we have reviewed the state of the art in the various layers that make up our middleware proposal. Now we are going to analyze existing wide-area middleware systems, focusing on how they interact with each of the layers described in the background. We shall see that most of these approaches do not provide developers with an abstraction level that makes their applications easy to implement. However, before describing such systems, we provide a brief overview of distributed middleware techniques available to date.

2.4.1 Traditional Distributed Middleware Overview

Several middleware approaches can be used to design distributed applications. In chronological order, *distributed objects* was the first approach to emerge followed by *distributed reusable components*. The future trend seems to follow another paradigm, called *service oriented architectures*.

The *distributed object* paradigm offers two main important facilities when building open systems: firstly, it defines the organization of a system as a collection of objects which collaborate to perform a common task; secondly, it achieves a high degree of reutilization, due to the mechanisms of inheritance and polymorphism. Distributed object middlewares offer a set of common services to deal with such objects. These may include *naming services*, to help locate objects, *persistence services*, to allow the persistence of objects' state information, *object communication services*, and many others.

Nevertheless, despite its demonstrated efficiency in software systems, this paradigm does not clearly express the distinction between the computational and the compositional points of view of an application. Moreover, the object's view prevails over the component's view as an entity that must be composed by others to conform an application. Based on this idea, the notion of *component orientation* emerges as an extension to object orientation [112].

Compositional models extend the object oriented paradigm by adding new abstractions and concepts that express composition relationships between system components. A component oriented environment emphasizes the definition of standard interfaces which indicate how their components must be used. These interfaces define the component as a collection of methods invoked whenever a service is required.

The use of components is based on the *plug-and-play* concept: that is, we can *connect* a component as a part of an application without needing to change it for it to start working. This idea applies to many commercial products, and eases the building of configurable applications whose functionalities depend on their aggregated components.

Normally, component-based software is built on top of frameworks, which provide the life cycle services required by components. These frameworks may also manage component activation and passivation, persistence, naming, etc.

One step beyond components, *service oriented architectures* (SOA) expresses a perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation.

SOA can also be regarded as a style of information systems architecture that enables applications to be created that are built by combining loosely coupled and interoperable services. These services interoperate on the basis of a formal definition which is independent of the underlying platform and programming language. SOA can support integration and consolidation activities within complex enterprise systems, but SOA does not specify or provide a methodology or framework for documenting capabilities or services.

2.4.2 Globe

A number of companies have advocated peer-to-peer solutions to problems such as distribution of streaming media, web hosting, distributed auctions, etc. There is renewed interest in a large body of distributed systems research on resource sharing and collaboration in both LAN and WAN environments. In particular, the so called *WAN-OS* projects such as Legion [90] or Globe [115] are well suited for supporting arbitrary p2p applications since their goal is to make the Internet look like a single parallel machine by hiding (to the extent desired by the developer) all the complexities associated with vastly different machines, local operating systems, communication protocols, local resource management, access control, and security policies.

The Globe System [115] aims to support numerous users, clients and objects through the Internet. One of the most important features of Globe is its distributed shared object concept, which allows objects to be replicated and distributed between different machines. However, this means that Globe can only provide one invocation type: that of synchronous calls. It has neither support for notifications nor callbacks. Therefore, there is no concept of one-to-many calls in Globe and, as a consequence, there is no efficient group communication service.

One of Globe's important hot spots is its wide-area location service which maps object identifiers to the locations of moving objects. Globe arranges the Internet as a hierarchy of geographical, topological, or administrative domains, effectively constructing a static world-wide search tree, much like DNS. Information about an object is stored in a particular leaf domain, and pointer caches provide search short cuts. The Globe system handles high load on the logical root by partitioning objects among multiple physical root servers using hash-like techniques.

The distributed objects used in a Globe application are hosted on a collection of *Globe object servers*. An object server is a user-level process that can host *local representatives* (i.e. replicas) of a large number of objects. A machine running a Globe object server is known as a *Globe site*. In addition to an object server, a Globe site may run additional processes that implement the Globe naming service or allow non-Globe clients to access distributed shared objects.

Globe builds fault tolerance and high availability through its *replication subobject*, which is responsible for keeping the state of object replicas consistent according to the consistency model chosen for any particular distributed object. Object state is periodically checkpointed to disk in order to be able to re-create objects and their replicas in case of a node crash.

Dynamicity is currently not supported on Globe because it has not been designed to cope with a regularly changing set of participants. As a consequence, a Globe site requires more than 20 external libraries or tools, most of which have to be installed by the site administrator himself, as they are not typically installed on most systems. Installing all these packages is time-consuming and tedious. Furthermore, after installing the packages the site administrator has to configure the various components of the Globe middleware, which involves setting a lot of parameters. Finally, to finish the configuration the administrator must register his site with Globe's group, which is a manual process that takes some time.

Globe was designed with the requirement of usability in mind [58], and it therefore provides an easy-to-use API so distributed objects and applications are simple to develop.

2.4.3 Legion

Legion [90] provides an object based service model so that objects can be replicated and located arbitrarily transparently. Legion's scheme is similar to the one used by Globe for separating the object name from its address. However, the main difference between both systems is the way objects are considered. In Globe, objects are assumed to be physically distributed over many resources in the system. However, in Legion, objects can be physically distributed over multiple physical resources, but are expected to physically reside in a single address space. These conflicting views of objects result in different mechanisms for object communication: Globe loads part of the object (called a local object) into the caller's address space, whereas Legion sends a message of a specified format from the caller to the callee. Further, although Legion may provide support for data streams, its purpose is broader than standalone support for the communication paradigm.

Legion does not comply with the dynamicity requirement, which is not targeted to the edges of the Internet. Legion's main aim is to provide a worldwide supercomputer, composed of relatively static powerful servers, much like the target of Grid Computing. It provides fault tolerance mechanisms, but keeps in mind that failures will not be as frequent as in a highly dynamic environment.

The high availability requirement is partly implemented, since it provides object replication primitives, but the semantics of group communication are left to the developer. It also provides fault tolerance mechanisms, and programming Legion applications is not a trivial task.

Finally, Legion does not provide any kind of load balancing mechanisms since, as stated by the authors, an object could easily become a bottleneck and limit application performance. Our idea of load balancing would perfectly suit this scenario. Therefore, a vast number of requests should be dealt with by being distributed among active object replicas transparently by the middleware. Legion does not do so in a transparent way, and this responsibility is left to the application developer.

2.4.4 JxtaJeri

JxtaJeri [30] is an integration of JxTA [39] and Jini [48] that lets programmers use the Java RMI programming model to invoke services over a JxTA p2p network. This package uses JxTA's sockets to implement a *Jini Extensible Remote Invocation (JERI)* [29] transport. JxtaJeri enables a service to expose its remote interfaces over the JxTA network. A programmer can use the higher-level remote procedure call model to construct services. JxtaJeri is based on unstructured JxTA networks, which means that resource location is probabilistic.

This middleware does not allow for implicit load balancing, high availability or fault tolerance mechanisms. Developers have to explicitly deal with these issues.

By using JxTA protocols, JxtaJeri can benefit from many advantages such as network address translation and firewall traversal. There is very little information about this project, and at the time of the writing of this thesis even its main website was unavailable.

2.4.5 The Peer-to-Peer Sockets Project

The Peer-to-Peer Sockets Project [38] (P2P Sockets) reimplements Java's standard `Socket`, `ServerSocket`, and `InetAddress` classes to work on the JxTA p2p network, rather than on the standard TCP/IP network. It also includes ports of many popular web packages, such as the Jetty web server [28], the Apache XML-RPC client and server libraries [6], and the Apache Jasper JSP engine [5], to run on the P2P Sockets framework.

P2P Sockets support high availability, dynamicity and fault tolerance requirements, and since they are based on JxTA, they can easily do things that ordinary server sockets and sockets cannot handle. First, creating server sockets that can fail-over and scale is easy with P2P Sockets. Many different peers can start server sockets for the same host name and port, such as `www.deim.urv.cat` on port `80`. When a client opens a P2P socket to `www.deim.urv.cat` on port `80`, it will randomly connect to one of the machines that is hosting this port. All of these server peers might be hosting the same web site which for example, makes it very easy to partition client requests across different server peers or to recover from losing one server peer.

Even though P2P Sockets provides many interesting services, it lacks the usability and the level of abstraction required for higher-level developers. Programming at the socket layer is too low-level when distributed applications are being developed. Therefore, it is hard to achieve common services like group communication, proximity awareness, and even data persistence, without needing to hard code them. As a consequence, we believe P2P Sockets does not provide the level of abstraction required to develop complex wide-area distributed applications.

2.4.6 The Common API for Structured Overlays

The Common API [72] for Structured p2p Overlays attempts to identify the fundamental abstractions provided by structured overlays and to define APIs for the common services they provide. As the first step, a *key-based routing API (KBR)* is defined, which represents basic (tier 0) capabilities that are common to all structured overlays. The KBR is easily implemented by existing overlay protocols and it makes it possible to efficiently implement higher level services and a wide range of applications. Thus, the KBR is the common denominator of services provided by existing structured overlays. In addition, a number of higher level (tier 1) abstractions are identified and it is shown how they can be built upon the basic KBR. These abstractions *include distributed hash tables* (DHT), *group anycast and multicast* (CAST), and *decentralized object location and routing* (DOLR).

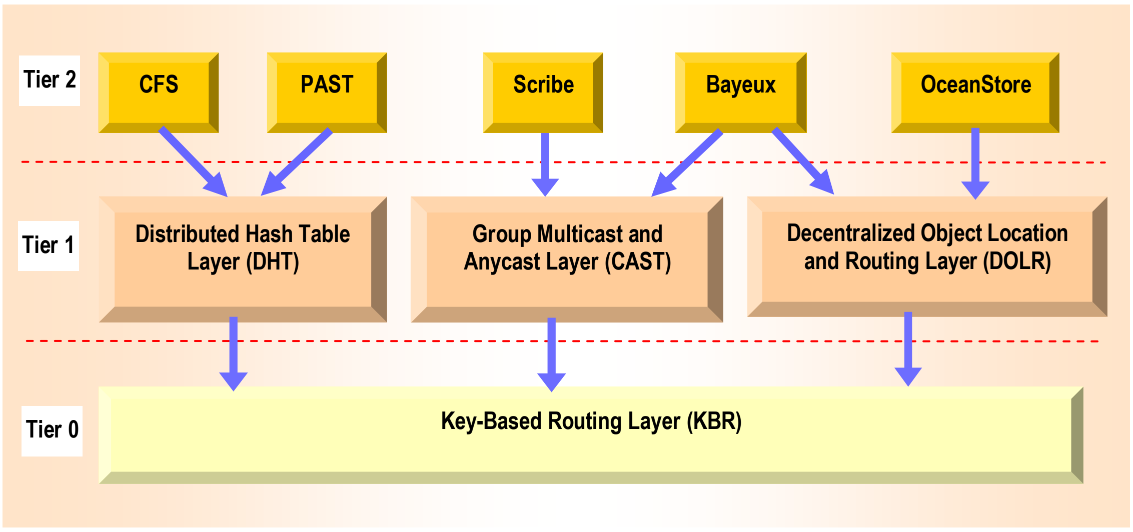

Tier 2
CFS
PAST
Scribe
Bayeux
OceanStore
Tier 1
Distributed Hash Table Layer (DHT)
Group Multicast and Anycast Layer (CAST)
Decentralized Object Location and Routing Layer (DOLR)
Tier 0
Key-Based Routing Layer (KBR)

Figure 2.15. Common API Diagram
Basic abstractions and APIs, including Tier 1 interfaces: distributed hash tables (DHT), decentralized object location and routing (DOLR), and group anycast and multicast (CAST).

Figure 2.15 illustrates how these abstractions are related. Key-based routing is the common service provided by all systems at tier 0. At tier 1, we have higher level abstractions provided by some of the existing systems. Most applications and higher-level (tier 2) services use one or more of these abstractions.

The DHT abstraction provides the same functionality as a traditional hash table, by storing the mapping between a key and a value. This interface implements a simple store and retrieve functionality, where the value is always stored at the live overlay node(s) to which the key is mapped by the KBR layer. Values can be objects of any type.

The DOLR abstraction provides a decentralized directory service. Each object replica (or endpoint) has an *objectID* and may be placed anywhere within the system. Applications announce the presence of endpoints by *publishing* their locations. A client message addressed with a particular *objectID* will be delivered to a *nearby* endpoint with this name.

The CAST abstraction provides scalable group communication and coordination. Overlay nodes may join and leave a group, multicast messages to the group, or anycast a message to a member of the group. Because the group is represented as a tree, membership management is decentralized. Thus, CAST can support large and highly dynamic groups. Moreover, if the overlay that provides the KBR service is proximity aware, then multicast is efficient and anycast messages are delivered to a group member near the anycast originator.

Table 2.9 summarizes the methods all these interfaces must implement

DHT	DOLR	CAST
put (key, data)	*publish (objectId)*	*join (groupId)*
remove (key)	*unpublish (objectId)*	*leave (groupId)*
value = get (key)	*sendToObj (msg, objectId, [n])*	*multicast (msg, groupId)* *anycast (msg, groupId)*

Table 2.9. Common API's Tier 1 API
All services defined at tier 1 require interfacing with the lower key-based routing API layer (tier 0), which is the core all structured overlay network implementations must provide.

Table 2.10 briefly describes the methods and their main use:

Method signature	Method type	Description
`void route (key: K, msg: M, NodeHandle hint)`	Message routing	Forwards a message *M*, towards the root of key *K*. The optional *hint* argument specifies a node that should be used as a first hop in routing the message.
`void forward (key: K, msg: M, NodeHandle nextHopNode)`	Message routing	This upcall is invoked at each node that forwards message *M*, including the source node, and the key's root node. The upcall informs the application that message *M* with key *K* is to be forwarded to *nextHopNode*.
`void deliver (key: K, msg: M)`	Message routing	This function is invoked on the node that is the root for key *K* upon the arrival of message *M*.
`NodeHandle[] local_lookup (key: K, int: num, boolean: safe)`	State access routing	This call produces a list of nodes that can be used as next hops on a route towards key *K*, such that the resulting route satisfies the overlay protocol's bounds on the number of hops taken.
`NodeHandle[] neighborSet (int: num)`	State access routing	This operation produces an unordered list of nodehandles that are neighbours of the local node in the ID space. Up to *num* nodehandles are returned.
`NodeHandle[] replicaSet (key: K, int: max_rank)`	State access routing	This operation returns an unordered set of nodehandles on which replicas of the object with key *K* can be stored.
`update (NodeHandle: n, boolean: joined)`	State access routing	This upcall is invoked to inform the application that node *n* has either joined or left the neighbour set of the local node as that set would be returned by the *neighborSet* call.
`boolean range (NodeHandle: N, rank: r, key: lkey, key: rkey)`	State access routing	This operation provides information about ranges of keys for which node *N* is currently a *r-root*

Table 2.10. Common API's Tier 0 API

As we can clearly observe, the Common API provides the upper levels with three interaction layers which perfectly fit into the layers we have defined throughout this chapter: a **wide-area routing layer** (KBR), an **application-level multicast layer** (CAST), and an **object persistence layer** (DHT). Even though it defines the same layers we propose, the Common API does not comply with all of our requirements for a wide-area middleware. This is because the approach is far too low-level. Developers will find it difficult to deal with messages, updates, puts and gets. It is clear that the Common API provides the primitives required to *implement* many services, but not the services themselves. Therefore, there is no implicit load balancing service, no fault tolerance, and no high availability.

2.4.7 Grid Computing

The term *Grid computing* [62] originated in the early 1990s as a metaphor to suggest that accessing computer power is as easy as accessing an electric power grid.

The Grid can be defined as *the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources* *distributed across multiple administrative domains based on the resources availability, capacity, performance, cost and users' quality-of-service requirements* [27].

The ideas of the Grid were brought together by **Ian Foster**, **Carl Kesselman** and **Steve Tuecke**, the so called *fathers of the Grid*. They led the effort to create the Globus Toolkit [76] incorporating not just CPU management (e.g. cluster management and cycle scavenging) but also storage management, security provisioning, data movement, monitoring and a toolkit for developing additional services based on the same infrastructure including agreement negotiation, notification mechanisms, trigger services and information aggregation. In short, the term Grid has much further reaching implications than the general public believes. While Globus Toolkit remains the *de facto* standard for building Grid solutions, a number of other tools have been built that provide a subset of services needed to create an enterprise Grid.

Grid computing offers a model for solving massive computational problems by making use of the unused resources (CPU cycles and/or disk storage) of large numbers of disparate computers, often desktop computers, treated as a virtual cluster embedded in a distributed telecommunications infrastructure. Grid computing's focus on the ability to support computation across administrative domains sets it apart from traditional computer clusters or traditional distributed computing.

This approach means that secure authorization techniques must be used to allow remote users to control computing resources. Grid computing involves sharing heterogeneous resources (based on different platforms, hardware/software architectures, and computer languages), located in different places belonging to different administrative domains over a network using open standards. In short, it involves virtualizing computing resources.

Focusing on related work in Grid Computing and Distributed Object Technology, we can find that existing RPC-based solutions are typically built using the GridRPC API [109]. This API was designed to address one of the factors that hindered widespread acceptance of Grid computing – the lack of a standardized, portable and simple programming interface. Since the GridRPC interface does not dictate the implementation details of the servers that execute the procedure call, there are several implementations of the GridRPC API, each of which has the ability to communicate with one or more Grid computing systems.

Two well-known implementations of GridRPC are: one which lies on top of NetSolve [62], and Ninf-G [62], which is a a full reimplementation of Ninf on top of the Globus Toolkit [76].

NetSolve has three main entities: the *client*, the *server* and the *agent*. Clients and servers model the typical behaviour in RPC systems, whereas agents maintain a list of all available servers, perform resource selection for all client requests and ensure load balancing of the servers. In its latest versions, NetSolve also includes support for basic Kerberos authentication. Further, fault detection and recovery is managed in such a way that it is transparent to the user. The agent keeps track of the status of all available servers so if there is a problem the agent can choose a new server to handle the problem.

Ninf-G is a GridRPC system built on top of the Globus Toolkit. Globus provides Grid's lower-level features such as authentication, authorization, secure communication, directory services, and many others. Nevertheless, the Globus Toolkit alone is insufficient for programming the Grid at higher-level layers. Ninf-G is a full reimplementation of Ninf. This system provides a mechanism to *gridify* an application or a library by using the Ninf IDL language to generate the necessary stubs. The executable is registered into Globus' *Monitoring and Discovery System (MDS)*, and Globus-I/O is used for communication between client and server.

Even though Grid toolkits, and more specifically the Globus Toolkit, apparently seem to comply with many of the requirements we have proposed, this is far from the case. It is true that the Grid provides scalability, high availability, fault tolerance, and even load balancing. Nevertheless, it is more focused on solving large computational problems or dealing with huge datasets. Therefore, Grids are normally composed of very powerful servers which deal with these kinds of calculations. It is true that Grids define a very strict security framework, so that operations between nodes can be permitted or denied. However, this limits its dynamicity, since each node willing to enter a Grid will need to obtain a set of public/private keys. Moreover, Grids do not easily support the use of computational resources on the edges of the Internet for the reason stated above.

As a consequence, we believe that Grids' functionalities are not enough to solve the problem stated in this thesis, although a relatively new variation of Grids might be able to: **p2p Grids**.

In an attempt to merge the best of both p2p and the Grid worlds, [62] defines the concept of a p2p Grid. A p2p Grid contains a set of services that includes the services of Grids and p2p networks and supports natural environments with features of both limiting cases. In p2p Grid architecture, Web services play a very important role. In addition, there is an event service which links these Web services and other resources. In p2p Grids, everything is a resource, and they are exposed directly to users and to other services. An important feature is that these entities are built as distributed objects which are constructed as services whose properties and methods can be accessed by a message-based protocol.

When we studied this world, no P2P grids were based on a structured p2p substrate. Therefore, in [98] we adopted the term **structured peer-to-peer grid**, defined as a p2p Grid whose network routing substrate is based upon a structured key-based routing overlay infrastructure. The main advantage of this approach is that these kinds of Grids can benefit from better message routing and its inherent locality properties.

Therefore, the structured p2p Grid concept seems to be a good starting point, even though it is not a middleware, but an architectural premise on which middleware services can be built.

2.4.8 Wide-Area Component-Based Architectures

A number of component-oriented architectures have been developed over the years. CORBA Component Model [35], DCOM [32] (or the .NET Framework [54]), and Enterprise Java Beans (EJBs) [46] are perhaps the most popular of the traditional client-server based component models. There are also p2p and Grid component architectures, which include, for example, Fractal/ProActive [60] or P2PComp [75].

CORBA Component Model (CCM), EJBs, and .NET are all based on client-server environments. They provide a rich set of services such as transactions, security, persistence, naming or events. Their container models usually require powerful server machines to run on. CCM, EJBs, and .NET are aimed at client-server scale applications, whereas our framework is for wide-area applications. All these solutions can support fault tolerance, load balancing and high availability by means of expensive cluster server topologies, and hardware load balancers.

In the Grid world there are approaches like Fractal/ProActive [60], which is defined as a hierarchical and dynamic component model. Nevertheless, its approach is different from ours, in the sense that virtual node and virtual machine mapping is performed on the component's deployment descriptor, thus not allowing self adaptation to node failures. As a consequence, it does not support fault tolerance nor dynamicity. Moreover, it cannot benefit from any group communication network proximity services, since it has no support for them at present.

Finally, P2PComp [75] is built on top of an unstructured p2p network, and its main aim is to address the development needs of mobile p2p applications. It features a lightweight container model, and provides many services, including synchronous / asynchronous remote invocations, hot swapping, service fetching and ranking. All these services are basically designed for highly mobile and dynamic applications. Fault tolerance and high availability can be emulated with the hot swapping service, which automatically tries to re-locate a service which for some reason becomes unavailable. However, there is no scalability proof of P2PComp (since evaluations only involve up to five devices), and it is not dynamically adaptive to high request peaks, so it does not support load balancing.

2.5 Conclusions

In this Chapter we have defined the requirements for our wide-area middleware proposal. We have presented the big picture of our generic proposed model, and have outlined that it is based on three main core layers, the result of the specified requirements: a **wide-area routing substrate layer**, a **wide-area application-level multicast layer**, and a **wide-area persistence service layer**. These tiers are the main building blocks which support our whole proposal.

We have analyzed the state of the art of each of these layers, and discussed their main features. We have described the existing decentralized routing layers, focusing on p2p architectures, and their evolution. We have also described the background to wide-area event systems, and proposed a reference model for comparison. Finally, we have analyzed the existing wide-area persistence systems, which are suitable for data storage and retrieval facilities.

Having described all the background, we analyzed the related work on existing wide-area middleware systems. We compared them and related them to each of the layers and to our requirements. Table 2.11 summarizes the requirements and the related work. It can be seen that none of these middleware approaches fully complies with the requirements we have described, so we conclude that none of them can provide what we need.

	Globe	Legion	JxtaJeri	P2PSockets
Scalability	✓	✓	✓	✓
Fault Tolerance	✓	✓	✗	✓
Load Balancing	✓	✗	✗	✗
Dynamicity	✗	✗	✓	✓
Use of edges of Internet	✗	✗	✓	✓
High Availability	✓	Partly	✗	✓
Usability and Programming Abstractions	✓ Distributed objects Object location service	✗ Distributed objects Object location service	✗ Remote Method Invocation	✗ Too low level

	Common API	Grid	CORBA/ .NET/EJB	P2PComp	Our Model
Scalability	✓	✓	✗	✗	✓
Fault Tolerance	✗	✓	✓	✓	✓
Load Balancing	✗	✓	✓	✗	✓
Dynamicity	✓	✗	✗	✓	✓
Use of edges of Internet	✓	✗	✗	✓	✓
High Availability	✗	✓	✓	✓	✓
Usability and Programming Abstractions	✗ Location service Group communication Too low level, however	✓ Distributed objects and components Too complex	✓ Distributed components Location service Group communication	✗ Distributed components Location service Group communication	✓ Distributed objects and components Location service Group comm.

Table 2.11. Comparison of the different Wide-Area Middleware approaches
Some of them may not be particularly targeted to wide-area environments. Fractal/ProActive has not been included, because it is based on the Grid and can therefore be considered part of the Grid column.

Therefore, no wide-area middleware approach is scalable, provides transparent fault tolerance and load balancing services, is adaptable to constant node joins and leaves, and uses the resources provided by the computers at the edges of the Internet. Moreover some approaches are also of limited usability and do not provide high availability guarantees.

Consequently, there is a need for a wide-area middleware platform which fulfills all the requirements described in this chapter. If they are all to be fulfilled, we need the three layers already mentioned, which can also introduce such new functionalities as proximity calls using *anycast* primitives provided by the event service, decentralized object location and distributed interception among others. We also have the chance to use the potential of structured peer-to-peer overlay networks as our underlying network substrate, and benefit from their inherent properties which include fault tolerance, efficient message routing, self-healing, self organization, and many others. As a consequence, and although our proposed model is thought to be generic regarding the specific implementations of the three defined layers, we shall implement our wide-area middleware on top of a structured p2p overlay network. This is, to the best of our knowledge, the first complete middleware platform that has been built on top of this kind of network.

By using the three main core pieces together, we build our wide-area middleware framework which provides a set of common services to the distributed application developer. The proposed generic model is designed and constructed using two complementary middleware approaches: remote objects and distributed reusable components. The remote object layer provides the foundations and the most important innovative services to the component layer. This component layer enables the lightweight components to be defined and deployed. The components can later be reused to provide a higher level of abstraction for the wide-area distributed applications.

The applications developed by our middleware will also satisfy the requirements of the wide-area applications:

- **The applications developed by our middleware encourage good use of the resources available on the edges of the Internet and can share resources**. Data is shared throughout the nodes that conform the network thanks to the routing substrate, and the persistence layers of our middleware, which store and retrieve data in / from nodes.

- **Applications can collaborate among groups**, using the group communication primitives of the wide-area application-level multicast infrastructure of our middleware.

- **Applications are fault tolerant, and resources are highly available**, by using transparent replication of data in the persistence layer of our middleware.

- **New wide-area applications** can be easily developed. Wide-area applications can be developed using p2p lightweight components. These components can also be easily deployed.

- **Applications can benefit from location awareness** if the application-level multicast layer of our middleware is proximity-aware.

- **Connectivity can be maximized**. This requirement strongly depends on the routing substrate used. NAT and firewall traversal can be successfully dealt with these days. Even though many of the structured p2p overlay network protocols do not explicitly deal with these problems, it is expected that in the near future, these technicalities will be resolved.

- **The security requirement**, which is beyond the scope of this thesis, is not explicitly solved by our middleware. However, we have tried to tighten the security policies in our SNAP deployment framework, described in chapter 4, by only allowing administrators to deploy applications onto the network by using public/private keys and signatures.

The next chapter introduces the proposed middleware framework and its inovative services, and describes how it fits in with the different layers so that all of its functionalities are provided to the upper layers. It is important to note that what is described is a generic proposal for a wide-area middleware framework, which means that it is not necessarily tied to any specific underlying technology. It is clear that when we implemented our prototype we had to make certain decisions. However, these were only design decisions, since the generic nature of our approach means that other layer components could have been chosen.

Chapter Three

3 Wide-Area Middleware Proposal

3.1 Introduction

This chapter describes our whole Wide-Area Middleware proposal. As we have stated above, our aim is to provide two complementary middleware layers, consisting of a wide-area remote object layer, and the naturally resulting wide-area distributed reusable component layer. Both these layers are built on top of the three main pillars described in the previous chapter. Therefore, access to lower-level layers is abstracted and made transparent to the application developer.

As seen in the architectural diagram (Figure 3.1), both layers sit on top of the same common building blocks, and the order in which they are placed determines the programming complexity. Therefore, the higher the layer is, the greater the abstraction of the application programmer.

These layers and their associated services are the most important contributions of this thesis. All these contributions will be discussed in greater detail throughout this chapter. However, we now enumerate them briefly below:

- Definition of a **Wide-Area Event-Based Remote Object Middleware**, which provides the typical remote object services, and the following:
  - Definition of a new set of remote object **invocation abstractions**. Our remote-object middleware provides the traditionally existing object-to-object (one-to-one) remote method invocations. Moreover, it provides object-to-objects (one-to-many) calls by using a wide area application-level multicast communications bus. If this underlying information bus also provides us with network proximity-aware primitives like *anycast*, we can also provide the *anycall* and *manycall* abstractions. Such remote

method invocation techniques allow a method to be invoked on one of the *nearest* objects which comply with a parameterized condition.

Moreover, *hopped* calls allow for fault tolerance when methods are invoked on dead objects: if another live replica of the object exists, it responds to the call. It is important to recall that such invocation abstractions aim to be generic in the sense that they are not closely coupled to a specific underlying information bus or routing substrate. Therefore, the underlying layers can be switched by others with the same functionalities, although the same interfaces must be respected.

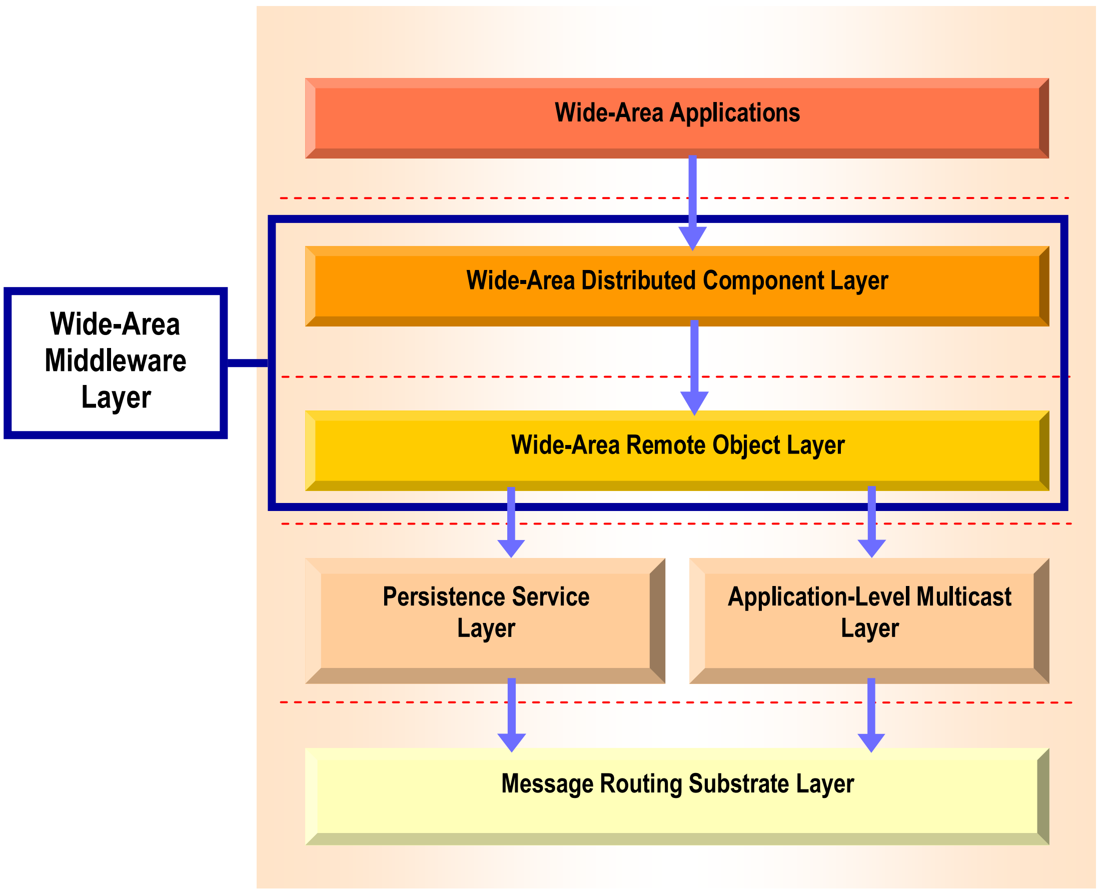

Wide-Area Applications
Wide-Area Middleware Layer
Wide-Area Distributed Component Layer
Wide-Area Remote Object Layer
Persistence Service Layer
Application-Level Multicast Layer
Message Routing Substrate Layer

Figure 3.1. Our proposed Wide-Area Middleware Architecture
Wide-Area distributed applications are built on top of our middleware using the component layer.

- Definition of a **decentralized object location service**. This service allows remote objects / components / applications to be located and inserted into our decentralized generic model. It is similar to any *naming* service, but proves a fault tolerant and scalable level of indirection. Any object data can be stored and located later by using simple *bind()* and *lookup()* primitive operations. One of the major advantages of this service is that it is inherent to the key-based routing substrate we use. As a consequence, it is generic enough to allow underlying layer switching by another different substrate, although it is required to follow the same contract.

This service is closely related to the decentralized persistent layer of our model. Any data to be stored in a persistent way can be recorded in a decentralized way, as happens with object or component handles. In order to support fault tolerance, data is replicated among a specified number of nodes. A set of algorithms is used to take care of bottlenecks and node overwhelming.

- **Distributed interception service**. By means of the underlying information bus, we provide primitives that can easily intercept remote object calls, in a similar way to Aspect Oriented Programming (AOP) techniques. Therefore, invocations to remote objects can be captured, analyzed, transformed, and even discarded. This service provides runtime interception with no need to change either the source or the target object code. Type-compatible interceptors are therefore added or removed in runtime by calling our model's interception service. This approach, for example, is used for monitoring and for providing load balancing to our objects or components.

- **Wide-area load balancing** through interceptors or the *anycall* abstraction. Two contributions that provide load balancing in wide-area distributed objects and components are described. Both these alternatives gracefully fit into our proposed generic middleware framework for global distributed application development. Basically, these two load balancing techniques target different domain areas. For those scenarios in which each object is aware of its own load, the *anycall*-based scheme selects the target object by letting each target node decide. This approach is rather stateless, and provides proximity aware support. The alternative scheme lets an interceptor know the state of each of the objects to be load balanced. Requests are directed to the interceptor which forwards the invocations to the less loaded object server (and defining which *load* policy is to be taken into account).

  Both schemes are complementary and target different use cases, providing the load balancing requirement with enough genericity and flexibility.

- Definition of a **Wide-Area Distributed Reusable Component Middleware**, which provides the most common services of these solutions, plus:

  - **A decentralized lightweight container model**. There is a transition from remote objects to distributed components, in such a way that application developers are provided with a higher level of abstraction. Therefore, we define a component-based reusable layer, which presents an alternative way of *holding* any component's life cycle routines. Distributed components are modelled as remote objects, including a life cycle service, and a decentralized deployment and location service. Instead of having a monolithic heavyweight container housing all components, we opt for letting each node that participates in the worldwide network be a lightweight container. Therefore, we allow components to be distributed throughout the network, and benefit from the underlying services provided by the remote object layer.

All these innovative contributions allow for an efficient, scalable, fault-tolerant, and decentralized middleware suitable for distributed wide-area application development.

This chapter is structured as follows: first we describe the main innovative contributions of our **wide-area remote object middleware**, and then we go on to describe our **wide-area reusable component infrastructure** built on top. Each of these sections includes an overview of their respective prototype implementations, called **Dermi** and **p2pCM** respectively, as well as extensive simulations and empirical evaluations.

3.2 The Wide-Area Remote Object Middleware Layer

The first element of our middleware proposal provides the main foundations for the development of wide-area distributed applications. Even though applications can easily be developed on top of these objects, the idea was to provide a higher level abstraction layer (the component layer) that used this remote object layer in order to make the development of these applications even more straightforward. Therefore, we used a well-known programming paradigm: distributed reusable components.

In this section we are going to describe the **object middleware layer**, and its innovative services. The idea behind a remote object middleware is to create distributed objects which can efficiently communicate through the network. These objects are expected to be created, reused, invoked, destroyed, etc. by other objects. Communication between objects is to be performed using the routing substrate or the event service (if *one-to-many* or *anycast* communication is involved).

Exceptions have to be dealt with in a distributed manner. Therefore, if a remote object is invoked and an exception occurs, this should be propagated back to the caller object so that it can be handled.

Remote object middleware solutions provide many services which are inherent to distributed object technology. These include a naming service, synchronous and asynchronous invocations, object replication, inheritance, pass-by value, pass-by-reference, and many others.

We wanted our wide-area remote object middleware to provide all these services to the upper-level layers, and also other innovative services which could benefit from the decentralized nature of the underlying substrate, and provide interesting features targeted to the wide-area domain.

The rest of this section is structured as follows: first we describe the innovative services offered by the remote object middleware layer. Later we go on to describe the rest of the common services provided by this layer, and the adaptations so that it can work in a decentralized environment. Finally, we present our remote object middleware's prototype implementation **Dermi**, and we validate our approach with an extensive set of simulations and empirical evaluations.

3.2.1 Innovative Services

Using the decoupled nature of the underlying event infrastructure, we created several innovative services that our remote object middleware provides to the application layer. Our approach includes object mobility, replication, caching and discovery services. However in this section we concentrate on the most innovative ones: **p2p call abstractions**, **decentralized object location**, and **distributed interception**.

3.2.1.1 p2p Call Abstractions

Figure 3.2 shows all of our remote object middleware's call abstractions. We divided them into two groups: *one-to-one* and *one-to-many*.

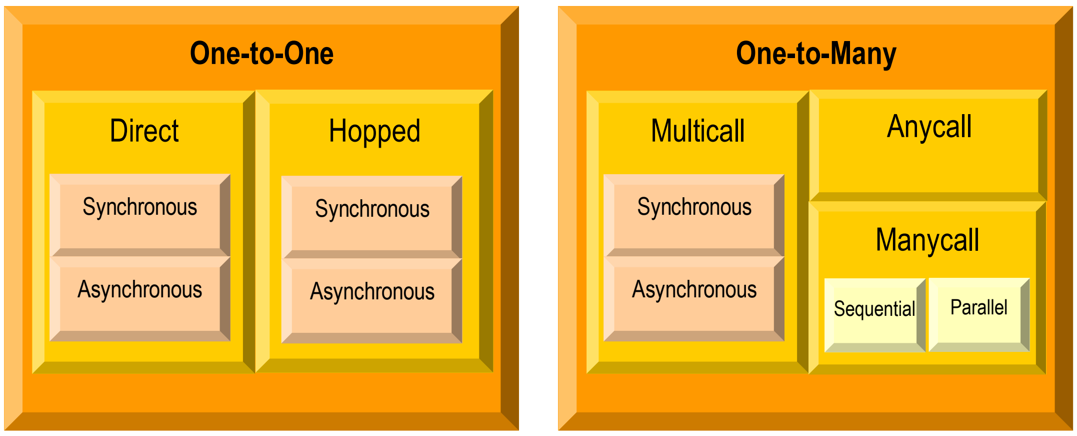


Figure 3.2. Our Remote Object Middleware's p2p call abstractions
(a) One-to-one calls involve only two entities: server and client. (b) One-to-many calls involve many entities: multiple servers and one client, or vice versa.

One-to-one calls. One-to-one calls can be synchronous or asynchronous, depending on whether a client wishes to block their execution until a result returns. One-to-one calls do not use the event service, which fits more effectively into one-to-many calls. In one-to-one direct calls, an object client (stub) sends a message directly to an object server (skeleton). To accomplish this, we have a direct mapping between the object and the physical network node in which it runs. Therefore, we use the server's **NodeHandle**, an object that represents the node's address and port number. Thus, we achieve a direct peer communication between both end objects. The results are returned the same way, producing a very efficient call that involves only two hops: one for the call and one for the returned results. In order to withstand all possible cases, any implementation of *one-to-one calls* must fully support direct synchronous calls, and asynchronous calls.

- **One-to-one *direct calls*** present several challenges because they are not tolerant to failures: when the server on which we wish to invoke methods goes down, it ceases to serve our requests. We solve this problem using **NodeIds** (an *alias* for the physical node in which the object runs) instead of **NodeHandles** (IP address), but this approach incurs additional overhead because a message routed to any given object might have to move through $O(\log n)$ (where n is the total number of nodes in the system) nodes before reaching its destination (this is applicable to any of the *structured p2p overlay routing protocols* described in the previous chapter. If another substrate is used, this number can be higher or lower). This philosophy is in opposition to that of direct calls, in which a message moves directly from source to destination.

- Using the routing substrate's key-based routing capabilities is the foundation for what we call **one-to-one *hopped* calls**. The advantage of using the **NodeId** to route messages to the server is that we can use any existing replication mechanism, thus providing some failure tolerance. When the server we are using goes down, the message is automatically routed to another server from the replica group, in a process transparent to the client, which continues to use the same **NodeId** to route messages. Hopped calls are not as efficient as direct calls, but they provide some fault tolerance.

One-to-many calls. Such calls are modelled using the event service layer by means of disseminating notifications. We only use the application-level multicast layer in these calls.

- The **multicall** abstraction is a remote invocation from one client to many servers or from one server to many clients (for example, to propagate state information). Multicalls can be synchronous or asynchronous and are modeled as one-to-many notifications. All clients subscribe to the same topic identifier (*objectUID* + *MethodID*) and the object server publishes events matching that subscription. As client numbers increase, this approach scales better than having point-to-point connections to any interested client. The approach also achieves transparency from clients to services — clients don't need to know the locations of all servers that provide a service. When we designed our system, we wanted to stay close to the chosen programming language. Thus, runtime stubs and skeletons are embedded. The stub code creates the appropriate subscription, decoupling the object server from clients.

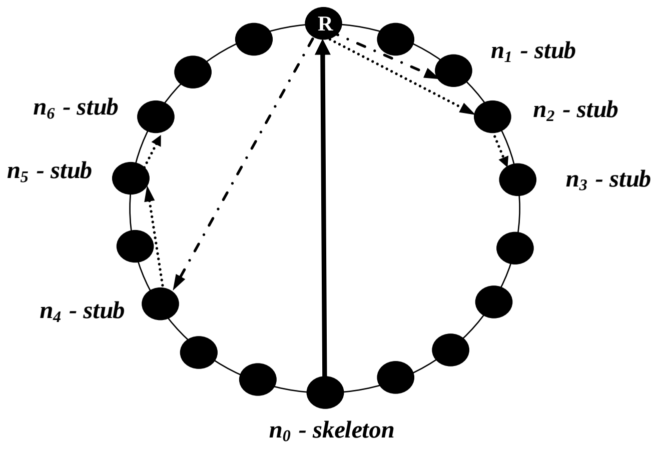


Figure 3.3. Multicall abstraction
Object at node n_0 sends a multicall headed towards the subscribers group root (labelled R). The call event is efficiently disseminated by the application-level multicast layer to all subscriber clients.

- **Anycall** is a new form of remote procedure call that benefits from network locality. If the application-level multicast layer provides us with an efficient *anycast* primitive, we use it to create a call to the objects that belong to the same multicast group (object replicas that can provide us with a service, for example). The anycall client is insensitive to which group object provides data; it only wants its request to be served. The idea is to iterate the multicast tree, starting from the closest member in the network. Once a member of the tree is found to satisfy the condition, it returns an affirmative result. If no group members are found to satisfy the anycall condition, a remote exception is returned to the caller.

  To illustrate the behavior of the anycall abstraction, consider how we might implement a CPU intensive application like SETI@home [41] or the United Devices Cancer Research Project [23] using our remote object middleware. These applications retrieve data units from servers, analyze them on home or office PCs, and return the results to the servers. Our anycall abstraction can provide a simple alternative to the data-unit retrieval process. Imagine, for instance, that we have several servers with available data units (see Figure 3.4). We can create a multicast group under the topic *AVAIL_DATA_UNITS*, which includes an identifier equal to *hash ("AVAIL_DATA_UNITS")*. When a client node wants to get a data unit, it executes `DataUnit du = anycall ("AVAIL_DATA_UNITS", getDataUnit)` to trigger an anycast message to the group; in response, the nearest group member checks whether it has any data units available. If it has, the group member returns the data unit to the client and the anycast message routes no further. If it has not, the anycast message is routed to another group member and so on, until a data unit is found or the message reaches the root, which means that none of the group members have available data units. This result throws an exception back to the client to provide proper notification.

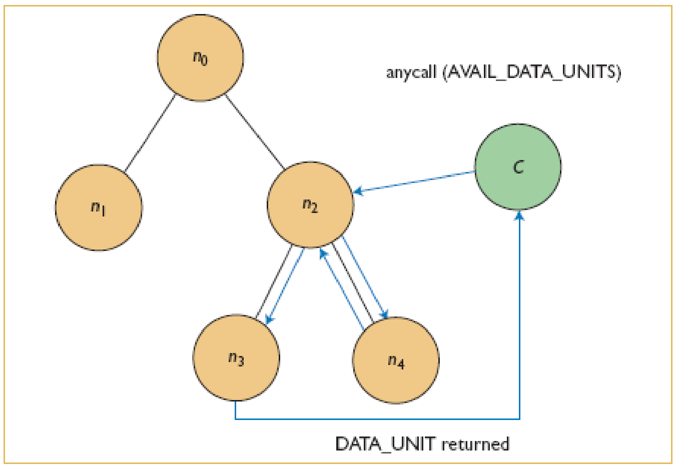


Figure 3.4. Anycall example
Client C anycalls to the AVAIL_DATA_UNITS group, reaching n_2 first, which has no available data units to serve. The multicast tree is iterated ($n_4 \rightarrow n_3$) until n_3

- A **manycall** is a variation of the anycall abstraction. It takes advantage of the *manycast* primitive of the application-level multicast layer and it therefore sends a manycast message to several group members, continuing to route until it finds enough members to satisfy a global condition. Like anycall, when an object (in the multicast tree) receives a manycall message, it first checks whether the object satisfies a local condition and, subsequently, checks whether a global condition (passed along with the message) is met. The manycall is successful when the global condition is met. To better understand the manycall abstraction, imagine a massive online voting scenario in which we need a minimum of *x* votes to do a certain job. We can send a manycall to the group so that each member can vote yes or no, according to its local condition (to approve the execution of a certain simulation, for example). After checking this local condition (voting yes or no), the object checks the global condition (have *x* votes been reached?). If they have, the voting process concludes successfully, communicating the result to the manycall initiator. If the global condition has not been reached, (the minimum number of votes *x* is not reached after iterating throughout the multicast server tree), the unfavorable result is passed to the client.

From the usability point of view, developers can easily label any remote object's methods with its type (*multicall*, *anycall*, *manycall*, ...), by using annotations on the remote object's interface definition. An example is shown in Figure 3.5. More information can be found in Annex A.

```
/**
 * This is the interface for the Seti anycall demo object
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DermiRemoteInterface
public interface Seti extends ERemote {

  // Method signature for the anycall method pair must be the same,
  // except for the return result, which needs to be boolean for the
  // condition method

  /**
   * This method returns data unit from one of the client's nearest server
   * @param system String Example parameter
   * @throws RemoteException If something goes wrong ;-)
   * @return String Data unit returned
   */
  @RemoteMethod (granularity = Granularity.ANYCALL)
  public String getDataUnit (String system) throws RemoteException;

  /**
   * This method is automatically called by the skeleton to check whether
   * the condition can be satisfied for each server
   * @param system String Example parameter
   * @throws RemoteException If something goes wrong ;-)
   * @return boolean true if condition satisfied (the server has
   *  available data units)
   */
  @RemoteMethod (granularity = Granularity.ANYCALL_CONDITION)
  public boolean getDataUnitCondition (String system)
                                                   throws RemoteException;
}
```

Figure 3.5. Definition of anycall methods in a remote object's interface

3.2.1.2 Decentralized Object Location Service

A scalable, stable, and fault-tolerant decentralized object-location service is needed to locate object references in wide-area environments. A centralized naming service could be a bottleneck for such a common task. We used the underlying persistence layer facilities to build our object-location service. For our prototype implementation (**Dermi**), we used a structured DHT, even though other algorithms could be used. However, other unstructured p2p networks, such as those based on Gnutella-like protocols, use flooding techniques, which do not guarantee deterministic resource location. By using a DHT-based approach to build our object-location service, we guarantee that a resource stored on the network will be found in at most $O(\log n)$ network hops — a stark contrast with the probabilism of unstructured p2p overlays.

Our p2p location service stores object-location information that can be used to find objects via human-readable names. As in other wide-area location services, our object names do not contain any embedded object’s location information to decouple their current location from their name. That is, an object’s name is independent of its location. We adopted a uniform resource identifier (URI)-style naming convention for objects (for example, *p2p://cat/urv/etse/deim/Simple*). Although we permit URI hierarchies that uniquely represent our objects, we use a secure hash algorithm (*SHA-1*) to hash this key and insert it into the DHT.

The process of inserting and looking up an object handle is shown in Figure 3.6:

- A specified node n_0 instantiates an object locally and inserts its handle into the decentralized object registry. To do so, it hashes the object's identifier name, and obtains an *id* which automatically maps to the node whose identifier is closest to the *id*. In the figure, this is node n_1.

- Once the object's handle has been inserted, we suppose that somebody else may wish to look up this object and invoke methods on it. Since we do not know its exact location, but we know its identifier name, we query the location service on n_2, and obtain the same *id* as before by hashing the object's name. Therefore, our request is redirected to n_1, which returns a handle to the object that is running on n_0.

- At this point, we can call the object's methods.

Naturally, if the node containing the object's location information fails, object lookups will fail as well, as the node that contains this information is missing. To avoid this problem, data replication mechanisms are used at the DHT tier. When an object handle is to be inserted, this is replicated among the *k* nearest nodes to the target node. This way, should the target node fail, information is not lost and the object's handle can be recovered from any of the *k* nearest nodes.

Another more efficient approach for wide-area scalable object location is to use a hierarchical system such as the Globe Location Service [114]. Globe objects are located by means of a dynamic adapting worldwide search tree. This approach is normally more efficient than the one presented here, because in most cases, any object can be located with only 2 network hops. However, if we used this system, our middleware would depend on an external hierarchical service for object location, which could be cumbersome if for some reason it becomes unavailable. Moreover, by using our system, we make better use of the Internet's infrastructure by using the resources at its edges, as we require.

Nevertheless, it is interesting to note that both systems can be used for our middleware. Globe's hierarchical search tree approach could be used for the sake of efficiency. However, if we prefer not to depend on an external location service, and make good use of the resources available at the edges of the Internet, our decentralized p2p object location system distributes object handles more sparsely and does not involve any hierarchies.

Our decentralized location service handles duplicates as well, throwing an exception if someone wishes to rebind an already bound object without unbinding it beforehand.

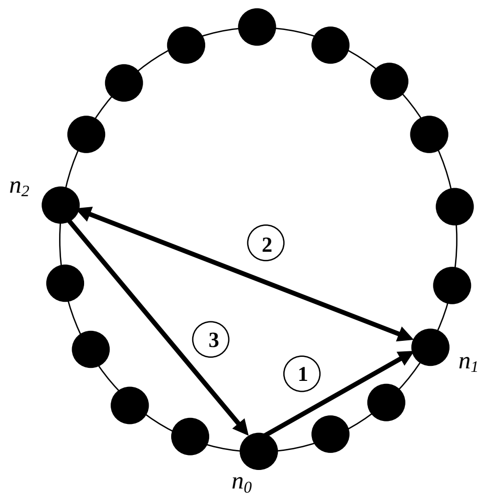


Figure 3.6. Example of object handle insertion and location
(1) Node n_0 creates an object locally and inserts its handle in n_1, which is obtained by hashing the object's name (*/simple*). (2) Node n_2 wishes to get the object's handle, hashes its name (*/simple*), and queries n_1 for the object's lookup data, which is returned. (3) Now n_2 can call the object's methods.

From the usability point of view, object insertion and looking up follows the same methodology found in traditional naming services (Figure 3.7).

```
...
// Load Dermi's connection properties
Properties env = Registry.getEnvironment ("dermi-config.xml");

// Create remote object (the first time)
SimpleImpl server = new SimpleImpl (env);
// We can use the object now
server.setAge ("29");

// Now that the object is created, we can bind in on the DOLR
Registry.bind ("p2p://simple_dermi_object", server);
...

// Look up an object in the registry
Simple client = (Simple) Registry.lookup ("p2p://simple_dermi_object");

// Execute remote object's methods
client.setAge ("30");
...
```

Figure 3.7. Binding and looking up an object from the decentralized object location service

3.2.1.3 Distributed Interception

Distributed interception enables concepts from connection oriented programming to be applied in a distributed setting. We can use this service to reconnect and locate type-compatible interceptors at runtime in a distributed application. For our application-level multicast layer to support this feature, its classes are very likely to be extended. Thus, we do not need to change the interceptor skeleton or the intercepted remote-object subscriptions each time an interceptor is added or removed. Distributed interception can be a very useful mechanism in dynamic aspect-oriented programming (AOP) environments.

In our prototype implementation (see Figure 3.8), our interceptor implementation takes advantage of the fact that all events sent to a multicast group in the event service layer (Scribe) first route to the group's *rendezvous point*. Each group's rendezvous point contains a list of pointers to other interceptor objects, which update every time an interceptor is added or removed. As a consequence, each time an event is sent to a multicast group, this notification arrives first at its rendezvous point, which checks whether it has interceptors. If there are no interceptors, the rendezvous node normally sends the event to the multicast group itself. Otherwise, the event passes sequentially throughout all the interceptors, which might transform it into a different event by changing its information. Finally, the event will be routed back to the rendezvous point, which will, in turn, send the intercepted event to the group members. We need a fault-tolerance mechanism in case the rendezvous point changes because network nodes are added or removed. Fortunately, the event service (Scribe) provides callbacks that notify us about root node modifications. The simplest approach would be to move all interceptor data from the old root to the new one, but this will not work if the root node fails. In this case, we must replicate all interceptor data among *k* nodes nearest the rendezvous point. To do this, we use our persistence layer.

Normally, there will be few interceptors because they are sequential and because efficiency quickly decreases as the number of interceptors increases. Nevertheless, to prevent this problem, and thus, reduce the number of network hops between interceptors, we could opt to move them directly to their source objects, which would allow events to be intercepted locally. This way, each time a publisher sends an event to its subscribers, the publisher itself will do the interception process locally, thus incrementing efficiency.

If this this interception mobility mechanism is used, distributed interception is also supported in *direct synchronous calls*. Notice that it is not possible to use interception mechanisms in *direct synchronous calls* because these calls do not use the application-level multicast service. Therefore, there is no rendez-vous point where interceptor pointers can be stored.

Distributed interception is difficult to implement in strongly coupled object systems, in which clients and servers must be notified of object changes. When a TCP connection is established among many clients and an object server, the insertion of a remote interceptor means that all clients should reconnect to the new interceptor and bind it to the remote server. Our solution does not affect client connections, which are represented as invariant subscriptions.

It is clear that one of the most important drawbacks of the distributed interception feature is its need to be natively implemented at the application-level multicast layer, since some upcalls need to be captured and explicitly handled. However, we plan to improve this feature by means of AOP-like features. The idea is that each node runs an AOP-aware virtual machine, in which we can deploy runtime aspects which would act as pointcuts and perform the distributed interception. This issue is currently being investigated.

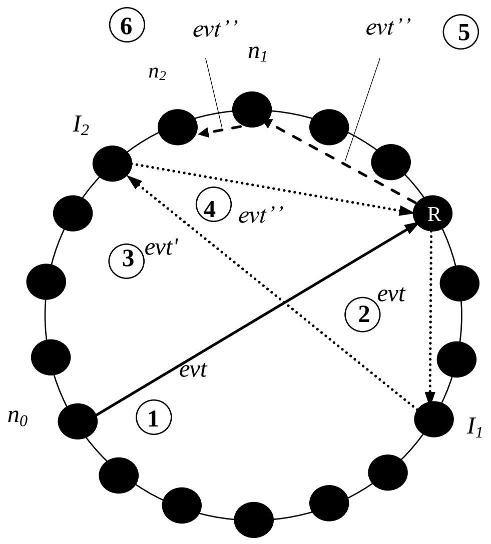


Figure 3.8. Distributed interception algorithm
The object at n_0 sends an event to the group rooted at R. This event is sent to the interceptor queue sequentially, thus transforming it (evt → evt' → evt''). Finally the event is sent back to the root, which ends up delivering it to group subscribers.

From the developer's perspective, distributed interception objects can be appended / removed in runtime by means of the naming service itself (see Figure 3.9).

```
...
// Load the registry first
Registry.loadRegistry ("dermi-config.xml");

// Look up the Simple object in the registry
Simple obj = (Simple) Registry.lookup ("p2p://simple_dermi_object");

// Instantiate and bind the interceptor object with the Simple object
LogInterceptorImpl server = (LogInterceptorImpl) Registry.loadInterceptor (
"dermi.samples.interception.LogInterceptorImpl", obj);

// The interceptor is loaded. All calls done to the Simple object will
// automatically traverse our interceptor now

...
```

Figure 3.9. Binding an interceptor to an already running object
The ***loadInterceptor()*** method allows dynamic interceptor binding.

3.2.1.4 Load Balancing Service

Load balancing can be defined as *distributing processing and communication activity evenly across a computer network so that no single device is overwhelmed*. Load balancing is especially important for networks in which it is difficult to predict the number of requests that will be issued to a server. For instance, busy web sites typically employ two or more web servers in a load balancing scheme. If one server starts to get swamped, requests are forwarded to another server with more capacity.

In distributed objects technology, the problem is the same: a resource which provides a service can become overwhelmed by many clients. In this case, the resource is clearly a centralized hot spot, and becomes a single point of failure. To avoid such hot spots, the object server is replicated Nevertheless, this is not the solution if the replica to be chosen depends on the current load. Therefore, as well as having multiple replicas of the same service, we need a mechanism to choose one or another according to certain parameters which may include server load or network bandwidth, to name the most common ones.

As an example, we can imagine a service like a video broadcast. If we expect our video server to be joined by a considerable number of clients, we should consider replicating it. However, this does not solve the problem. We require a mechanism to decide which replica is to be assigned to each client that connects to our video broadcast service. This mechanism should take certain criteria into account (in this case, network bandwidth, or server's node stress). Naturally, this assignment process must be made transparent to the client.

Having described the problem, and bearing in mind the services provided by our remote object middleware, we present two different load balancing algorithms which can be used in wide-area environments. Before describing them both, we outline the background they share.

Firstly our object server must be replicated among other nodes. Notice that our remote object middleware provides us with mechanisms to do this, which are transparent to the developer (this replication service is described in next section). This ensures that we have all the replicas required in one multicast group, and that any changes within the object server will be easily propagated to their replicas by sending event messages to the group (which will be received by all members). Once all the object server's replicas have been joined into one multicast group, we can opt for two load balancing options: ***decentralized load balancing management*** and ***interceptor-based load balancing management***.

3.2.1.4.1 Decentralized load balancing management

Decentralized **load balancing management** uses the *anycall* primitive for load balancing purposes. The working principle for this methodology is totally decentralized, meaning that there is no centralized authority that knows the load of each replica.

Instead, we will efficiently ask for the client's nearest replica to serve our request. If the replica feels it is not overwhelmed, it will serve us. If it feels that it is, it will answer that our request cannot be served. In the latter case, and by following the *anycall* behaviour, another object replica will be contacted and so on.

One scenario in which this load balancing scheme fits very well is in a distributed simulation of some kind. Imagine we have a group of servers waiting for data chunks to analyze. These chunks can be provided by using the anycall approach: the chunk is delivered to the **closest** server, which checks if it has already reached its maximum number of concurrent data chunks to be analyzed. If this is the case, it delegates processing to another server, and so on; otherwise it starts the chunk analysis. A diagram showing this process can be seen in Figure 3.10.

Notice this load balancing system is only valid for nodes that do not suffer network congestion. If nodes themselves become overloaded by network messages, they get saturated and will not be able to respond to our anycalls. This prevents our decentralized load balancing mechanism from working properly.

3.2.1.4.2 Interceptor-based load balancing management

To deal with possible network congested nodes, we have another way of balancing the load: **interceptor-based load balancing management**. In this case, we make use of the *distributed interception* mechanism. There is a special interceptor node, which will be aware of each server replica load. This idea is similar to the *agent* concept found in NetSolve [62]. Particularly, we add an interceptor object to the multicast group of the object server's replicas. This means that when any client wishes to call a server's method, the call will be transformed into an event that will be sent to the multicast group itself, and automatically intercepted by the distributed interceptor.

The interceptor will, in turn, be aware of the load of each replica and make a direct call to the replica whose load is sufficient to solve the client's request. All object replicas will periodically inform the interceptor about their possible congestion state, by sending special events to the group. These events will be captured and stored by the interceptor itself, thus making it aware of the overall replica status. Notice the interceptor will in fact know about each replica's capacity for serving requests, and will decide which will serve the current client's RPC. Naturally, this approach can make the interceptor itself a single point of failure: if the interceptor node fails, load balancing fails as well. We can use replication mechanisms for the interceptor itself, and assure correct load balancing even if it fails.

Clearly, if any of the replicas gets overloaded by network messages, the interceptor is aware so no more messages are routed to that node. Notice that in this case, load balancing continues to work properly, in contrast to the problem we have explained in the decentralized alternative. However, in this case, we have a single entity which controls the whole load balancing process. If we wish to distribute this task among all servers, we should use the first approach (decentralized).

One scenario in which this kind of load balancing scheme proves to be useful is the video broadcast example explained above. The broadcast of a video stream can overwhelm a server node's network adapter, which will therefore be unable to respond to anycalls (if we use the decentralized load balancing system). Nevertheless, if we use the interceptor approach, the interceptor itself knows the state of all available servers and tells the client the most appropriate one for connection. A scheme of this procedure is shown in Figure 3.10.

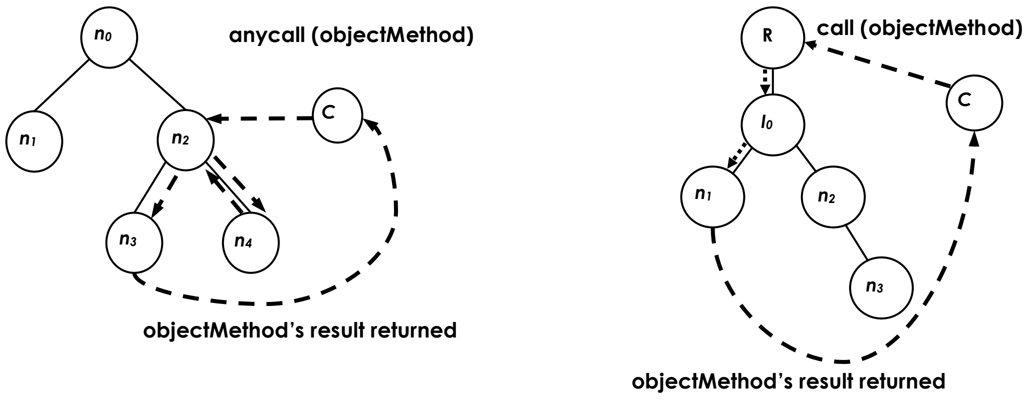


Figure 3.10. Example of a decentralized and interceptor-based load balancing scheme

3.2.2 Additional Services

Our wide-area remote object middleware needs to provide additional services which are already found in other remote object middleware solutions like RMI [47]. These include *dynamic class loading, remote exception handling, remote inheritance support, remote listeners support, pass by reference*, and others. All of them have been implemented in our proof-of-concept.

However, we also aim to provide other services which benefit from the underlying network substrate and its inherent services: application-level multicast, persistence, and routing layers. We briefly describe these services to show how they can be implemented on top of a decentralized infrastructure:

- **Object mobility** refers to the possibility of moving object servers to different locations and continuing to handle client requests. Object mobility can be easily accomplished in our remote object middleware by simply serializing the object's implementation to the remote endpoint. Before serialization, the skeleton removes all its subscriptions from the event service and, upon arrival at the remote endpoint, the skeleton reconnects to the event service and creates the subscriptions in the new location. Object clients (stubs) remain unaware of these changes since their current subscriptions are unaffected.

  In traditional object middleware, the strong coupling between clients and servers through TCP connections requires that all clients be notified so that they can reconnect to the new server location or use ad-hoc remote proxies instead. The first solution does not scale for a high number of clients and the second one is only an ad-hoc façade not suitable for unexpected scenarios. Our decoupled approach permits flexible object mobility and it can be used in different settings like server load balancing, spontaneous systems, agent systems and for highly dynamic and manageable remote services.

- **Object discovery** consists of using predefined Object UIDs to locate remote objects in an event bus. In this case, clients locate object servers with Ids associated to the object subscription. Object discovery is an extremely useful functionality in highly dynamic spontaneous scenarios such as wireless or mobile networks.

  Object discovery is usually solved in existing systems by means of the so called lookup services. The Jini [48] lookup service uses UDP broadcast to automatically discover services in the local environment. In fact, this is a nice solution that involves a one-to-many channel like UDP broadcast. Although it is a good solution in local area networks, it is not appropriate for remote endpoints, where UDP multicast or broadcast are not available. In these situations, using the decentralized event service layer would make the lookup service really accessible to remote locations.

- **Object replication** is an added functionality derived from the flexibility of the event bus. Objects are replicated by generating special stubs through the event system in order to maintain consistency and data among the remote object replicas. There is no central object server, so any of the clients can fail and we want the object's state to be preserved. Our approach is to have all replicas join a multicast group. When an object wishes to call a method from a replicated object, the stub sends the call to the group as a special *anycall*. This means that any of the object replicas (normally the nearest one) can respond to the call, which makes replication totally transparent to the user.

3.2.3 Proof-of-Concept Implementation: Dermi

As we have already mentioned, *structured p2p overlay networks* are an efficient, scalable, fault-resilient, and self organizing substrate for building distributed applications. This kind of network provides such interesting services as distributed hash tables (DHTs), decentralized object-location and routing facilities (DOLR), and scalable group multicast–anycast (CAST). These layers provide upper-level applications with an abstraction layer so that these services can be accessed transparently. For example, the DHT abstraction provides the same functionality as a hash table — associating key-value mapping with physical network nodes rather than hash buckets, like traditional hash tables. The standard **`put(key, value)`** and **`get(key)`** interface is the entry point for any application using the DHT.

All the functionalitites provided by these networks perfectly match the requirements of our entire model proposal: a routing layer, which corresponds to the key-based routing layer (KBR), which routes messages towards nodes according to a hash of the message key; an application-level multicast layer, which is responsible for providing wide-area event dissemination to multiple group members; and finally a persistence layer which is implemented as a DHT, thus storing key-value pairs.

Inspired by several applications that emerged as a result of these structured p2p substrates — including wide-area storage systems such as CFS [70] and PAST [73], event notification services such as Bayeux [118] and Scribe [66], and even collaborative spam filtering systems such as SpamWatch [44] — we developed the **Decentralized Event Remote Method Invocation (Dermi)** wide-area remote object middleware. Bearing in mind all the requirements and the already existing functionalities provided by structured p2p overlay networks, we decided it would be appropriate to implement our remote object middleware's prototype on top of this core substrate.

Dermi is built on top of a decentralized KBR p2p overlay network. It benefits from the underlying services provided by the p2p layer. Dermi uses the PAST or the Bunshin object-replication and caching system. Our system models method calls as events and subscriptions using the API provided by the CAST abstraction (which models a wide-area event service). The implementation of Dermi is currently available at our Web site [http://dermi.sourceforge.net].

Dermi uses Pastry as its structured routing substrate (KBR) and Scribe, a large-scale decentralized application-level multicast infrastructure built on top of Pastry, as its publish–subscribe message-oriented middleware (CAST). We chose Scribe because it provides a more efficient group-joining mechanism than other existing solutions, and it also includes multisource support. Additionally, our preferred KBR substrate choice was Pastry because its routing scheme is efficient, it takes account locality when routing messages, it is self-organizing and can gracefully adapt to node failures. All this makes Pastry one of the most interesting KBR implementations. Moreover the availability of a Java implementation at the time this thesis was begun, eased the adoption process. However, we could have used any other p2p KBR-based overlay network (or even none of them), provided that they share the same basic functionalities.

Dermi was strongly inspired by the Java RMI object middleware, which lets developers create distributed Java-to-Java applications in which remote Java object methods can be invoked from other Java virtual machines. It also uses object serialization to marshal and unmarshal parameters and does not truncate types, thus supporting true object-oriented polymorphism. Following this model, Dermi provides a **dermi.Remote** interface, a **dermi.RemoteException** class, and a **dermi.Naming** class to locate objects in our decentralized registry.

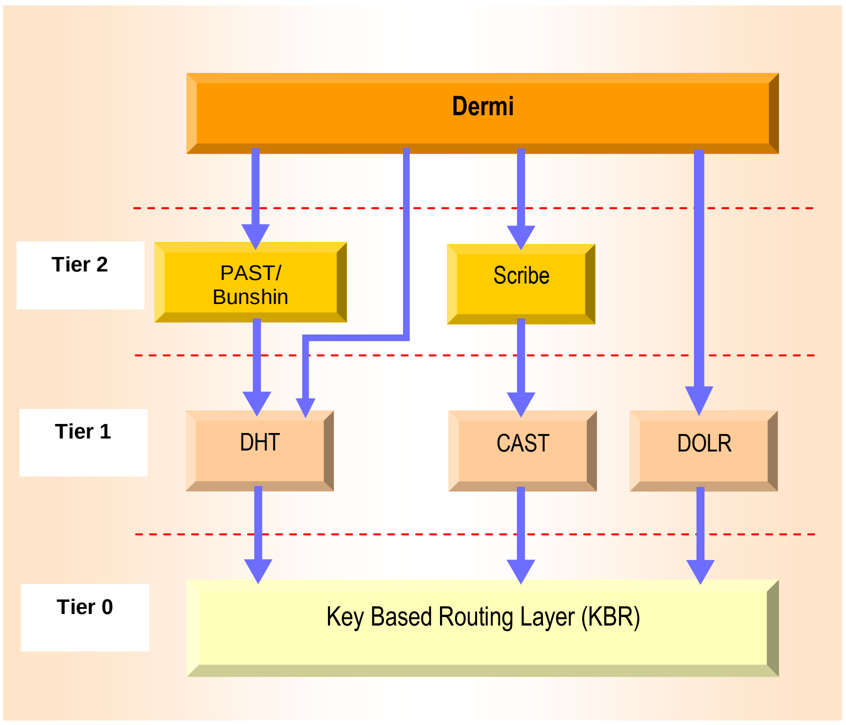


Figure 3.11. Dermi architecture diagram and how it fits within the Common API for Structured Overlays
Note that the KBR layer is the **Routing Layer**, the DHT and DOLR layers represent the **Persistence Layer**, and the CAST layer corresponds to the **Application-Level Multicast Layer**, as defined in our generic framework proposal requirements.

Early releases of our system included *dermic*, a tool that automatically generated stubs and skeletons for our remote objects. However, recent versions adopted Java 1.5's annotation mechanism and transparent dynamic proxies in order to avoid this process of stub and skeleton generation, since these are embedded in runtime. Together, these transparently manage object publications — subscriptions and their inherent notifications. Further, Dermi currently provides many other features found in Java RMI, such as remote exception handling, pass by value and by reference, and dynamic class loading.

Dermi also contains a communication layer between the stubs and skeletons — an important difference from RMI. In conventional RMI, a TCP socket is established between the caller (stub) and the callee (skeleton). Dermi stubs and skeletons use the underlying event service by making subscriptions and sending notifications to communicate the method calls and their results.

Figure 3.11 shows Dermi's architecture and how it maps onto the Common API [72] described before. As we can see, Dermi uses Scribe as its application-level multicast/anycast service, and PAST or Bunshin as its persistent DHT layer. Dermi can use the DHT directly as well (when no strict persistence is required). In addition, the DOLR layer is also used as its naming service. Note that the KBR layer is the **Routing Layer,** the DHT and DOLR layers represent the **Persistence Layer,** and finally the CAST layer corresponds to the **Application-Level Multicast Layer,** as they were defined as core requirements for our generic wide-area middleware proposal.

Notice however that this purely p2p approach also has drawbacks. One of them is how to perform the bootstrapping process: how can we find a contact node in the overlay to join? At this point, there are several approaches: for example, JxTA's Rendezvous point [39], Gnutella's Host-Cache [21] system or KaZaA's [42] indexing servers. One approach that can be applied to Dermi is the idea of a universal ring [65] expected to be joined by all participating nodes. Another drawback is that p2p overlays are not secure; even a small fraction of malicious nodes can prevent correct message delivery throughout the overlay. Techniques such as the ones described in [65] can be used to prevent some security attacks.

For further details on Dermi's implementation and API, see the annexes. We now go on to describe some issues about churn and failure handling in Dermi, and to explain the validation tests conducted on our prototype implementation.

3.2.3.1 Churn and Failure Handling

Nowadays, DHTs are a hot, ongoing research topic. One research area is *churn*, the continuous process of node arrival and departure. Researchers have demonstrated that existing DHT implementations can break down at the churn levels observed in deployed p2p systems [104], contrary to simulation-based results. Even though this is a hot research topic, we find the Bamboo approach very promising. As already described in Chapter 2, Bamboo is a new DHT that easily accommodates large membership changes in the structure as well as continuous churn in membership. Bamboo's authors say that it handles high levels of churn and that its lookup performance is similar to that of Pastry in simulated networks without churn.

Dermi partially addresses churn by controlling rendezvous-point changes. The node whose identifier is closest to the group's topic identifier is chosen to be the root (or rendezvous point) of a Scribe multicast group. When new nodes join or leave our DHT, another node identifier might then become closer to the group's topic identifier than its previous root. This means that every time a message is sent to the group, it will go to the new root rather than the old one. This can cause some events to be lost while a rendezvous-point change is in progress. Scribe notifies Dermi about these root changes via upcalls, and a buffering algorithm forwards lost events to the new root.

Dermi handles server failures in several ways. One is via the anycall abstraction. Consider, for example, an environment in which several servers offer the same service. When clients issue an anycall to this server group using Scribe's anycast primitive, each client should be directed toward its closest server replica. If any of these servers were to fail, however, the client would continue to be served, but by another server in the group. Thus, the only visible effect on the client side would be a slightly longer response time because it would no longer be served by its closest server. Another way to handle server failures is via replication mechanisms. With our decentralized location service, we must handle any possible node failures that can affect it. If a node that contains an object's location information fails, that object's lookups will fail as well. To solve this, we use data replication mechanisms, such as those provided transparently by the persistent layer we use: Bunshin [95] or PAST [73].

When an object handle is to be inserted, Dermi replicates its data among the *k* nearest nodes to the target node. When a target node fails, the object's handle can be recovered from any of its *k* nearest nodes.

3.2.3.2 Validation

We validated Dermi's approach using experimental measurements and simulations. We experimentally measured p2p call abstractions performance, as well as the decentralized load balancing scheme. We empirically tested the distributed interception mechanism as well as the load balancing scheme based on it.

3.2.3.2.1 Experimental measurements

We conducted several experiments to measure Dermi's viability using the PlanetLab testbed [69]. PlanetLab is a globally distributed platform for developing, deploying, and accessing planetary-scale network services. Any application deployed on it can experience real Internet behavior, including latency and bandwidth unpredictability. One of the things we measured was Dermi's call latency (how long it takes to perform a call). We conducted the experiments using 20 nodes from the PlanetLab network, located in a wide variety of geographical locations, including China, Italy, France, Spain, Russia, Denmark, the UK, and the US. We repeatedly ran the tests at different times of day to minimize the effect of momentary node congestion and failures. Before each test, we estimated the average latency between nodes to gauge how much overhead the middleware calls incurred.

3.2.3.2.1.1 Direct Synchronous Calls

The first test used one-to-one direct synchronous calls, which are achieved by establishing a direct p2p communication between two objects. Each test used 300 random invocations (*getter–setter* methods) for each pair of object nodes. As expected, this kind of invocation is the most efficient. The normalized incurred overhead is **1.27** (average call time/average latency), summarized in Table 3.1.

Object server node	*Object client node*	*Average latency*	*Average call time*
planetlab2.comet.columbia.edu	planetlab1.diku.dk	116	**149**
planetlab2.comet.columbia.edu	pl1.swri.org	60	**90**
pl2.6test.edu.cn	planetlab2.di.unito.it	522	**528**
planet1.berkeley.intel-research.net	planetlab5.lcs.mit.edu	84	**116**
planetlab1.atla.internet2.planet-lab.org	planetlab2.sttl.internet2.planet-lab.org	62	**73**

Table 3.1. Performance of one-to-one direct synchronous calls

3.2.3.2.1.2 Synchronous Multicall

Next, we tested one-to-many synchronous multicall using a group of 10 servers and a client invoking 300 setter methods on all of them. Table 3.2 shows the results. Because it is a synchronous test, the client remains blocked until all servers return from the invocation. Results show an average **463 ms** call invocation. In an attempt to make a better comparison of the first two test results, we conducted the same test, trying to synchronously invoke each server sequentially (the client calls each server in sequence). As we expected, performance degraded (**1,536 ms**), thus demonstrating multicall's ability to achieve one-to-many calls using the event service. On average (and in this case), multicalling is **3.32** times faster than sequential direct calls. This test demonstrated the viability of the multicall abstraction. Apart from being inefficient at using direct calls to simulate one-to-many calls, however, it is incorrect in terms of design: the test demonstrates only that it is faster to multicall rather than to make *n* one-to-one direct calls. The point is that it would also be conceptually incorrect for a client to know the *n* servers (which, in theory, provide the same service). In fact, the client will know the name of the service to invoke but not all the servers that provide that service (which can be removed, added, and so on). A client knows the name of the service to invoke, which is transparent on how many servers provide that service.

3.2.3.2.1.3 Anycall

To measure anycall performance, we used three nodes out of 20 that provided the same service. A set of clients began invoking anycalls on these servers. Each server provided clients with a standard data-unit set. When a server exhausted its data units, another server from the same group took its place, and so on. For anycalls, the results showed an average of **166 ms** for the first server, **302 ms** for the second, and **538 ms** for the third. These servers were chosen on the basis of proximity, so the server closest to the client was first, followed by the second-closest and then the last. The overall overhead for these calls was **1.46** (the normalized incurred overhead: average call time divided by average call latency). Results are summarized in Table 3.3.

Multicalling from planetlab1.diku.dk to the 10 servers		*Direct synchronous calling from planetlab1.diku.dk to…*	*Average call time*
463,49		planetlab2.comet.columbia.edu	**155.61**
		planetlab2.di.unito.it	**71.52**
		planetlab5.lcs.mit.edu	**248.05**
		planetlab6.lcs.mit.edu	**280.26**
		planetlab1.inria.fr	**94.60**
		planetlab2.nycm.internet2.planet-lab.org	**141.66**
		planet03.csc.ncsu.edu	**157.77**
		phys0bha-4a.chem.msu.ru	**76.30**
		pl1.swri.org	**202.53**
		planetlab2.ac.upc.es	**107.75**
Multicall aggregate time	**463,49**	**Direct call aggregate time**	**1536.05**

Table 3.2. Performance of one-to-many multicalls

3.2.3.2.1.4 Manycall

We tested the manycall sequential implementation under the same conditions as the anycall tests: clients sent manycalls to a group of three servers. In this case, we used the voting example described earlier. Each of the clients required an affirmative vote from each of the three servers. Once this task had been done, the manycall returned. On average, these invocations lasted **386 ms**, producing an overhead of **1.68**.

	Anycall response received from...	*Average latency*	*Average call time*
Anycalling from planetlab1.diku.dk	planetlab2.comet.columbia.edu	113.88	**166.88**
	planet1.berkeley.intel-research.net	178.46	**302.04**
	pl2.6test.edu.cn	393.67	**538**

Table 3.3. Performance of anycalls

3.2.3.2.1.5 Decentralized load balancing scheme

To measure decentralized load balancing performance in Dermi, we used three nodes out of 20 which provided the same service. A set of clients began making RPC calls on these servers (basically sending a CPU-intensive task). When a server could not accept more jobs, another server from the same group took its place, and so on. The results showed what we expected: the first jobs sent to the group were taken by the closest server to the client (thus confirming the utilization of the *anycall* locality property). When the object was overwhelmed with jobs, the responsibility fell back to the second closest server, and so on. With 300 RPC invocations, the average call times were **173.22**

ms for the first server (*planetlab2.comet.columbia.edu*), **298.33 ms** for the second one (*planet1.berkeley.intel-research.net*), and **546.03 ms** for the last one (*pl2.6test.edu.cn*). The origin client was located in *planetlab1.diku.dk*. The results shown in Table 3.4 demonstrate that the overhead incurred is acceptable, thus showing the viability of this decentralized alternative. This experiment was quite analogous to the one performed before when testing the *anycall* primitive, since the basis for the decentralized load balancing scheme is the utilization of the *anycall*. The results obtained do not differ so much between both experiments and this little difference is explained because PlanetLab nodes run multiple experiments and this can cause significant variations between different runs. However, this was not the case, since the experiment was repeated several times for different lengths of time so that these effects were minimized.

	Object server responding at…	*Average latency*	*Average call time*
Object client located at planetlab1.diku.dk	planetlab2.comet.columbia.edu	113.88	**173.22**
	planet1.berkeley.intel-research.net	178.46	**298.33**
	pl2.6test.edu.cn	393.67	**546.03**

Table 3.4. Performance of the decentralized load balancing scheme

3.2.3.2.2 Simulation Results

For our simulations we focused primarily on Dermi’s distributed interception capability and its associated load-balancing mechanism: *interceptor-based load balancing*.

3.2.3.2.2.1 Distributed Interception

Figure 3.12 shows a simulation of Dermi’s distributed interception mechanism. For the sake of clarity, the figure displays data only for the messages delivered to the event service’s application layer. The configuration used an overlay network of 40,000 nodes and a 20,000-node multicast group. We sent 20,000 notifications to the group and used FreePastry’s [40] local node simulation. We measured the interceptors’ node stress, which shows the number of messages received for such nodes. The first scenario (Figure 3.12a) shows the group with an interceptor located at a node other than its root. Results show the rendezvous-point node overhead: each message is sent twice to the root (from the publisher to the root and from the interceptor to the root). We can improve this scenario by making the rendezvous node and the interceptor the same node (using Dermi’s object-mobility service). In this case, global node stress is the same as if there were no interceptor.

What happens when the interceptor node is overwhelmed with event processing (rather than network load) ? Imagine transmitting a video stream to several groups of users, one of which wants to receive it in a different video format. This is a very demanding task if performed by the video publisher or by each affected group member. One alternative is to use an interceptor to do the data-conversion and deliver it to the group that wants it. Even so, the multicast group’s root node can be overwhelmed with CPU processing if the interceptor and root coincide in the same node. As Figure 3.12b illustrates, we might delegate such demanding processing to specialized interceptor nodes that produce more network node stress on the root; this would free the root from collapsing due to event processing. In this scenario, the root node selects four equivalent interceptors in a round-robin policy. In real life, this illustrative case can be extended to reduce the interceptor's network and CPU stress. A root node's stress increases with the number of remote interceptors, which has the counter effect of relieving the root from unnecessary CPU processing. (Although the rendezvous node can be a relatively CPU-weak node, we selected powerful CPU nodes).

As part of the test, we simulated random failures in these four interceptor nodes using an ad-hoc recovery-mechanism policy that restarted new interceptors when the number of live ones decreased to one. (We can modify this policy to respawn new interceptors when other conditions are met, but for clearer simulation results, we opted for our default condition.) As each interceptor fails, the node stress for the remaining ones noticeably increases. When they are all down except one, our recovery mechanism enters and respawns three new interceptors, thus softening the node stress and load balancing our system to where it was at the start of the simulation.

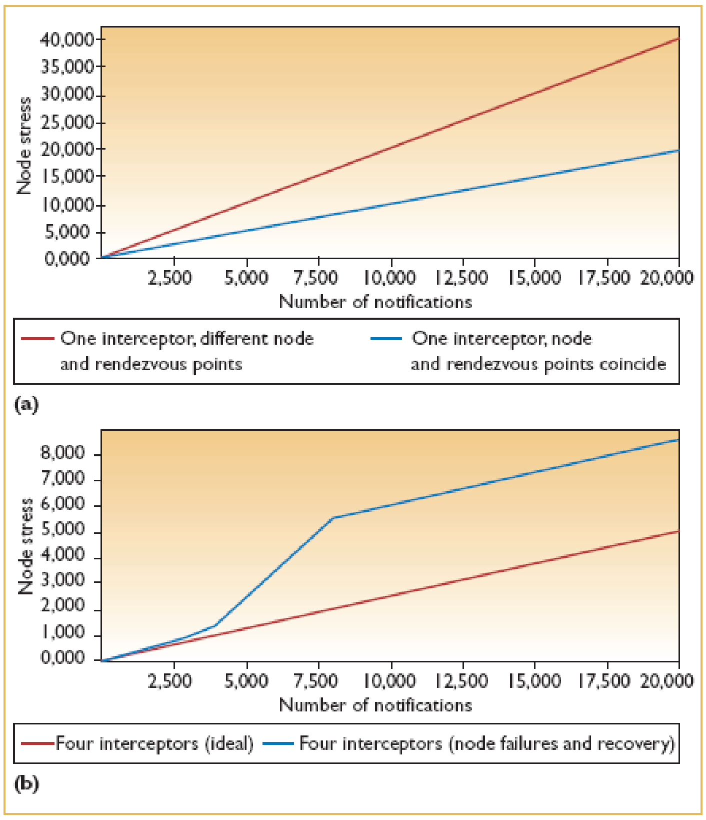


Figure 3.12. Distributed Interception Simulations
(a) Scenario 1 shows node stress for one interceptor node. Better node stress is achieved when the root and interceptor coincide in the same node. (b) Scenario 2 shows mean node stress for four interceptors. The ideal case is shown in red (no node failures). The blue line shows a case in which up to three interceptors fail and then recover. Mean interceptors' node stress increases in such cases.

3.2.3.2.2.2 Interceptor-based Load Balancing Scheme

This experiment is very similar to the previous one, since *interceptor-based load balancing* is based on the use of Dermi's distributed interceptors. Figure 3.13 shows a simulation of Dermi's interceptor-based load balancing scheme. In this scenario, we simulated a group of nodes (objects), all belonging to the same multicast group, which means that their load is automatically balanced by Dermi. For clarity, the figure displays data only for the messages delivered to the event service's application layer, discarding other kinds of messages (namely heartbeats, maintenance, etc.). The configuration used an overlay network of 8000 nodes and a 1000-node multicast group. A total of 20000 messages were sent to this group and we used FreePastry's [40] local node simulation facilities. The multicast group contains a single interceptor, which decides to which node the message will be directed. Note that in a real testbed, we can consider that each node contains a single object, and the interceptor decides how the load balancing of requests is distributed among them.

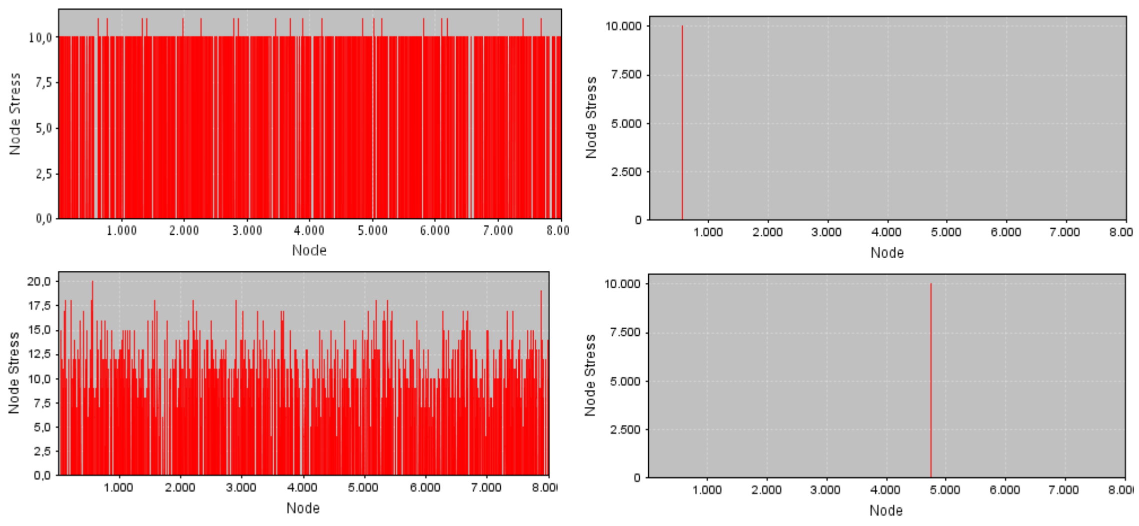


Figure 3.13. Interceptor-based load balancing simulation
The topmost chart shows the simulations using a round-robin load balancing scheme, whereas in the second case, a random assignment is used.

The first configuration used was an ideal case in which the interceptor load balanced among all nodes using a round-robin policy. We focused on the node stress incurred in the group's nodes, which measures the total number of messages received. In this case, we observe the correct (and ideal) message distribution among all nodes in the multicast group. However, the interceptor itself, as well as the group's root node (which in this case, coincide), are by far, the most overwhelmed nodes. This is the main problem that arises in this case: depending on its overall capacity, the interceptor itself may be overloaded with messages. This can be seen in the top righthand chart in Figure 3.13. Naturally, in order to reduce this overhead, the interceptor itself may be replicated as well, as we have discussed in the previous experiment.

In the second configuration the interceptor randomly chose among the objects in the group. Naturally, the results show that some nodes become more overwhelmed than others because of the random distribution used. These results can be extrapolated to the real world, where the unpredictability of many factors such as network bandwith or node congestion may affect the ideal load balancing factors. By using appropriate mechanisms (periodic messages, global state information, etc), the interceptor can choose one node or another.

The same problem as in the previous case arises again: the interceptor can become a bottleneck, so it can be replicated or, if replication is not a viable option, the decentralized load balancing alternative can be chosen.

3.2.3.2.3 Discussion

Using the PlanetLab testbed, we verified that Dermi does not impose excessive overhead on distributed object invocations; one-to-many invocations elegantly fit with the application level multicast service, and anycalls and manycalls obtain good results because of the inherent network locality.

Throughout our simulations, we found that the principal hot spot of our distributed interception mechanism was rendezvous-point overloading (when there was more than one interceptor per group). This problem is endemic to most group-multicast algorithms, and approaches such as creating rendezvous node hierarchies have been proposed in literature to alleviate it. Scribe currently does not support this feature, and although it has the advantages of being a single entry point to the group (thus, being able to perform access control, distributed interception, and so on), it might also become a hot spot in terms of being overwhelmed by messages.

3.3 The Wide-Area Distributed Component Middleware Layer

The continuous advances in Computer Science and Telecommunications have changed the way software applications are being developed. In particular, the increase in computing capacity, the reduction in hardware and communications costs, and the emergence of global data networks have maximized the use of distributed systems. This means that existing programming models cannot naturally cope with the complex requirements of these kinds of systems. Therefore, new programming paradigms like coordination, component-oriented programming, or mobility appeared to improve software application development processes.

Nowadays, one of the most popular approaches is *Component-based software development* (CBSD), which tries to establish a basis for design and development of reusable software component-based distributed applications. This discipline has attracted increasing interest from the academic as well as the business point of view.

Traditional component-based architectures are mainly client-server based. This approach is suitable for local-area and metropolitan-area scope applications. Many technologies like EJBs [46], CORBA CCM [35] or DCOM [32] (and its evolution, Microsoft's .NET Framework [54]) have proven to be successful in these fields. However, when trying to develop a component-based application which is to be deployed in a wide-area environment, limitations of the client-server architecture rapidly arise. Subsequently, the server itself may become overwhelmed with multiple client requests. An array of clustered servers is sure to help in such a case, but may well be economically expensive.

Here is where decentralization and the wide-area remote object middleware described throughout this chapter can be useful. Although our remote object layer is a solid building block for wide-area distributed applications, it is not as abstract as the developers would like. From the software engineering perspective, it would be interesting if we could develop reusable component-based applications, and a remote object middleware is usually not enough. A lightweight component-oriented model would facilitate even further the development of worldwide accessible applications. Therefore, we believe evolution is required to a higher level of abstraction, to *decentralized wide-area components*.

Wide-area component models are not a novel concept. As we explained when describing related work in this field, some approaches use Grid (ProActive [60]) or unstructured p2p infrastructures (P2PComp [75]). However, we propose a new decentralized component model built on top of a generic wide-area remote object middleware. Our model provides traditional component services like naming, activation, passivation, event notifications, and persistence on top of a decentralized infrastructure. We use the underlying object middleware to implement these services in a decentralized and efficient way. Specifically, component deployment and location takes place in a decentralized and fault-tolerant naming service. When components are to be activated and invoked, we use the network's locality properties to call the client's nearest replicated instance. For stateful components, state will transparently be maintained by all their replicas. The different events transmitted from and to the component instances are propagated by an application-level multicast protocol. Moreover, to handle possible high invocation peaks, we use decentralized load balancing techniques, and new component instances are subsequently activated on demand.

The main contributions of our **wide-area distributed component framework** are the following:

- It is a new **wide-area component model** that runs on top of a decentralized network infrastructure.

- It uses a **decentralized component location and deployment facility**.

- It implements a resilient and autonomous **lightweight container model**, which provides component's life cycle services, and many others. Component instantiation is of special interest because it takes into account underlying network locality properties (if available).

We have built a prototype implementation of this distributed component framework called **p2pCM**. This middleware runs on top of Dermi and it uses all of its services, which benefit from using a structured peer-to-peer overlay network approach.

In the rest of this section, we describe our distributed component model, its architecture and services, as well as its innovative services and its lightweight container model. For more details about the practical implementation prototype **p2pCM**, please refer to the annexes.

3.3.1 Architecture and Services

Before describing our new component model in greater depth, we briefly define the terms *component* and *component model*.

In [112], a component is defined as a *unit of software application composition with contractually specified interfaces and explicit context dependencies that can be developed, acquired, added to the system and composed of other independent components, in time and space*.

Component interfaces determine the operations that a component implements, and the operations it uses from other components during its execution.

A distributed component-oriented model is an architecture that defines components and their interactions. It must provide a packaging technology for deploying binary component executables. Moreover, it needs a container framework for injecting life cycle, thus permitting activation and passivation of component instances. Other services include security, transactions, persistence, events, and others.

Our wide-area distributed component-oriented model is thought to implement most traditional component model services, although they adapt them to the underlying decentralized topology. These include:

- A **decentralized component location and deployment facility**.
- A **decentralized lightweight container model** which provides components with the following services.
    - Component life cycle service
    - Component persistence
    - Adaptive component activation

The following sections describe each of these innovative services.

3.3.1.1 Decentralized Component Location and Deployment

All components must be previously registered in the system so they can be used by any client. This *deployment* phase is found in all traditional component models, as well as in ours.

The idea is that any component can be *packaged* into an archive, jointly with its metadata. It is then uploaded to the system, where it is automatically *deployed*, and its metadata registered in some kind of naming service. This metadata includes all types of information that a component should make public about itself, and its properties. This information allows containers, environments, development tools, and other components to discover the functionalities that a component provides via *introspection*.

The factory class contains all the necessary annotations for describing the component. Apart from component's metadata, the component archive must somehow be uploaded to the system. Traditional component models normally provide a centralized naming service, in which a component's metadata is bound and becomes available to clients. However, when trying to apply the same paradigm in wide-area network environments, this approach is not sufficient. Over time, the centralized naming entity can be overwhelmed with requests.

In our model, we use the remote object middleware's decentralized naming service to store all the metadata of a component (read in *deployment* time), as well as its class files. This is done to allow dynamic component class loading in clients that do not have the necessary component classes. As described above, our naming service benefits from the efficient overlay network routing properties thus hashing the component's identifier, and storing the values in the node whose key is closer to this hash.

We need to provide some kind of fault-tolerance mechanism to support possible node failures. To do so, we use the persistence service provided by our remote object middleware.

When attempting to instantiate a component, we first need to query the naming service to find its metadata. To do so, we only need the component's identifier, which can be an absolute URI like *p2p://results.deim.etse.urv.cat*, or any kind of pre-generated GUID like in COM [32]. Notice that our approach differs quite considerably from traditional component models, which need to know the address of the machine that hosts the naming service. In our case, we only need to know the component's identifier: the naming service runs on all of the nodes in the network. To which machine (or node) a component's metadata is mapped is irrelevant to us.

3.3.1.2 Decentralized Lightweight Container Model

Components normally exist and cooperate within *containers*. Containers are software entities which can hold other entities, and provide a shared interaction environment.

The container is responsible for managing components' life cycles, and notifying them about life cycle events such as activation, passivation, or transaction progress. Components provide event interfaces that the container automatically invokes when particular events occur. The container also provides components uniform access to services such as persistence, security, transactions, and many others. In traditional client-server based component models, the container itself is usually based on a web application server, database server, operating system, etc. These kinds of containers are rather monolithic and consume large amounts of resources thus requiring powerful machines to run on. This philosophy remains in stark contrast with that of p2p, where machines are usually treated as equals, and applications run on them must adapt to each node's own capacity and limitations.

For our component model we have taken these considerations into account and opted for designing a **decentralized lightweight container model**. In our case, all of the nodes that belong to the network are containers and, as such, they can house many components. The idea is that any component can be run in any node (except for any restriction), because each node runs a lightweight container.

Our containers are fault resilient and autonomous: components are replicated throughout the network. If a container fails, surely other containers housing those components will exist in the network, because they are automatically replicated.

Now we shall describe the different services our decentralized container model offers to components.

3.3.1.2.1 Component Life Cycle Service

The container is responsible for the activation and passivation of components. The process of a client interacting with a component instance starts with the location of the factory class, which is responsible for obtaining a reference to the component by calling ***createInstance()***.

The factory method ***createInstance()*** accepts several parameters, which include the component's interface to be returned (our model supports the extension interface pattern; that is multiple interfaces), and the instance unique identifier. When calling this method, several things may happen:

- *No other component instance is already active in the network*. In this case, the component instance will be activated in our local container, and a local reference will be returned so as we can interact with it.

- *Other component instances are already active in the network*. In this case, we return a reference to the **closest instance** (basically, an *anycall* is done to get the closest instance's stub) to us. However, if this closest instance informs us that it cannot accept more requests (it may be overwhelmed), a local copy is activated. Note that more complex algorithms could be adopted, which depend on the component's utilization pattern. A study of more techniques is beyond the scope of this thesis.

Whatever the case, a reference to the component is returned (unless an exception is thrown). Once we have this stub reference, we can call the interface's methods, which are executed on the remote component. We have adopted a similar approach to that of COM/COM+ [80] for supporting multiple interfaces. Every component interface extends ***ComponentInterface*** (similar to COM's *IUnknown*), and it thus provides the ***queryInterface()*** method to obtain references to the other component's interfaces.

Figure 3.14 shows this component model's general architecture. Basically, the idea is that any client willing to access a wide-area component needs to look the component factory up on the decentralized naming service. The client requests the factory class from the naming service and, when instantiated, it asks the corresponding container node to return a component stub reference to the client. The client can then call the component via the stub.

Component instances are passivated when they do not receive any remote invocations for a certain amount of time so as to save resources on the node. This time depends on properties that are determined by the container itself, which include memory utilization, network bandwidth, etc. The container calls the ***passivate()*** method on the component, so it can proceed with the appropriate instance finalizations.

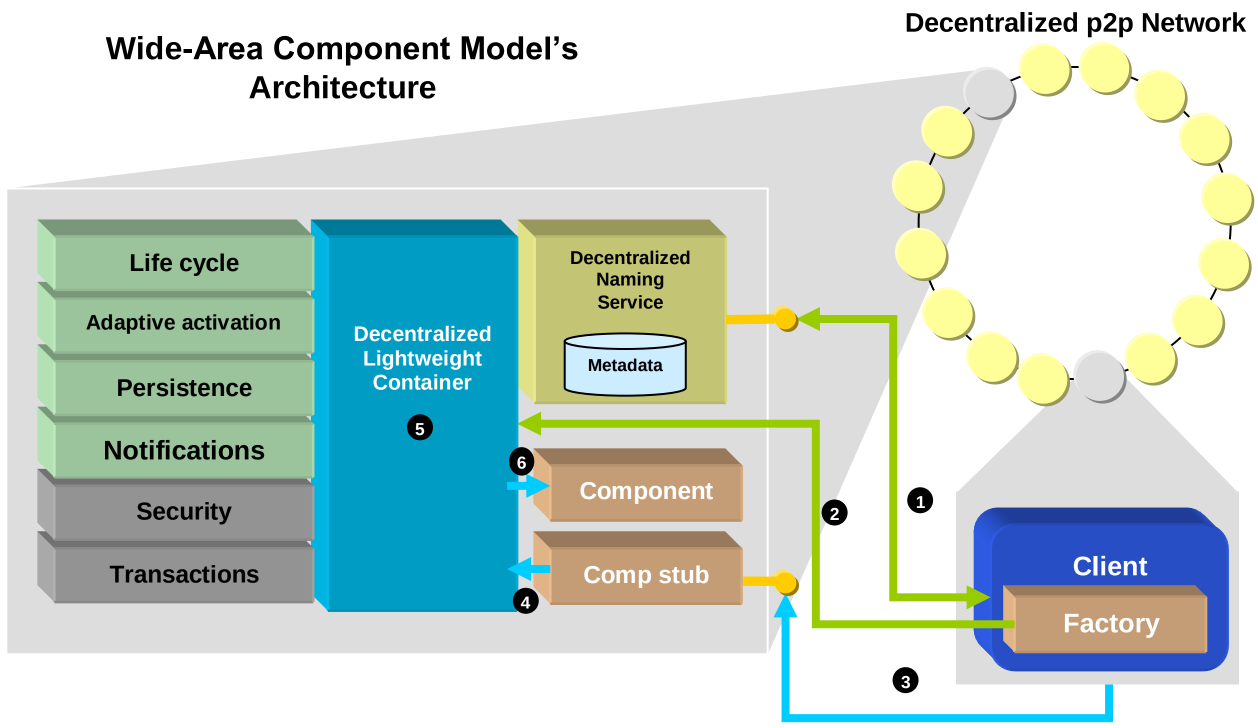

Wide-Area Component Model's Architecture
Decentralized p2p Network
Life cycle
Adaptive activation
Persistence
Notifications
Security
Transactions
Decentralized Lightweight Container
5
Decentralized Naming Service
Metadata
6
Component
Comp stub
4
2
1
3
Client
Factory

Figure 3.14. Diagram of the Wide-Area Component Model's architecture
Clients look up the component factory, which returns a component instance reference residing on another node. Note that each node in the network runs a decentralized lightweight container runtime environment. The client requests the factory class from the naming service (1), and when instantiated, it asks (2) the corresponding container node to return a component stub reference to the client. The client can then call the component via the stub (3, 4, 5, 6).

3.3.1.2.2 Component Persistence

Components in our model can be of two types: **stateful** and **stateless**. Stateless components do not maintain a state while stateful ones do. As stated in the section above, component instances may be replicated throughout the network. This object replication is maintained by Dermi joining all replica objects into the same multicast group. Whenever the object's state changes, an event is sent to this group, and all replicas update accordingly.

For stateful objects, it is also necessary to provide a persistence mechanism to take care of the total passivation of component instances. If all component instances are passivated, their shared state is lost. The persistence strategy is chosen by the component itself by means of the *persistence* annotation in the deployment phase. Currently there can be two types of persistence:

- **Container Managed Persistence** (*PersistenceType.CONTAINER*). In this case, just before calling the *passivation* method, the container calls ***getRemoteObjectState()*** on that instance. The returned object's state is then inserted and replicated accordingly using the naming service. The next time a component instance is activated, and no other running instances are found, it will get the state from the naming service.

- **Component Managed Persistence** (*PersistenceType.COMPONENT*). In some cases, the above method is not enough because more fault tolerant mechanisms are needed. Persistence is managed exclusively by the component. Whenever the ***passivate()*** method is called, it can perform its own persistence strategy, or it can even use a timer which stores the state every *x* seconds.

3.3.1.2.3 Adaptive component activation

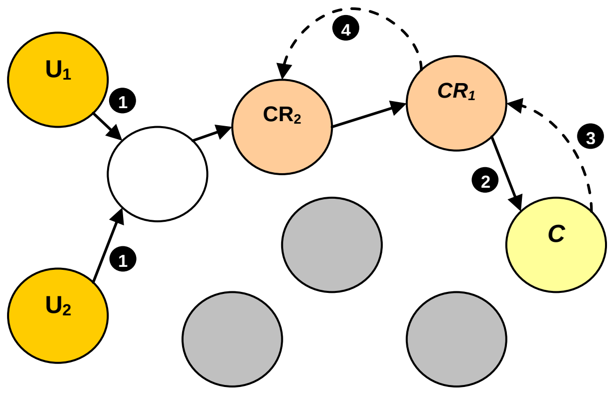
Component instances are created when a client wishes to interact with the component. Depending on the closest instance and load, new replicas are created or the already active instance is used. In either case, and especially in wide-area environments, some instances could become temporarily overwhelmed with requests from nearby clients. To avoid this problem, we use an **adaptive component activation** mechanism, which we shall describe below (see Figure 3.15).

Each container is set on a physical node in the network and runs a key-based routing overlay substrate. The node can easily know which immediate nodes are delivering messages to it (simply by attaching previous node information on the message by means of a ***getPreviousHopHandle()*** method). The container on each node should continuously sense different node parameters (memory and CPU utilization, network bandwidth, number of component invocations, etc.) for each component instance. If an upper threshold is surpassed, it means the component (*C*) is becoming overwhelmed with a request peak and we start the adaptive component activation algorithm.

The affected container identifies the immediate peak request forwarding neighbour, and it is ordered to activate a new replicated component instance (*CR*) on it. Note that this newly activated entity falls within the natural pathway from the client to the destination component instance. This scheme improves and de-stresses the original instance (*C*) but if left alone, it would absorb the peak all by itself. The strategy adopted by *CR* is to act as a *request faucet*. For all of the requests addressed to *C* which fall into the pathway that leads to *CR*, *CR* will process some of them (according to a specific load balancing policy that should be dynamically calculated). This scheme diminishes the number of requests to *C*, and achieves the desired request peak load balancing. Should the number of requests be incremented and the threshold surpassed once again, the methodology would be the same. Consequently, placing component replicas all along the approaching overwhelmed pathway would reduce the stress on these component instances.

Once the request peak has passed, component instances that have not been called for a certain amount of time would be automatically passivated by the container. This scheme would activate replicas on demand where there is more need, and passivate them when they are not required.

The activation hooks for this adaptive activation scheme have been designed for our **p2pCM** prototype, and we are working on getting container node parameters (possible through Java 1.6's onwards). Therefore, many parameters, such as the *request faucet* drop rate, still have to be fine tuned,. Evaluating and performing tests for this phase remains as future work.



1. Clients ***U$_1$*** and ***U$_2$*** are continuously invoking components on ***C***.

2 - 3. ***C*** is overwhelmed and activates a new replica (***CR$_1$***), which absorbs many calls directed to ***C*** from the same pathway.

4. Despite the new replica, request stress is not alleviated and ***CR$_1$*** activates a new one (***CR$_2$***), which now suffices.

Figure 3.15. Adaptive component activation scheme

3.4 Prospective Uses of our Proposed Wide-Area Middleware

We foresee many prospective applications that could benefit from the services provided by our middleware proposal in the next few years. These application-domain examples include:

- **Enterprise edge computing**. Akamai's EdgeComputing initiative [3] proposes that enterprises deploy wide-area systems that are accessible from different countries to benefit from the locality of the Akamai network. This means that the code that reads, validates, and processes application requests executes on Akamai's network, thus reducing the strain on the enterprise's origin server. Our services can also provide resource location and network locality in a decentralized fashion without depending on proprietary networks. For example, a company could deploy a wide-area e-commerce system interconnected with our underlying p2p substrate and using Dermi's services.

- **Grid computing**. The Open Grid Services Architecture (OGSA) [37] has standardized several services for developing distributed computing applications. Grid components can be accessed using Web services and located using naming services. State changes can be propagated using event services. Nevertheless, there is ongoing research into creating other wide-area grids. Our services could help solve many of the problems this work will encounter. For example, our decentralized object-location service can be very useful for locating resources in a deterministic way. Further, we can exploit grid locality with anycall abstractions and propagate changes using multicall abstractions. Our distributed interception mechanism could also be used for load-balancing purposes.

- **Multiagent systems**. The Foundation for Intelligent Physical Agents [17] specifications define a multiagent framework with services for agent location and a messaging service that supports ontologies. Nonetheless, the research community has not yet proposed a wide-area multiagent system. For example, AgentCities [2] utilizes a central ring to interconnect several agent infrastructures. (The AgentCities network is open to anyone wishing to connect agents and services. The initiative already involves organizations from more than 20 countries involved in a significant number of different projects.) A scalable multiagent system could use the services we propose to achieve decentralized agent location; it could also benefit from network locality to use agent services and from multicalls to propagate state changes simultaneously to many agents.

Finally, the *Computer-Supported Cooperative Work (CSCW)* domain is an interesting arena for wide-area applications. Developers could use our infrastructure to build social networks, massive multiuser games, and online communities.

- **Shared Session Management**: a shared session is essential to any CSCW toolkit because it defines the basis for shared interactions in a common remote context. Shared sessions can be implemented as reusable distributed components. This approach helps us benefit from several p2pCM component services, some of which are implicit:
  - **Session location**. Session components are instantiated as components. This means that if any instance of this session already exists on the network, we can interact with it directly. Moreover, if several of them exist, we are guaranteed to interact with the one *closest* to our node, in terms of network proximity.
  - **Session persistence management** is accomplished by specifying the default session component persistence policy. This feature of our component middleware involves serializing an object's state into the decentralized naming service, which replicates this data among the destination node's closest neighbours.
  - **Session state propagation**. Whenever session state has to be propagated among all other session users, the *multicall* primitive from the remote object layer is used. All session instances are rooted in a multicast tree. When an event is triggered from any of them, all other replicated instances are updated accordingly in a transparent manner. This approach can also be applied to any of the session's inner objects.
  - **Session load balancing** is accomplished by the *adaptive component activation* scheme. Whenever any of the session's instances becomes overwhelmed for any reason (typically a bunch of requests), another instance is activated in the most overwhelmed neighbour node. Therefore, it will alleviate the request rate and achieve load balancing in a transparent way to the developer.
- **Awareness Mechanisms**: an awareness component can also be created to monitor shared session interactions. We can use these services:

  - **Distributed interception**. This feature is used for *attaching* the monitoring component to the components it should monitor for interactions. In this case, every time an event (interaction) is sent (via *multicall*) to the object's multicast group, the awareness component will intercept the event and store it for later statistical or data mining purposes.

  - **Other services**. Like the session component, the awareness component benefits from decentralized persistence management, location, and load balancing.

- **Coordination policies**: The main functionality of the coordination components is to establish group coordination policies among groups of objects contained in a shared session. Basically, the main feature used is:

  - **Distributed interception,** which again plays an important role in achieving coordination. Before sending an interaction (event) to an object, this event is captured by the coordination component, which will check whether the user, object, etc. is authorized to perform the action in a coordinator-specified role.

We believe that these essential CSCW services can be very useful for developing wide-area collaborative applications.

3.4.1 A Sample Distributed Computing Application Scenario

One scenario in which we could use our middleware services is the development of a wide-area distributed computing application. We should point ut that the services provided by our infrastructure are hooked into the application development cycle. This section describes a sample case for our wide-area middleware proposal. The next chapter introduces a real implemented case: **SNAP**.

One possibility is to develop a wide-area distributed computing application similar to a SETI@Home [41] or United Devices Cancer Research Project [23] model. These applications normally require a central server which distributes computing units to home computers for analysis. Our component model could be used to build an application which would be efficiently fault tolerant and resilient to high request peaks.

Our approach does not discard bridging our component model with others. Continuing with our example, we must assume that there is an application server which houses a heavy component responsible for reading data from Arecibo's radiotelescope [8]. This *DataFeed* component could be an EJB or a CORBA component. The *DataFeed* can communicate with other external components which transparently run on p2pCM.

The idea is that interested nodes activate a *Processing* component on their local containers (this can be done by the application clients themselves, or directly by the *DataFeed* component, depending on the activation policy). This component can thus be receptive to the *DataFeed* component's calls. The *Processing* components are aligned into a multicast group, so that the data feeder can request data unit analysis by *anycalling* or *multicalling* to the *Processing* component group.

Once each *Processing* component finishes its data unit analysis, instead of returning data to the central *DataFeed* component (which may create an important bottleneck there), it passes data to another *Results* component by means of an event. This component manages the results persistence by replicating itself and storing state information in the DHT infrastructure.

Whenever the *DataFeed* component wishes to obtain the results (cyclically every *x* hours), it will *anycall* its closest *Result* component instance, which provides it with the latest results. Additionally, at the output of the *Result* component, we can attach a *Filtering* component, which can be queried by the *DataFeeder* to obtain the most significative results (i.e. *I want the 10 most significative Gaussians so far*). A diagram of this architecture can be seen in Figure 3.16.

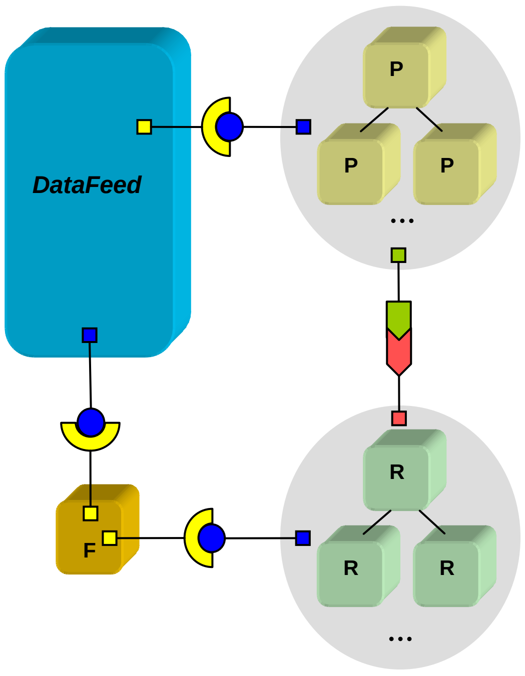


Figure 3.16. Component diagram of the p2pCM's sample distributed computing application (P refers to *Processing* components, R to *Results*, and *F* to *Filtering*). *P* and *R* are, in this case, replicated component instances. Note the interactions between different components.

This is a usage scenario in which we believe our component model could be useful. Nevertheless, as we have pointed out at the beginning of the previous section, other combinations of p2p networks and our component model are possible.

3.5 Summary

In this chapter we have presented the two pillars which sustain our whole wide-area middleware proposal. We first introduced a **wide-area decentralized event-based object middleware** (and its **Dermi** implementation). We have analyzed its architecture and described all the services it provides, which make good use of the three main core layers (see previous chapter): the routing layer, the application-level multicast layer, and the persistence layer.

Our Dermi prototype is based on a structured peer-to-peer network substrate, and we have shown that it is an efficient and viable object middleware by means of experimental and empirical evaluations.

Dermi is not a finished product yet. It is continuously evolving, and it can be downloaded from http://dermi.sourceforge.net. We are continuing to improve it by including several useful features. In contrast to sequential manycalls, for example, we are thinking of implementing *parallel* manycalls for better performance. The behavior would be similar to a multicall, as we again multicast to the tree, though starting from the closest client's node in the group, rather than from the root itself, thus taking locality into account. Results could be communicated back to the client by all the group's nodes that satisfy the condition. Once the client receives all necessary data, it discards other incoming messages from remaining group members. For very large groups, a multicast tree search could be purged by specifying a maximum depth to cover, thus preventing a client from becoming overwhelmed with messages. This methodology would incur more node stress in the client, but it would be more efficient in terms of parallelism. We are also thinking of adding authentication mechanisms to prevent malicious nodes from compromising our system's participants. Public-key cryptography is required to achieve this goal, which adds a performance expense, although the KBR layer should provide us with some security primitives to get Dermi hooked into. Finally, we are making our rendezvous-point change buffering algorithms more consistent to account for all possible uses.

The second main block of our entire proposal is a **decentralized component model framework** (and its practical implementation, called **p2pCM**). This model builds on top of the remote object middleware layer, and it provides a higher level of abstraction to application developers by permitting components to be reused through many applications.

Therefore, our wide-area middleware proposal fulfills all the requirements enumerated in the previous chapter. These are:

- **Scalability**. A Wide-Area Middleware framework must be based on a scalable routing substrate for efficient message routing. *Structured peer-to-peer overlay networks* provide an efficient and scalable message routing substrate. Therefore, communication between decentralized nodes is efficient and follows a *Key-Based Routing* scheme, where a specified *key,value* pair is efficiently mapped to a particular node, in the form of a giant distributed hash table (DHT).

- **Fault tolerance and high availability**. These requirements are fulfilled by our middleware proposal, since it provides transparent mechanisms for data access and replication. Any object or component can be automatically replicated and its state transparently maintained by our middleware. This way, once a node housing any object or component becomes unavailable, the *closest one* takes its place, by means of a transparent internal *anycall* invocation.

- **Load balancing**. Our middleware provides two different load balancing mechanisms. One is based on the *distributed interception* feature, and the other on the *anycall* abstraction. Both of them are complementary and may be used in different scenarios. This service is also available to application developers, which may declaratively specify which load balancing scheme they wish to utilize for their objects, components, and applications.

- **Dynamicity**. Heterogeneous and very different types of nodes may constantly join and leave the overlay network. In this case, the overlay network transparently manages node departures and arrivals for us. Once a node joins the network, it acquires *responsibilities* (that is to say, it is given key-pair values to store), and once a node leaves or disconnects abruptly, its replicated values are redistributed among the remaining participating nodes.

- **Use of the resources on the edges of the Internet**. The idea is to make good use of all the scattered and unused resources of any of the peers forming the network. Since it is a peer-to-peer decentralized network, all members are treated as equals and, therefore, they can all contribute resources to their conforming network. As a consequence, storage and network bandwidth is shared between all participants, since they have to hold state and replication data, and also permit messages to be routed (and forwarded) through them.

- **Usability and programming abstractions**. Our middleware has been designed with usability in mind. Therefore, our prototype implementation has been specially architected following the ease-of-use directive. Thus, it is very easy to develop new objects, components, and applications on top of Dermi and p2pCM. This ease of use is demonstrated in the annexes at the end of this thesis, in which we describe our middleware's API. The use of Java annotations enormously facilitates method *tagging* for specifying desired features and attributes. Moreover, we provide the developer with numerous programming abstractions, which include the availability of remote objects and/or components, a scalable object/component location service, and a set of innovative group communication abstractions (namely *multicall, anycall, manycall,* etc).

We have demonstrated that our wide-area middleware proposal complies with all due requirements. Other wide-area middleware approaches lack some or many of these requirements, making them difficult to use, since many services are not implicitly provided or are even non-existent. Moreover, our proposal is, to the best of our knowledge, the first to envision a wide-area middleware framework based on a structured peer-to-peer overlay network.

In the next chapter, we introduce a proof-of-concept implementation of a wide-area application which uses our remote object and component middleware. Even though p2pCM provides the developer with components, a new API and way of programming must be learned. Our first idea envisioned some kind of a global framework for developing wide-area applications whose learning curve was as low as possible, thus maximizing the **usability** requirement. The idea of having lightweight containers was the tip of the iceberg; and as we were strongly inspired by the J2EE application framework, we thought we could *adapt* it to the wide-area world, trying to avoid bottlenecks, scalability problems and making it transparently fault tolerant.

This is how our wide-area deployment framework application called **SNAP** was created, and it serves as a proof of concept for our proposed wide-area middleware architecture, since it is built on top of it and makes extensive use of its services.

Chapter Four

4 Applications of our Middleware: SNAP

4.1 Introduction

In this chapter we present a use case of our proposed architecture and its concrete implementation in the form of Dermi and p2pCM. We present **SNAP**, which aims to be a wide-area J2EE-compatible web application deployment framework, for easy transitioning from client-server applications to wide-area ones. This product can be downloaded and evaluated from its website: http://snap.objectweb.org.

As we said throughout this thesis, it is not usually easy to develop a wide-area distributed application from scratch, since not so many wide-area middleware solutions exist. As a consequence, developers have to waste their time reinventing the wheel by building common services. Such basic services include fault tolerance, data replication and caching, security, application-level multicast, and many others.

Even though it is not so easy to develop these kinds of applications, some p2p toolkits have emerged mainly for the development of collaborative enterprise applications. Perhaps one of the most widespread is Groove [24], which follows a p2p model to initiate and maintain workspaces where users exchange text with instant messenger software, applications, voice and video in real time through various panes within one frame or skin. Enterprise collaboration is nowadays a hot topic and this is demonstrated by the acquisition of Groove Networks by Microsoft Corporation. The future relevance of wide-area collaboration is becoming a reality.

When collaborative applications need to be implemented, Groove provides an extensive API. Naturally, these applications are designed to be run within Groove's proprietary environment. Moreover, if these applications are to be implemented, this new API, which is not generic enough to support wide-area p2p applications, must be understood.

Therefore, there is a growing need for a wide-area p2p platform which allows developers to build their applications, with no need to learn a new programming API. Access to persistence, security, and failover services should be transparent. But is it possible for a traditional client-server web application to be easily transformed into a p2p web application with minimal descriptor changes thus providing worldwide scalability? To answer these questions, we propose **SNAP**: Structured overlay Networks Application Platform.

SNAP is completely based on our previous contributions (Dermi and p2pCM), and it offers a novel approach, which includes:

- **Easy adaptation of existing client-server J2EE-compatible applications to a wide-area p2p scope**: adding a new XML descriptor is enough to port a J2EE web application to SNAP.

- **Secure web application deployment** on a decentralized infrastructure. This means that only authorized users can deploy previously signed web applications on SNAP.

- **Decentralized embedded services** like *persistence, load balancing, fault tolerance* and *edge computing* are also available in the SNAP application development.

SNAP is thus oriented to wide-area lightweight web application deployment on a decentralized network. Since it follows the p2p paradigm, it is designed for lightweight applications which do not require huge amounts of node resources. It can be suitable for enterprise wide-area collaboration, the elaboration of wikis, weblogs, and static webpages that require worldwide scalability.

Our system is clearly not adequate for applications relying on large back-end databases, since it would be infeasible to copy and synchronize large databases to light and volatile p2p clients. Moreover, those applications which deal with Enterprise JavaBeans (EJBs) are also excluded due to the heavyweight nature of the EJB components themselves.

We believe SNAP can become the foundation for enterprise collaborative application deployment on a secure, non-proprietary p2p network. Throughout this chapter we describe SNAP's architecture and services, which have been validated by an empirical evaluation on top of the PlanetLab network.

4.2 Related Work

To the best of our knowledge, SNAP is the first wide-area oriented platform to provide decentralized deployment of J2EE-compatible web applications on top of a structured p2p network.

Chameleon [55] is a decentralized middleware design that dynamically allocates resources to multiple service classes inside a global server cluster. It is based on an epidemic protocol for disseminating state and control information. All services are expected to run within certain Quality of Service (QoS) parameters. A request routing algorithm is implemented which forwards requests to other neighbor nodes if a server is overloaded. This is similar in SNAP, which uses the underlying *anycall* abstraction. However, in SNAP we currently consider network proximity as a QoS parameter. We are working on developing more complex QoS algorithms to allow dynamic application adaptation.

As far as **edge services** are concerned, Akamai's EdgeComputing initiative [3] proposes that enterprises deploy wide-area systems that are accessible from different countries to benefit from the locality of Akamai's network. This means that the code that reads, validates, and processes application requests executes on Akamai's network, thus reducing the strain on the enterprise's origin server. SNAP's services also provide resource location and network locality in a decentralized way, but without depending on proprietary networks. For instance, a company could deploy a wide-area collaboration system using the SNAP infrastructure. However, Akamai's nodes are going to be more robust than the traditional p2p network members, since they are static dedicated servers. They are used, then, to improve performance, take advantage of network proximity and provide outstanding reliability. On the other hand, SNAP targets a different domain. We provide lightweight edge computing, because the p2p network is less reliable, and we target different application domains: collaboration, wikis, web services, etc.

Our concept goes further than such other approaches as p2p web hosting (YouServ [61]) or structured p2p content distribution networks (Coral [77]). SNAP provides true application deployment based on web standards on top of a p2p network. Coral is a structured p2p content distribution network, which allows a user to run a web site that offers high performance and meets huge demand. It uses a p2p DNS layer that transparently redirects browsers to participating caching proxies, which in turn cooperate to minimize load on the source web server. SNAP's location service (*p2p://...*) is similar to Coral's (*.nyud.net:8090*). Nevertheless, Coral uses another indexing abstraction (called a *distributed sloppy hash table (DSHT))*, which creates self-organizing clusters of nodes that fetch information from each other to avoid communicating with more distant or heavily-loaded servers. However, the purpose of SNAP is not only to provide a redirection service, but also to allow secure web application deployment as well as the other services mentioned above.

In the Grid world, ActiveGrid Grid Application Server [1] is designed to scale applications across horizontal grids of commodity computers. ActiveGrid Grid Application Server interprets applications at runtime and can deploy them using a variety of proven deployment models and multiple data caching patterns. Indeed, this solution provides many interesting and scalable services like dynamic node registration, data caching, session management, transaction management, and interface fragment caching. However, SNAP's scope is completely different, since it is not aimed at grid computing or real-time applications. For these domains, solutions such as ActiveGrid are more suitable.

Other related projects include Magi [63], which enables information sharing and messaging on any device using web-based standards like WebDAV [52], or XML. It also includes a micro-Apache HTTP server [4] which provides a link to every instance of Magi, a set of core services, and a generic extensible interface. However, Magi is not based on a structured p2p network so it does not provide services like network proximity awareness, which SNAP uses extensively to locate closest replicas.

The JxTA project [39] is also partly related to SNAP since its main aim is to provide an open collaboration platform that supports a wide range of distributed computing applications which enables them to run on any device with a digital heartbeat. Even though JxTA provides a set of APIs ideally suited for wide-area application development, their design is too low-level, and they are too complicated to learn and use. They do not even deal with p2p components, which is a major drawback when developing complex applications.

4.3 Architecture

SNAP's architecture is shown in the diagram below (Figure 4.1). The idea is that each node in the structured p2p network holds a copy of SNAP running. The members of the p2p network are typically end-user machines which may join and leave the network frequently.

Each node in the overlay network is conformed basically by a p2p network routing layer, which routes all messages to / from the network. Upper layers are the object (Dermi) and component (p2pCM) layers through which the lightweight webserver interacts with the p2p network. The SNAP core component provides all of SNAP's services to web application developers. We shall now describe each of SNAP's layers in order to better clarify the figure.

4.3.1 Peer-to-Peer Routing Layer

As we mentioned earlier, p2p systems and algorithms have evolved in recent years. From the early central index scheme used in Napster, and the flooding techniques of Gnutella, to the structured p2p key-based routing (KBR) overlays there has been an important leap. We have chosen to sustain SNAP on top of a network of this kind, thus taking advantage of the Dermi remote object middleware, and the p2pCM reusable component framework. Therefore, SNAP can exploit its already well known inherent properties: scalability, fault-resilience, self-organization, routing efficiency, network proximity organization, etc.

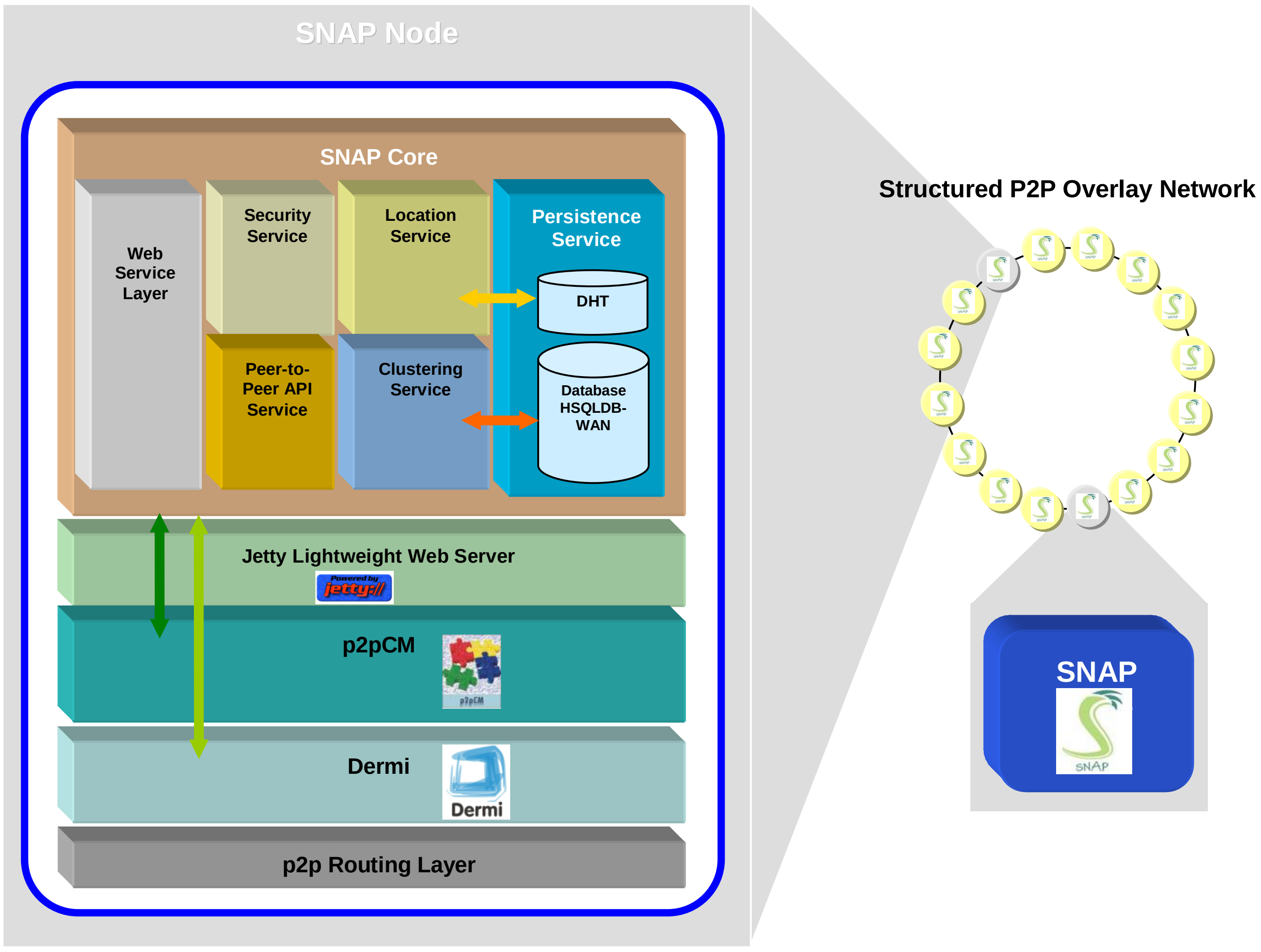


Figure 4.1. SNAP's architecture diagram
SNAP's services are sustained on top of the Jetty Web Server, as well as the p2pCM component model, and the Dermi object middleware.

4.3.2 Dermi and p2pCM Layers

On top of the routing layer, there is a distributed object layer (Dermi), on top of which there is a distributed component layer (p2pCM). Both these layers provide the foundations for SNAP's services.

4.3.3 Lightweight Web Server Layer and SNAP's Core Layer

Since the aim of SNAP is to allow decentralized web applications to be deployed, we must make this possible in each of the network's conforming nodes. This is why we need a lightweight webserver which runs on each member of the SNAP network. **Jetty** [28] runs in the **Lightweight Webserver Layer**, since it perfectly suits our requirements: it is lightweight enough, and it is also written in Java. This is the layer that allows web application access and deployment for each of SNAP's nodes. Therefore, clients can connect through their favourite web browser to any of SNAP's nodes in order to access any deployed web application.

The last layer in SNAP's architecture is the one containing the system's core. This is the most important layer, since it glues all the others together and provides the innovative services that developers can benefit from. This layer is described in the next section and focuses on the services provided.

4.4 Innovative Services

SNAP's Core and Services Layer essentially contains SNAP's main kernel, which includes all of the functionalities and services used and exposed to web applications. Before moving on each of SNAP's services, we outline the most important pillars of our infrastructure. These are **secure web application deployment**, and **web application activation on demand**, both of which are vital for understanding how SNAP works.

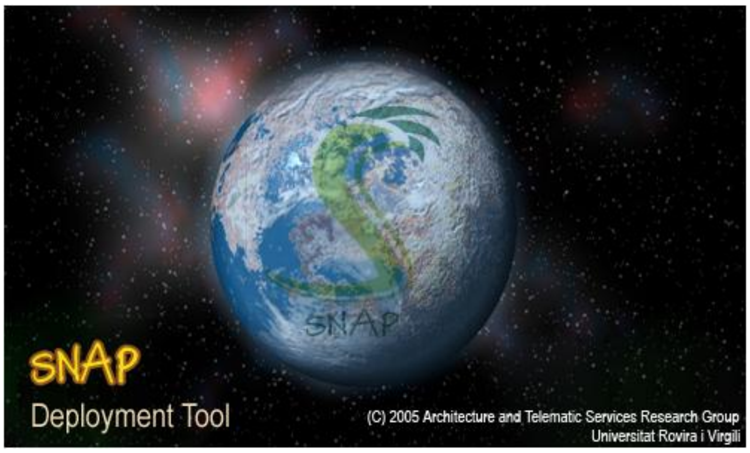
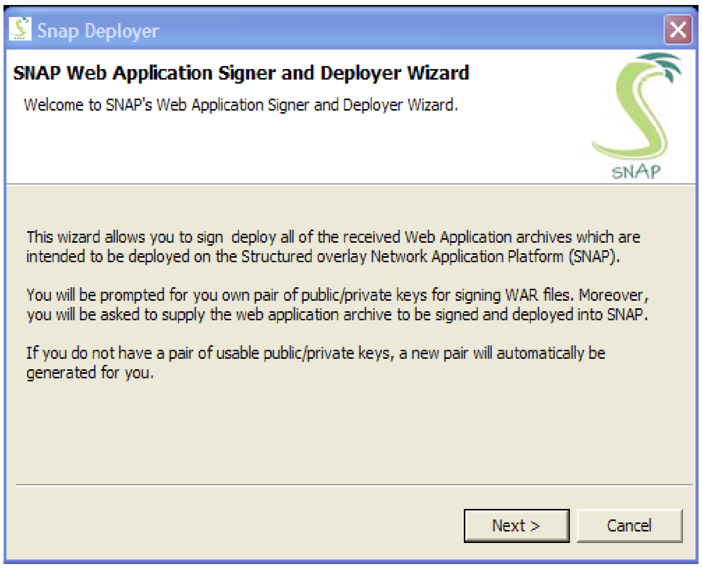

SNAP
SNAP
Deployment Tool
(C) 2005 Architecture and Telematic Services Research Group
Universitat Rovira i Virgili

Figure 4.2. SNAP Deployer splash screen
This tool allows web applications to be deployed on a SNAP network.

- **Secure and decentralized web application deployment**. A J2EE application authentication service is already bundled into SNAP. Indeed, restrictions must be made since not everybody has the right to deploy any web application onto the infrastructure. In order to prevent abuses and malicious uses, we have opted for centralizing the deployment phase, so that **only the administrator** of the network can *install*, *deploy*, and *monitor* applications on SNAP. As a consequence, the anarchy generally found in p2p networks is controlled, and users are prevented from unauthorized manipulation of web applications.

  Obviously, before deploying an application, the administrator's signature must be present. This is achieved by using the **SNAPDeployer** tool, which manages the process of embedding this signature in any *web application archive (WAR)*. If the signature is unavailable, the file is not deployed, so users are unable to use it since it will not be bound in the application's naming space. We use public/private key cryptography and certificates to implement this functionality.



SNAP Web Application Signer and Deployer Wizard

Welcome to SNAP's Web Application Signer and Deployer Wizard.

This wizard allows you to sign deploy all of the received Web Application archives which are intended to be deployed on the Structured overlay Network Application Platform (SNAP).

You will be prompted for you own pair of public/private keys for signing WAR files. Moreover, you will be asked to supply the web application archive to be signed and deployed into SNAP.

If you do not have a pair of usable public/private keys, a new pair will automatically be generated for you.

Figure 4.3. SNAP Deployer Wizard

Deployed web application data is not stored solely on one unique node. These signed applications are replicated among different network nodes (we use Dermi's own DHT service (Bunshin), in order to guarantee fault tolerance and avoid bottlenecks.

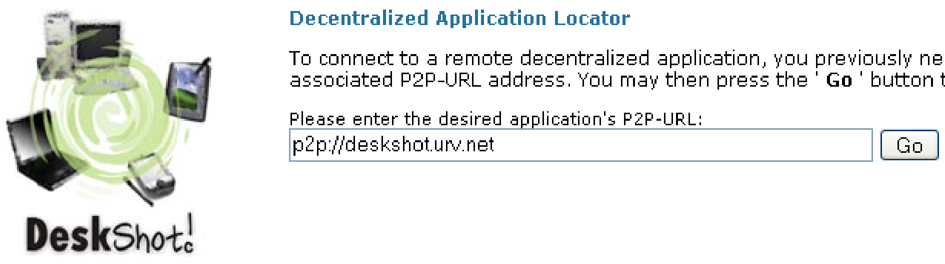
- **Web application activation on demand**. p2p networks are very dynamic environments and, as such, web applications can be activated on demand when required. In SNAP's scenario, every time an application is requested by any client, it is automatically downloaded from the p2p network, deployed and instantiated on the local webserver. If other extra-services are required (eg. database), these are also activated on demand. All accesses are local to that lightweight server instance.

  However, this only happens whenever no active instances of that web application are already running on the network (we use p2pCM's *component instantiation* feature). In this case, the client is automatically redirected to the **closest** webserver which hosts that active application. This **edge service** allows multiple replicas of an application to be currently running on multiple network nodes. Changes are transparently and accordingly replicated along all replicas by our p2pCM component model.

The transparent services provided to web application developers include the following (see Figure 4.1):

- **Uniform web application location (Location Service)**. Accessing SNAP's web applications requires special *Uniform Resource Identifier (URI)* style addresses (eg: *p2p://deskshot.urv.net*). These locators uniformize the address space, as well as the application's access independently of its real location (IP address), and the service provider. SNAP internally redirects requests to these applications to the real IP addresses (which may change over time if nodes fail, new ones join, etc). Therefore, these p2p locators do not have real location information embedded in them.

In this case, whenever we want to access a SNAP application, we introduce its *p2p://* URL in the *decentralized application locator* (Figure 4.4) textfield located in SNAP's home. After that, we are automatically redirected to an already active instance, or if none exists, it is activated locally.



To connect to a remote decentralized application, you previously need to enter its associated P2P-URL address. You may then press the ' **Go** ' button to open it.

Please enter the desired application's P2P-URL:
p2p://deskshot.urv.net

Figure 4.4. SNAP's Decentralized Application Locator

- **Adaptation and load balancing (Clustering Service)**. SNAP optimizes network resources and thus adapts to application's load increments. This way, the minimum number of nodes where the application can be replicated can be specified. A *cluster* of J2EE web applications is then automatically formed, transparently to both users and developers. All requests are distributed accordingly depending on their physical origin, by means of Dermi's *anycall* abstraction.

  Internally, this J2EE web application cluster is managed as a replicated p2pCM component, in SNAP. Periodic checks are performed between group members to assure other members' liveness. Once a deficit has been detected between the desired and the expected number of cluster nodes, a recovery algorithm is executed, thus rebalancing the cluster and activating new application instances on neighbor nodes. Application instances can be passivated as well, whenever they have not been used for a specified time (*dbPassivationThreshold* parameter in Figure 4.5), checkpointing the global state to the DHT, so as to free up resources on the node. This adaptation algorithm will then be further refined so that it can dynamically self-adapt to high demand peaks, thus providing more robust and adaptive load balancing. We are currently using self adjusting techniques to research new ways of providing such dynamic adaptation.

- **Persistence Service**. Our infrastructure basically provides two types of persistence modes, depending on the application's needs.

  - *Replicated file warehouse* is used to store the object's or component's state in the DHT structure. In this mode, data is automatically replicated among various nodes thus guaranteeing its availability in case of node failures. We are currently using a simple version control system, but this does not ensure that there will be no consistency issues with concurrent updates. We are studying alternative ways of solving this *multiple-writers* problem. One possible solution could be to use a log-based data structure. In this way, updated data is only appended and never modified. Therefore all changes can be committed. Naturally, there may still be conflicts, but at least no data is lost, and it is easy to determine any conflicts that have occured. Normally, this is a good persistence mode when simple state information needs to be stored, and there is no to perform complex queries on it.

- *Replication and clustering of relational databases*. Whenever our applications require persistence on relational databases, SNAP provides transparent replication between the different cluster members of the web application. This **edge service** allows dynamic database activation on demand. Additional clustered databases will be activated immediately to provide load balancing. Moreover, a timeout period can be specified (whenever the database receives no requests), after which the database is deactivated so as to free up resources in the node. Following the p2p philosophy, we have consequently opted for a very lightweight database engine, HSQLDB [26], slightly modified to fit our needs (it instantiates a group membership management p2pCM component, along with a data replication p2pCM component, to take care of fault tolerance and data dissemination along all other database instances belonging to the same application). In case all the application's database instances have been deactivated, the state is checkpointed to the *replicated warehouse* (the DHT), thus guaranteeing that future database activations recover state correctly from the last checkpoint.

  In order to maintain state and consistency throughout all database replicas, a p2pCM component manages group membership, along with another data replication component, which propagates state changes. Combined with these, a distributed interceptor object (traversed every time an update is sent, and conveniently replicated for fault tolerance) provides event ordering. This is to avoid consistency failures in case of concurrent updates. This alternative is adequate if we consider that this mode of operation only requires a few nodes for data storage (no more than 5 to 10 is enough). Both the group membership component, which keeps track of the liveness of the cluster members, and the distributed interceptor provide lightweight ordering, which is adequate for most applications. Nevertheless, we are studying the use of other mechanisms to provide total or causal ordering which need to be further researched if strong consistency checks are to be provided.

- **p2p Application Programming Interface (p2p-API) Service**. Since SNAP resides on top of the p2pCM component model, and the Dermi object middleware, it provides a natural gateway to these frameworks' APIs. Therefore, we can easily create components or lower-level objects which use their services of distributed interception, application level multicast, or their p2p invocation abstractions (*anycall, manycall, multicall, directcall, hoppedcall,* etc.), thus enriching application developers with new possibilities.

- **Web Service Layer**. Service Oriented Architectures (SOA) are an emerging paradigm these days. We believe that this concept can also be applied in wide-area p2p environments. In fact SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. In this case, a service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners.

  We have added web service capabilities to SNAP in order to try to overcome one of its main limitations: its Java-centric approach. Therefore, with this layer we enable the deployment of web services in a p2p network, thus allowing interoperability with other technologies. For more information about the services offered by this layer, please refer to [96].

4.5 SNAP's Use Case Scenario

In this section, we describe how SNAP's services are used in the life cycle of a web application which is to be deployed on SNAP. This process involves some required phases and other optional ones. However, our intention is to clarify the interaction points between the SNAP infrastructure and the application we wish to deploy.

First, the J2EE web application is **adapted** to run on SNAP; next, it is **deployed** to make it available worldwide. Whenever the application is to be accessed, it has to be **located** first, and **activated** dynamically. Application data is accessed and **persisted** transparently by SNAP's persistence service, and automatically **load balanced** for concurrent accesses. Finally, **fault tolerance** is also provided by SNAP when nodes where the application is instantiated suddenly stop working.

Since SNAP allows lightweight J2EE applications to be easily adapted, the first step is to prepare the application so that it can be deployed on SNAP.

The **Adaptation phase** is accomplished through an easy, automated process of signing and packaging (via the *SNAP Deployer* tool), which also creates a new XML deployment descriptor file (*snap-war.xml*) (see Figure 4.5). The administrator can automatically adapt static web application contents in the SNAP network too, with no need to change a line of code. When dealing with applications which work with relational databases, developers can choose to use direct JDBC connections (thus making slight changes in the way the JDBC connection is obtained (calling SNAP's *Application.getConnection()* method), or using *DataSources* (where they only need to update the DataSource's JNDI name in the application's *web.xml* file), without touching a line of code. The result of this phase is an administrator signed web application archive (.WAR), which is ready to be deployed on the SNAP network.

The **Deployment phase** consists of uploading the signed web application onto SNAP. This deployment step is different than in traditional web application servers. To prevent abuses and malicious uses, we do not allow everybody to deploy any web application onto the infrastructure. For this reason we opted to centralize the deployment phase and, therefore, only the administrator of the network can *install*, *deploy*, and *monitor* applications on SNAP.

Once the web application has been successfully deployed onto SNAP, we are ready to access it. To do so, we use a standard web browser client, and redirect to our local machine (or any machine we know SNAP is installed in), where SNAP's home is presented. There we can insert a SNAP application locator (p2p-URL - eg: *p2p://deskshot.urv.net*), which we will use to access the p2p application. In the **Uniform web application location phase**, SNAP internally redirects requests to its applications to the real IP addresses (which may change over time if nodes fail, new ones join, etc). To achieve this redirection, the DHT is queried about the p2p locator's meta-information. This meta-information has previously been introduced in the DHT in the deployment phase. Therefore, the signed WAR file is retrieved and checked against the administrator's public key for a match. Now that the web application package has been located, it is time to activate the web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>
    SNAP Web-Application descriptor
  </comment>
  <entry key="appName">SNAP TestSuite Application</entry>
  <entry key="appContext">snaptestsuite</entry>
  <entry key="appp2pUrl">p2p://snap.deim.urv.cat</entry>
  <entry key="persistence">database</entry>
  <entry key="clustering">4</entry>
  <entry key="dataSource">jdbc/SNAPDs</entry>
  <entry key="dbInitialPort">9999</entry>
  <entry key="dbDataPath">/WEB-INF/data/</entry>
  <entry key="dbPassivationThreshold">10</entry>
</properties>
```

Figure 4.5. Sample deployment descriptor file for any SNAP Application (*snap-war.xml*)

Peer-to-peer networks are very dynamic environments and as such SNAP allows web applications to be activated on demand. Every time an application is requested by any client, it is automatically downloaded from the p2p network (the DHT), deployed and instantiated on the local webserver. If other extra-services are required (e.g. database), these are also activated on demand. All accesses are therefore local to the lightweight webserver instance. It is important to note that this process only takes place when no available active instances of that web application are found to be already running on the network. This **Web Application Activation-on-Demand phase** is more complex when there are active web application instances already running on the network.

The idea is that all instances of a specific web application form a multicast group, and we use Dermi's *anycall* feature, which sends an *anycast* message to the closest member of the group. This way, when there are active web application instances, the client is automatically redirected to the closest webserver which hosts that active application replica, in case it is not too overwhelmed, which causes local activation. This edge service allows multiple replicas of an application to be currently running on multiple network nodes. An application instance is modelled as a p2pCM component and, therefore, all state changes that occur within that instance, are accordingly replicated among all others.

At the current stage, our SNAP web application has been successfully activated (either locally or accessing the closest instance), and we start interacting with it. At some point, it requires access to the **Persistence service**. If the application requires access to a relational database, SNAP's persistence type is set to that of *replication and clustering of relational databases* at deployment time. In this operation mode, SNAP provides transparent data replication between the different instances of the web application.

If developers wish to take advantage of the DHT wide-area persistence, the *replicated file warehouse* mode can be used. Normally, this is a good persistence mode when simple state information needs to be stored, and there is no need to perform complex queries on it.

Over time, our web application may start receiving massive invocations. In order to allow our application to adapt to load increments, SNAP provides an **Adaptation and Load Balancing Service**. We can therefore specify a minimum number of nodes where the application can be replicated (the *clustering* attribute in Figure 4.5). A *cluster* of J2EE web applications is then automatically formed, transparent to both users and developers. All requests are distributed according to their physical origin, by means of Dermi's *anycall* abstraction.

Finally, and with reference to the **Fault Tolerance Service** we can imagine that when we access a web application, nodes may fail. Naturally, if we are redirected by SNAP to an active instance, and it stops working, we shall receive the typical "404 Page Not Found" error in our browser. In this case, we can try to access the application again using SNAP's redirection service (the p2p-URL locator) through SNAP's home page, and be transparently redirected to another live instance – the closest one, or a new local one if the closest one is too overwhelmed. This step, which works with no modification on the client web browser, must be taken manually. Nevertheless, more *automatic* approaches require the client browser to be modified (via a *plug-in*) so that it can detect such failures, and automatically call SNAP's system to redirect to another active instance.

SNAP also has a simple **Application Management API** in which the administrator can monitor applications via reflection (see which nodes host application instances, start/stop applications, monitor database use, etc). Nevertheless, these services need to be further researched and expanded.

For further information on how applications can be adapted, deployed and executed from within SNAP, please refer to Annex C at the end of this thesis.

4.6 Empirical Evaluation

We have performed extensive tests on our architecture in order to validate all the services. As with Dermi, we deployed SNAP in PlanetLab [69]. The PlanetLab testbed is the perfect environment to simulate real Internet conditions and therefore check SNAP's performance in a wide-area setting.

We concentrated on performing general failover and performance tests. We chose more than 100 nodes from distinct and varied geographical locations, and performed the tests at different times of the day to minimize network latency and CPU load effects as much as possible.

The initial setting was a bunch of empty SNAP nodes, in which an administrator-signed web application is deployed. After successful deployment, this application is called from one random node. The static clustering factor is equal to 4, so after initial activation, 4 replicas are spawned among the node's neighbours. The Web application passivation time was set to 10 minutes, thus guaranteeing non unloading interferences during the tests.

The deployed application is a simple test which uses the replicated database (inserting some data, and selecting it back), performs a few calls to SNAP's introspection API (to query for application's replica addresses), and counts the time elapsed since the starting request.

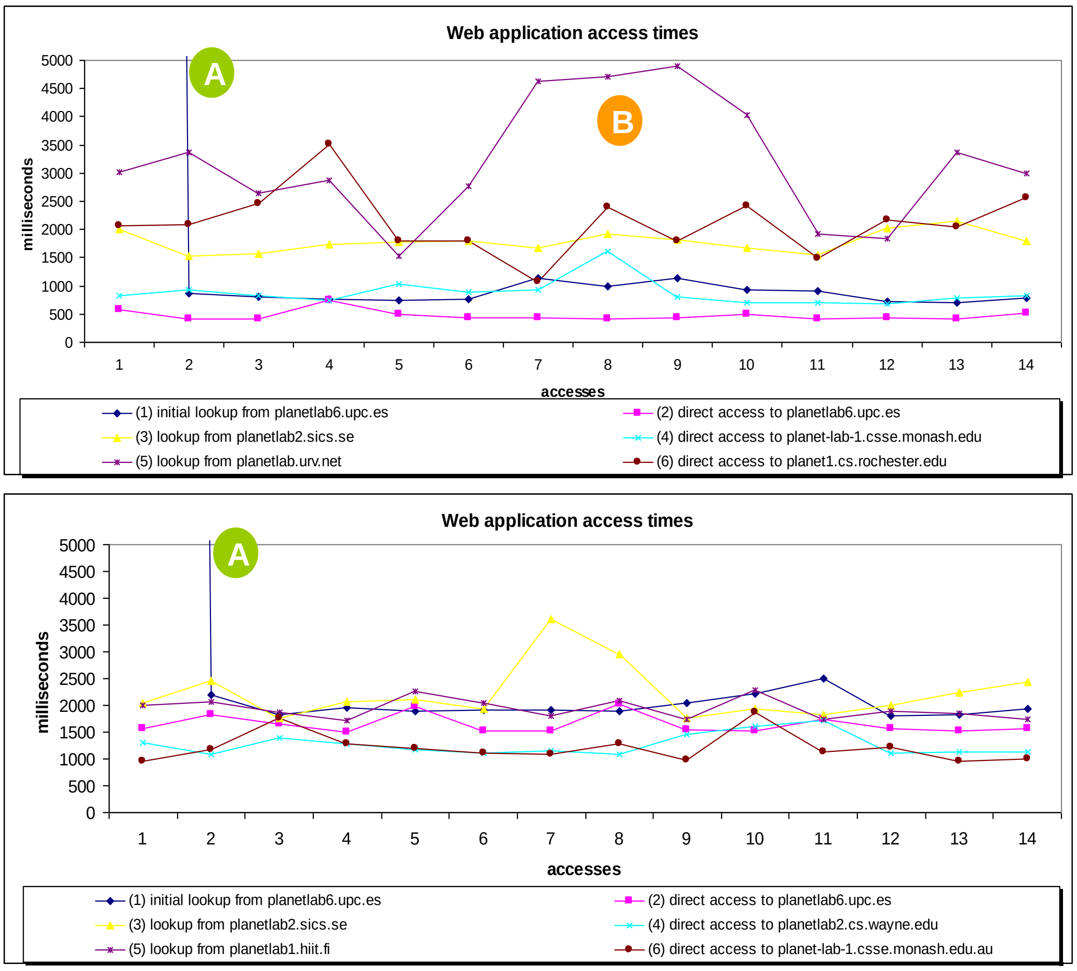


Figure 4.6. SNAP Web application access times
These are two different tests which mainly show web application access times when doing lookups and direct accesses. Note that initial web application invocation requires much more time (hence the initial peak), since it needs to activate the required components as well as the database.

Our aim with this experiment was to **quantify the overhead imposed in application lookups**, rather than with direct application access. Naturally, we also wanted to observe SNAP's behaviour when dealing with unexpected node failures.

Before going deeper into the results, we should consider that nowadays PlanetLab is a rather overwhelmed network, which runs constant multiple experiments on most of its nodes. This is why access times appear to be relatively high, even when web applications are accessed directly (served by Jetty itself and without SNAP interferences).

Results of the experiment are shown in Figure 4.6. This figure consists of two charts, which show different data for the same kind of test, but with different physical nodes. The idea was to access the *snaptestsuite* application from different nodes, using lookups and direct accesses. Results must be read sequentially, and following the specified series order (1)→(2)→(3), and so on. Since both charts correspond to the same experiment, but with different node configurations, we shall only comment on the first one, since the second graph's interpretation is basically the same.

One thing that both charts have in common is that web application start up for the first time tends to be relatively high (varying from 102,000 to 130,000 ms), since components and database must be initialized. This explains the *initial peak* in series (1) indicated by **A**. Note that in the future we will fine tune this initial phase in order to reduce startup time (we could vary timeout periods, and reduce retransmission delays). Note that this only happens **in the application startup phase** and, of course, subsequent calls to the same web application incur much lower access times.

Subsequent access times in series (1) (from the first chart) correspond to looking up the web application locally (entering *p2p://snaptestsuite.urv.net* on SNAP's main browser window). The application was started on *planetlab6.upc.es*. By looking up the application, SNAP redirects us to the closest active instance, which in this case is the one running on the local node. This behaviour incurs a slight overhead, since a DHT lookup as well as an anycall is done. Mean access time in this case is 857 ms. This can be observed when looking at series (2), which called the same application on the same node, but performed a direct call (by entering *http://planetlab6.upc.es/snaptestsuite* on our web browser). Mean access times in this case are 488 ms.

The next test (shown in series (3)) involves shutting down SNAP on *planetlab6.upc.es*. Naturally, it is now impossible to directly access the application via the web browser on that node, because it is down, and therefore accesses to the application fail. However, if we access it by looking up from another SNAP node (again entering *p2p://snaptestsuite.urv.net* on SNAP's main page – trying it from *planetlab2.sics.se*), we are redirected to another active replica (in this case, *planet-lab-1.csse.monash.edu*). We again perform continuous lookups, which redirect us to the closest replica (it is usually *planet-lab-1.csse.monash.edu* or *planetlab2.cs.wayne.edu*, or *planetlab1.cs.wayne.edu*, or even *planetlab2.cs.unc.edu*). This shows us that the underlying p2p routing substrate (in our case FreePastry [40] from Rice University) performs Proximity Neighbour Selection (PNS) [107], which continuously updates neighbours according to their latency. Sometimes, access times vary slightly due to these node changes, as indicated by **B**, on series (5). Mean invocation access time for series (3) was approximately 1,790 ms. When accessing *planet-lab-1.csse.monash.edu* directly (which was the node that we were redirected to most often, as shown in series (4)), mean invocation time was 880 ms.

These results are interesting. Notice that **users will not normally perform continuous lookups to work with an application**. Using this service, they will be automatically redirected to an active replica, and work with it, always **in direct access times** (eg: 880 ms, as seen before). However, when the server or the application **stops working, they can use another SNAP node, and perform an application lookup**, which will redirect them to another consistent instance in an **acceptable time** (eg: 1,790 ms, as seen before). This shows not only that SNAP's failover mechanisms work, but also that if nodes fail, we are transparently redirected to the closest copy, which answers in a relatively short time, and subsequent accesses are performed without overhead.

The last tests involved shutting down SNAP on *planet-lab-1.csse.monash.edu*, and moving on to *planetlab.urv.net* to perform *snaptestsuite*'s lookups (series (5)). The average access time was 3,185 ms (*planet1.cs.rochester.edu* mainly answered), whereas direct access times (series (6)) to this node were about 2,143 ms.

As nodes holding applications fail, new ones are populated with active copies of the application. The total number of replicas stays at around four all the time, thus guaranteeing failover in case of abrupt or graceful node failures.

Using the PlanetLab testbed, we verified that SNAP does not impose an excessive access time overhead (the mean normalized incurred overhead is **1.636** (average lookup time divided by average access time), and only when doing '*p2p://...*'-style lookups, which does not happen most of the time (only when an application needs to be accessed).

It is important to highlight that application location (via '*p2p://…*') produces an overhead due to DHT lookups, and *anycalls*. However, once you are connected to the application, performance is good, since access to that local webserver is direct. As a consequence, this demonstrates that SNAP web applications are scalable.

In conclusion, we have proven that the failover mechanism gives good results because of the inherent network locality exploited by SNAP.

4.7 Prospective Uses of SNAP

SNAP's main aim is to facilitate the migration of typical, lightweight J2EE applications from traditional client-server models to a wide-area scalable solution. Even though Dermi and p2pCM primitives can be used by SNAP application developers, a J2EE application can be deployed with minimal descriptor changes and without touching a single line of code. Therefore, in this section we outline SNAP's future prospective uses.

One of the most important features of SNAP, which has already been highlighted, is that it is easy to use. The idea is to facilitate as much as possible the transitioning process of any client-server based J2EE application to a SNAP application. Therefore, as we have described, already existing J2EE applications can **be easily ported to SNAP** through an easy and automated process of signing and packaging (via the *SNAPDeployer* tool), which also creates a new XML deployment descriptor file (*snap-war.xml*). The administrator can easily deploy static web application contents in the SNAP network too, without needing to change a line of code.

When dealing with applications which work with relational databases, we have also tried to make the transition to SNAP as transparent as possible. Developers can choose to use direct JDBC connections (thus making slight changes in the way the JDBC connection is obtained) or *DataSources* (where they only have to update the DataSource's JNDI name in the application's *web.xml* file), without touching a line of code.

The idea is to make it easy for developers to use SNAP. In fact, unless they wish to access native replicated file warehouse features, or p2pCM components, already existing J2EE applications seamlessly adapt to SNAP with an automatic procedure of packaging, signature, and deployment.

One interesting application which can be easily built with SNAP is the decentralization of the **Bittorrent/Suprnova** p2p tracker distribution system. This p2p file sharing system, described in [102], distributes file links in the form of small indexes stored on a central web application. The fake-file problem can be solved with only 20 moderators combined with numerous other volunteers. However, system availability is hampered by the global nature of client/server architecture: when the central web server is overwhelmed or down, no tracker distribution can occur. If this web application is implemented using SNAP, load balancing and failover can occur transparently, thus allowing seamless access to these files all the time, and solving any bottleneck in Bittorrent's tracker distribution.

We foresee other interesting prospective applications that could benefit from these services in the near future. Such application-domain examples include:

- **Enterprise wide-area collaboration**. This application domain is currently one of industry's main areas of interest. Groove's [24] acquisition by Microsoft Corporation shows that this interest is growing. When wide-area applications need to be developed for this area, it is not easy to use existing frameworks to adapt already built client-server applications. In SNAP, there is no painful transition, since J2EE compatible applications can easily be deployed without any change.

  Moreover, when all of the p2p network's potential needs to be exploited, we can use SNAP's p2p-API service, which enables Dermi's call abstractions (*anycall, multicall, ...*), and p2pCM components to be used.

  Other applications which can be developed in this domain also include the typical file-sharing ones or even instant messaging.

- **Wikis, weblogs, and static webpages**. Another domain which could greatly benefit from SNAP's infrastructure is that of weblogs, wikis or even static webpages. Normally these applications do not have excessive requirements and are relatively lightweight. We could even reuse existing applications and deploy them onto SNAP. Why could this be interesting? From a *failover* point of view, if the central server fails, the application becomes unreachable. Using SNAP, users would be redirected to another active instance, and thus still be able to work with the application. From a *load balancing* point of view, the same node would not always receive the requests: they would be distributed throughout the active application replicas. Finally, from a *network proximity* point of view, the replica which you would be redirected to, would be your closest instance in terms of network latency.

  These applications could also be extended, if necessary, by storing and retrieving relevant information onto the DHT. They could use our DHT's inverted list functions (provided by Bunshin and accessible via Dermi and p2pCM) to store and retrieve keywords, for example. Therefore, scalability is guaranteed, since these keywords are disseminated and replicated throughout the network without having a single point of failure, as found in client-server models.

- **Service Oriented Architecture (SOA) applications**. Web services are an emerging paradigm these days. We believe that this concept can also be applied in wide-area p2p environments. In fact, SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. In this case, a service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners.

  In fact, p2pCM components could easily be extended to become web services. As a consequence, we are currently working on an XML-RPC façade to provide access to these components via HTTP and XML. Future applications will be built on top of these web services, and provide access to a *wide-area service oriented architecture*.

  This research line is part of SNAP's future development and needs to be further investigated.

- **Collaborative Edge Services**. On top of the SNAP edge application infrastructure, and using SNAP's edge services, we plan to construct a set of Collaborative Edge Services that will help to create p2p environments in a decentralized setting. These services are the following:

  - User and Profile Management: Using the replicated persistence service on top of the DHT, we are creating a p2pCM component that securely stores user information. The use of login/password or certificates thus guarantees authenticated access to personal user resources and information. Furthermore, a user will be able to store public keyword information in his/her personal profile, and this information is properly stored in the DHT (as inverted lists) in order to enable efficient queries over those keywords. This profile management service is very important for the p2p Community Service, which we will present later in this section. Finally, the user interface for this p2p User and Profile management service is a Java Web application that must be deployed over a SNAP network.

  - p2p Workspaces: We have constructed a J2EE Web application that also benefits from SNAP's file replication mechanisms to upload or retrieve contents from shared folders. The owner of a workspace can thus invite other members and permit joint collaboration over the shared knowledge. In order to access the p2p Workspace service, the user must acquire an identity in the SNAP network by means of the previous User and Profile Management service. At this stage, we only permit files to be shared in shared workspaces, but we also plan to define an extensible artifact contract like the Groove's Tools extensions.

  - p2p Domain Name System: Inside a SNAP network, we can locate resources using p2p URLs like: p2p://www.urv.net or p2p://carles.pairot@urv.net. The assignment of p2p URLs to deployed applications and resources is the responsibility of the SNAP network administrator. Using a Web interface (p2p Domain Name System), the administrator can thus assign p2p URLs and domains to requested applications of users inside the network. Note that SNAP is a controlled and secure p2p environment that can be managed by a company or online community. Only the private key of the SNAP's network administrator permits access to this p2p Domain Name System.

- Bidirectional Hyperlink system: This application allows users to create p2p links (p2p://objectweb.org) to resources and applications found in the SNAP network. The relevance of this service is that p2p links are stored in the DHT persistence system with both source and destination. Furthermore, our APIs show the number of incoming and outgoing links for a URL and also online notification of new established links. Therefore, we can easily find out how many links point to my p2p URL (p2p://carles.pairot@urv.net), and retrieve them to know who connects to me, but also find out in real time if someone creates a link to my p2p resource. This hyper linking system is a valuable resource that contains graph information of the social networks and communities in the overlay.

- Community Service: Finally, this service is our query engine over the information contained in the overlay. Using the information stored in the DHT by the User Profile Service, the p2p Hyperlink system, and the tagged resources (with keywords), the Community Service allows advanced searches of the existing collaboration communities. We can thus search a community using keywords, and find users with similar interests. We can also rank the results of a query using incoming links to that resource as a metric (similar to Google's PageRank [22]). We believe that this service is very important for obtaining introspection information of the network, but also to boost user collaboration and the creation of communities around keywords.

We consider that all these services can help to build p2p collaboration communities that benefit from edge services in the network. An enterprise can cheaply create a professional community with advanced tools for online collaboration. The data and computing resources of the community do not reside in expensive central servers, but instead are replicated over desktop machines or if necessary some dedicated backbone server.

Furthermore, when the J2EE standard is used the environment is open source and extensible, so existing web applications can be deployed without problems. A company can then offer their subscribers free web pages in the p2p network, free blogs or forums, or even mix centralized and edge deployed applications.

Another scenario in which SNAP's can be used is in highly concurrent systems relying on ordered synchronization between multiple users (such as that found in games or highly collaborative systems). By using the *distributed interception* feature in our Dermi object middleware, we can provide transactional support, and allow user synchronization.

We wish to make it clear that SNAP is targeted to the development and deployment of **lightweight applications**. Therefore, for example, it is not targeted to applications relying on large back-end databases, because it would be infeasible to copy and synchronize large databases to the p2p clients.

4.8 Summary

In this chapter, we have presented a proof-of-concept for our prototype wide-area middleware implementation: the SNAP framework. SNAP provides seamless J2EE-compatible applications that are scalable and accessible from a wide-area p2p environment. We have presented SNAP's main features which include **easy adaptation of already existing J2EE applications** to our framework, **secure and decentralized web application deployment**, and finally **transparent benefit from embedded services like persistence, load balancing, fault tolerance, and edge computing**.

It is important to note that when talking about J2EE-compatible applications, we exclude those which deal with Enterprise JavaBeans (EJBs). EJBs are heavyweight components which would not be deployed efficiently due to the heterogeneity of the nodes' computing power, bandwidth, and resources.

SNAP was validated on the PlanetLab testbed, and we proved that our framework's approach is viable and that the system performs acceptably. SNAP can be downloaded at http://snap.objectweb.org, under a LGPL license. Moreover, we are continuing to improve SNAP and include more useful features.

We have taken the first step in preventing unauthorized application tampering by deploying administrator-signed web applications. Naturally, we need to continue investigating further in order to reduce future potential security threats. We believe that in the coming years, structured p2p networks will continue to evolve, and security is a crucial aspect if these technologies are to take off completely.

We are also researching the way to fine tune the clustering algorithm so that it is not as static as it is now. The idea is to dynamically spawn replicas that depend on the application's load. Besides, these replicas can be activated near the overwhelming nodes so as to reduce overall load impact.

In addition, we are researching new forms of web services which can provide a new view of the SOA arena. Since SNAP components can easily be transformed into web services, we can apply new concepts to this field. These web services can be inserted in the DHT and located without depending on a centralized naming service. Moreover, whenever invoking them, we could use SNAP's locality functions to invoke the nearest web service, thus having **locality-aware web services**. New approaches could include fault tolerant or locality-aware web services, which would benefit from the underlying routing substrate's inherent properties.

Finally, we believe that for this kind of wide-area networks to be successful, an essential aspect is users' critical mass. The more potential users are attracted, the more the aggregated services and, as a consequence, the network runs significantly better, with more services, and more added value. SNAP was accepted by the ObjectWeb Middleware Consortium to be one of its projects [http://snap.objectweb.org]. With this acceptance, and its immediate official deployment on the PlanetLab testbed, we believe that SNAP has the potential to attract users, and that it can be a suitable foundation for the future development of scalable enterprise collaboration, and fault tolerant distributed wide-area web applications.

Chapter Five

5 Conclusions and Future Work

5.1 Conclusions

The development of a wide-area middleware platform is a complex task. In this thesis we have analyzed already existing solutions which try to accomplish this objective, and we have stated that none of them elegantly achieves their goal. For a wide-area middleware platform to be successful, we believe it should fulfill a set of stated requirements. In our prototype implementation, we have opted for *structured p2p overlay networks*. However, our ideas are not restricted to this underlying level substrate functionalities. They aim to be generic enough so that they can be applied to any decentralized underlying substrate solution.

- **Scalability**. A Wide-Area Middleware framework must be based on a scalable substrate for efficient message routing. *Structured peer-to-peer overlay networks* provide an efficient and scalable message routing substrate. Therefore, communication between decentralized nodes is efficient and follows a *Key-Based Routing* scheme, in which a specified *key,value* pair is efficiently mapped to a particular node, in the form of a giant distributed hash table (DHT).

- **Fault tolerance and high availability**. These requirements are fulfilled by our middleware proposal, since it provides transparent mechanisms for data access and replication. Any objects or components (and even web applications, when using SNAP), can be automatically replicated and its state transparently maintained by our middleware. This way, once a node housing any object or component becomes unavailable, the *closest one* takes its place, by means of a transparent internal *anycall* invocation.

- **Load balancing**. Our middleware provides two different load balancing mechanisms. One is based on the *distributed interception* feature, and the other one is based on the *anycall* abstraction. Both of them are complementary and may be used in different scenarios. This service is also available to application developers, which may declaratively specify which load balancing scheme they wish to utilize for their objects, components, and applications.

- **Dynamicity**. Heterogeneous and very different types of nodes may constantly join and leave the overlay network. In this case, the overlay network transparently manages node departures and arrivals for us. Once a node joins the network, it acquires *responsibilities* (that is to say, it is given key-pair values to store), and once a node leaves or disconnects abruptly, its replicated values are redistributed among the remaining participating nodes.

- **Use of the resources on the edges of the Internet**. The idea is to make good use of all the scattered and unused resources of any of the peers forming the network. Since it is a peer-to-peer decentralized network, all members are treated as equals and, therefore, they can all contribute resources to their conforming network. As a consequence, storage and network bandwidth is shared between all participants, since they have to hold state and replication data, and also permit messages to be routed (and forwarded) through them.

- **Usability and Programming Abstractions**. Our middleware has been designed with usability in mind. Therefore, our prototype implementation has been specially architected following the ease-of-use directive. Thus, it is very easy to develop new objects, components, and applications on top of Dermi, p2pCM, and SNAP respectively. The use of Java annotations enormously facilitates method *tagging* for specifying desired features and attributes. Moreover, by using dynamic proxies, developers remain unaware of what occurs behind the scenes. Usability comes at its maximum level in the form of SNAP, which merges all the concepts of our middleware, and applies them to J2EE infrastructures. As a consequence, development and/or adoption of new/existing J2EE applications is a very simple matter, for the experienced Java programmer. Moreover, our middleware solution provides developers with a set of programming abstractions which allow for rich distributed application development primitives. We provide remote objects and components, a decentralized naming service, efficient and innovative group communication primitives, etc.

We have demonstrated that our wide-area middleware proposal complies with all due requirements. Other wide-area middleware approaches lack some or many of these requirements, making them difficult to use, since many services are not implicitly provided or are even non-existent. Moreover, our proposal is, to the best of our knowledge, the first to envision a wide-area middleware framework based on a structured peer-to-peer overlay network.

As a result of this set of requirements, we believe that such a middleware platform must be sustained on top of the already presented **wide-area routing substrate**, as well as an **application-level multicast** infrastructure, and a **decentralized persistence service**. The second layer makes efficient one-to-many communication possible, and also enables proximity-aware *anycast/manycast* primitives such as *anycall/manycall* to be used. The third layer makes it possible to store any object/component/application data efficiently, and in a fault tolerant way, throughout the network.

By using this three-layered architecture for our middleware model, we have achieved the following objectives:

- We have defined a generic architecture middleware model made up of these three layers which provides a set of generic common services. This model is generic enough for it to be applied to different software designs.

- We have analyzed the state of the art in each of our middleware proposal's core layers, and by comparing them to already existing wide-area middleware solutions, we have observed that none of them provides wide-area application developers with enough services.

- We have designed and materialized this generic proposed model by means of two complementary middleware approaches: remote objects and distributed reusable components. The remote object layer provides the component layer with the foundations and most important innovative services. This component layer allows the lightweight components to be defined and deployed. These components can later be reused to provide a higher level of abstraction so that wide-area distributed applications can be composed.

    - In the context of our remote object middleware, we have defined a new set of remote object invocation abstractions. Therefore, we provide the traditionally existing object-to-object (one-to-one) remote method invocations, as well as object-to-objects (one-to-many) calls by using a wide area application-level multicast communications bus. If this underlying information bus also provides us with network proximity-aware primitives like *anycast*, we can also provide the *anycall* and *manycall* abstractions. We have also defined a special set of fault tolerant calls, namely *hopped calls*.

    - We have defined a decentralized object location service. This service allows remote objects / components / applications to be located and inserted into our decentralized generic model, by storing data on the persistence layer.

    - We have also outlined a distributed interception service. By means of the underlying information bus, we provide primitives that can easily intercept remote object calls, similar to Aspect Oriented Programming (AOP) techniques. Therefore, invocations to remote objects can be captured, analyzed, transformed, and even discarded.

    - Our remote object middleware also provides wide-area load balancing by using interceptors or the *anycall* abstraction. Both schemes are complementary and target different use cases, providing the load balancing requirement with enough genericity and flexibility.

    - When defining our wide-area component model, we have adopted a decentralized lightweight container model. Distributed components are therefore modelled as remote objects, including a life cycle service, and a decentralized deployment and location service.

- Finally, we have presented a proof of concept implementation which directly benefits from the underlying framework services. This proof of concept is called SNAP, and it consists of a wide-area application deployment service. SNAP provides application developers with fault tolerance, persistence, interoperability via web services, clustering, and other services. This application of our middleware serves to demonstrate its practical viability and its usability.

We have designed a prototype implementation of both the remote object middleware (Dermi), and the reusable component framework (p2pCM). We have performed simulations of our middleware components, and have demonstrated the viability of these prototypes by means of empirical evaluations on top of the PlanetLab testbed. Since the PlanetLab network models the behaviour of the real conditions in the Internet network, the results obtained can be extrapolated to a wide-area changing environment. Therefore, we have demonstrated that our middleware incurs an acceptable overhead and that invocations to Dermi objects and p2pCM components, as well as SNAP applications, are efficient.

We believe that this wide-area middleware proposal is the first step in providing a solution that will ease the complex task of developing wide-area scale distributed applications. Structured peer-to-peer overlay networks are of particular interest because we believe that, in a near future, such networks will start expanding exponentially. There are in fact many examples of such networks that are already working (e.g. eMule Kad and BitTorrent DHT. However, we believe that the future of p2p remains in this research line. Many more services still require further research.

5.2 Future Work

In this research, we have aimed to propose a wide-area middleware framework that is generic enough to be used in any of the decentralization paradigms available. Therefore, we think that our ideas are applicable independently of the underlying infrastructure. We expect this work to be continued in the future, and hope that these ideas can be exploited in different application domains.

In particular, we believe that this thesis opens the way for two lines of future work. The first one is based on our *p2pWeb model*, the main aim of which is to provide service oriented architectures with innovative features. The second one involves the creation of a wide-area autonomic computing infrastructure for supporting self-adjusting applications on a global scale.

5.2.1 The p2pWeb Model

We foresee promising cross-fertilizations of peer-to-peer and Web models in the coming years. Although both models are already influencing each other, the lack of seamless integration between them makes it difficult to achieve constructive synergies.

Therefore, we have proposed the **p2pWeb model**, which provides decentralized solutions for service description, publication, discovery and availability, following the web services standards.

Our p2pWeb model aims to bring all the benefits and unused resources of the edges of the Internet to the mature and standardized world of Web applications and services.

As a consequence of this thesis, research into integrating p2p and the Web has been initiated. Naturally, some problems need to be tackled:

- A new **Service Oriented Architecture (SOA)** needs to be defined that is suitable for the p2pWeb scenario. The idea is to provide seamless access to applications and components through open standards such as **web services**.

- **Web services** must be integrated into a **p2pWeb** network. Web service access mechanisms which benefit from the underlying invocation abstractions introduced in this thesis should be studied. Therefore, web services could benefit from the p2p layer primitives, and *proximity aware web services,* for example, could be provided. In this way, we can achieve interoperability between other heterogeneous platforms and programming languages.

- **Transparent location, load balancing and fault tolerance for p2p web services**. Since we are based on a p2pWeb network, web service location should be decentralized, and fault tolerant: if we wish to invoke a particular service, we should obtain a reference to the appropriate instance, even if there are overwhelmed or failed nodes.

We believe that all the features of our p2pWeb model can be of special interest for the creation of communities, and for the development of future collaborative decentralized applications. With the addition of a service oriented architecture for wide-area access, our platform could benefit from interoperability and the addition of new services which take into account the whole network's inherent properties.

5.2.2 Peer-to-Peer Middleware for providing Autonomic Computing

A concept that is emerging as a very interesting research topic is *Autonomic Computing*. Autonomic Computing was an initiative started by IBM in 2001. Its ultimate aim is to create self-managing computer systems to overcome their rapidly growing complexity and to enable them to grow further.

In any autonomic system, the human operator takes on a new role by not controlling the system directly. Instead, a set of general policies and rules serve as input for the self-management process. For a system to be considered to be *autonomic*, it must be **self-configurable**, meaning that it must allow for automatic configuration of components; **self-healing**, meaning it should provide automatic discovery, and correction of faults; **self-optimizing**, meaning that resources should be automatically monitored and controlled to ensure the optimal functioning; and, finally, **self-protecting**, meaning that it must allow proactive identification and protection from arbitrary attacks.

Designing an autonomic system is a complex task. It is even more complex if a wide-area autonomic computing system is to be designed. To the best of our knowledge, at present there are no wide-area autonomic computing systems, so this is an area for further research.

By linking autonomic computing systems with the concepts presented throughout this thesis, we believe that the p2p middleware services we have been describing could be used in the development of a wide-area autonomic system. This research line, then, is very ambitious since its main aim would be to provide the basis for a wide-area *autonomic infrastructure* within which we could design and deploy autonomic systems, by defining rules and policies. This infrastructure should be able to cope with dynamicity, scalability and reliability. Dynamicity is an important requirement for this system, since it should be adaptable to constant network node joins/leaves, and also allow runtime policy addition and removal.

In software engineering, the programming paradigm of **aspect-oriented programming** (AOP), also called aspect-oriented software development (AOSD), attempts to help programmers separate concerns, or break down a program into different parts that overlap in functionality as little as possible. In particular, AOP focuses on the modularization and encapsulation of crosscutting concerns. We believe AOP techniques can be used to achieve our wide-area autonomic computing infrastructure goal by intercepting existing code and allowing dynamic adaptations.

This research line follows on from the work initiated by this thesis, since our contributions can serve as the core substrate on which a wide-area autonomic computing framework can be developed.

Chapter Six

6 Publications

In this Chapter we outline all the publications related to this thesis. Our publications vary from national and international conference proceedings as well as national and international publications in journals and magazines.

6.1 Dermi Remote Object Middleware

- C. Pairot, P. García, and A. F. Gómez Skarmeta. Wide-Area Middleware: The Need for a New Layer. *Internal Research Report. Departament d'Enginyeria Informàtica i Matemàtiques. DEIM-RR-03-005*. April 2003.
  - In this paper, we focus on event-based systems and describe several of them that have been proposed in recent literature. We analyze their features, and propose a new wide-area middleware layer to solve the shortcomings encountered.

- C. Pairot, P. García, A. F. Gómez Skarmeta, R. Rallo, and R. Mondéjar. DERMI: Middleware para aplicaciones de trabajo en grupo descentralizadas. *Jornadas Técnicas RedIRIS 2003*, Palma de Mallorca, Spain, November 2003. ISSN 1139-207X.
  - In this article, we present our first outline of Dermi, a middleware for the development of decentralized workgroup applications, built on top of a peer-to-peer network modelling a distributed hash table. Dermi provides numerous services, such as synchronous and asynchronous remote method invocations, decentralized object location, object mobility, distributed interception and two new invocation abstractions, namely *anycall* and *manycall*.

- C. Pairot, P. García, and A. F. Gómez Skarmeta. Towards a Peer-to-Peer Object Middleware for Wide-Area Collaborative Application Development. *Workshop Trabajo en Grupo y Aprendizaje Colaborativo: experiencias y perspectivas. X Conferencia de la Asociación Española para la Inteligencia Artificial, CAEPIA 2003*. Donostia, Spain, November 2003.
  - In this paper, we focus on the collaborative perspective of building Dermi applications. We analyze and propose how Dermi's abstractions can be used to build wide-area collaborative services.

- C. Pairot, P. García, and A. F. Gómez Skarmeta. Dermi: A Decentralized Peer-to-Peer Event-Based Object Middleware. *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*. Tokyo, Japan. March 2004, pp. 236 - 243. ISSN: 1063-6927. ISBN: 0-7695-2086-3. Acceptance Rate: 17.7%.
  - In this paper, we present Dermi and describe all its services, which include anycall and manycall abstractions, the distributed interception service, and the decentralized object location service.

- C. Pairot, P. García, and A. F. Gómez Skarmeta. Dermi: A New Distributed Hash Table-based Middleware Framework. *IEEE Internet Computing Magazine*. Vol 8, No. 3, May/June 2004, pp. 74-84. ISSN: 1089-7801.
  - In this article, we further explore Dermi's services, and present simulation results and empirical evaluations which demonstrate the viability of our approach.

- C. Pairot, P. García, A. F. Gómez Skarmeta, and R. Mondéjar. Achieving Load Balancing in Structured Peer-to-Peer Grids. *Lecture Notes in Computer Science (LNCS) Volume 3038. 4th International Conference on Computational Science (ICCS 2004). 1st International Workshop on Active and Programmable Grid Architectures and Components (APGAC 2004)*. Kraków, Poland. June 2004, pp. 98-105. ISSN: 0302-9743. ISBN: 3-540-22116-6.
  - In this paper, we introduce the concept of a structured peer-to-peer grid and present our contribution to this new world by means of Dermi. In addition, we focus on the design and implementation of a load balancing facility by using the functionalities provided by this middleware. We present two different approaches to achieve load balancing in our system: a completely decentralized solution by using the *anycall* abstraction, and a distributed interceptor-based one. Both of them can be used in a wide variety of scenarios, depending on needs.

- C. Pairot, P. García, R. Mondéjar, and A. F. Gómez Skarmeta. Towards a Peer-to-Peer Object Middleware for Wide-Area Collaborative Application Development. *Revista Iberoamericana de Inteligencia Artificial*. Vol. 8, No. 24, Winter 2004, pp. 55-65. ISSN: 1137-3601.
  - In this article, we further extend the collaborative capabilities Dermi offers to the development of wide-area CSCW applications.

- C. Pairot, P. García, A. F. Gómez Skarmeta, and R. Mondéjar. Towards New Load-balancing Schemes for Structured Peer-to-Peer Grids. *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*. Vol. 21, January 2005, pp. 125-133. ISSN: 0167-739X.
  - In this article, we elaborate on the decentralized load balancing mechanisms of Dermi, and validate our approach by means of simulations.

6.2 p2pCM Component-Based Middleware

- C. Pairot, P. García, R. Mondéjar, and A. F. Gómez Skarmeta. p2pCM: A Structured Peer-to-Peer Grid Component Model. *Proceedings of the 5th International Conference on Computational Science. 2nd International Workshop on Active and Programmable Grid Architectures and Components*. Lecture Notes in Computer Science (LNCS), Volume 3516. Atlanta, USA, May 2005. ISSN: 0302-9743. ISBN: 3-540-26044-7.

  - In this paper we present p2pCM, a new distributed component-oriented model aimed to structured peer-to-peer grid environments. Our model makes innovative contributions such as a lightweight distributed container model, an adaptive component activation mechanism, which takes into account network proximity, and a decentralized component location and deployment service.

- C. Pairot, P. García, R. Mondéjar, and A. F. Gómez Skarmeta. Building Wide-Area Collaborative Applications on top of Structured Peer-to-Peer Overlays. *Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2005).* Linköping, Sweden, June 2005, pp. 350-355. ISSN: 1524-4547. ISBN: 0-7695-2362-5. **Workshop's Best Paper and Presentation Award**.
  - In this paper we elaborate on p2pCM. We focus on how the services provided by p2pCM can be used to implement essential computer-supported cooperative work (CSCW) services, such as shared session management, awareness and coordination policies, and show a sample application. We believe that all of the features our component-oriented model provides can be very promising for the development of future wide-area distributed CSCW applications.

6.3 PlanetDR and SNAP

- C. Pairot, P. García, R. Rallo, J. Blat, and A. F. Gómez Skarmeta. The Planet Project: Collaborative Educational Content Repositories on Structured Peer-to-Peer Grids. *Proceedings of the 5th ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005). Second International Workshop on Collaborative and Learning Applications of Grid Technology and Grid Education (CLAG + Grid.edu 2005)*. Cardiff, United Kingdom, May 2005, pp. 35-42. ISBN: 0-7803-9074-1. Acceptance Rate: 50%.
  - In this paper we present the Planet Project. Its main goal is the generation of educational content and wide-area distribution. In this respect, we present a distributed content repository (PlanetDR) which has been built on top of Dermi. PlanetDR follows the IMS Digital Repositories Interoperability standard through an implementation of the eduSource Communication Language (ECL) protocol. PlanetDR has been extended to support a federation mode, which to the best of our knowledge is the first attempt to provide an alliance of content repositories throughout a structured peer-to-peer grid. Moreover, several collaborative tools are presented, which will be integrated in the content's life cycle in order to promote knowledge communities around educational content hierarchies.

- R. Mondéjar, P. García, C. Pairot, and A. F. Gómez Skarmeta. Towards a Decentralized p2pWeb Service Oriented Architecture. *Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2006)*. Manchester, England, June 2006.
  - In this paper, we present the p2pWeb service oriented architecture (SOA). The p2pWeb model offers decentralized solutions for service description, publication, discovery and availability, following the web services standards. The three innovative contributions in p2pWeb SOA are: easy integration of web services into a p2pWeb network, secure and decentralized web services deployment, and transparent location, load balancing and fault-tolerance p2p mechanisms.

- C. Pairot, P. García, and R. Mondéjar. Deploying Wide-Area Applications is a Snap. IEEE Internet Computing Magazine. Vol. 11, No. 2, March/April 2007, pp. 72-79. ISSN: 1089-7801.
    - In this article, we present the Structured overlay Networks Application Platform (Snap): a J2EE-compatible wide-area Web application deployment infrastructure. Due to its structured peer-to-peer overlay network substrate, Snap offers three benefits to wide-area Web application deployment: easy adaptation of existing J2EE applications to a scalable network, a secure and decentralized deployment environment, and transparent embedded services, such as persistence, load balancing, fault tolerance, and edge computing.

Chapter Seven

7 References

[1] ActiveGrid Grid Application Server, available at http://www.activegrid.com. Access Date: 2006-05-30.

[2] AgentCities, available at http://www.agentcities.org. Access Date: 2005-05-05.

[3] Akamai EdgeComputing Service, available at http://www.akamai.com/en/resources/pdf/brochures/Akamai_EdgeComputing_Service_Brochure.pdf. Access Date: 2006-05-30.

[4] Apache HTTP Server Project, available at http://httpd.apache.org/. Access Date: 2006-08-15.

[5] Apache Tomcat, available at http://tomcat.apache.org/. Access Date: 2006-08-15.

[6] Apache XML-RPC Libraries, available at http://ws.apache.org/xmlrpc/. Access Date: 2006-08-15.

[7] Architecture and Telematic Services Research Group at Universitat Rovira i Virgili, available at http://www.etse.urv.cat/recerca/ast/. Access Date: 2006-05-31.

[8] Arecibo Observatory - National Astronomy and Ionosphere Center, available at http://www.naic.edu/. Access Date: 2006-08-15.

[9] AudioGalaxy, available at http://www.audiogalaxy.com/. Access Date: 2006-05-29.

[10] BearShare Gnutella Client, available at http://www.bearshare.com/. Access Date: 2006-08-15.

[11] Berkeley Open Infrastructure for Network Computing (BOINC), available at http://boinc.berkeley.edu/. Access Date: 2006-05-29.

[12] Bittorrent, available at http://www.bittorrent.com/. Access Date: 2006-05-29.

[13] Condor: High Throughput Computing, available at http://www.cs.wisc.edu/condor/. Access Date: 2006-08-25.

[14] DataSynapse: Virtual Application Infrastructure Software, available at http://www.datasynapse.com/. Access Date: 2006-08-25.

[15] eMule Project, available at http://www.emule-project.net/. Access Date: 2006-05-29.

[16] Folding@home Distributed Computing, available at http://folding.stanford.edu/. Access Date: 2006-08-15.

[17] The Foundation for Intelligent Physical Agents, available at http://www.fipa.org/. Access Date: 2006-05-30.

[18] Freenet: The Free Network Project, available at http://freenet.sourceforge.net/. Access Date: 2006-08-15.

[19] The GISP Project, available at http://gisp.jxta.org/. Access Date: 2006-08-15.

[20] Gnucleus - Gnutella client, available at http://www.gnucleus.com/. Access Date: 2006-08-17.

[21] Gnutella.com, available at http://www.gnutella.com/. Access Date: 2006-05-29.

[22] Google Technology, available at http://www.google.com/technology/. Access Date: 2006-08-29.

[23] GRID.ORG's Cancer Research Project, available at http://www.grid.org/projects/cancer/. Access Date: 2005-05-29.

[24] Groove, available at http://www.groove.net/. Access Date: 2006-05-29.

[25] GroupKit, available at http://www.groupkit.org/. Access Date: 2006-05-29.

[26] HSQLDB Database, available at http://hsqldb.org. Access Date: 2006-05-29.

[27] IBM Solutions Grid for Business Partners: Helping IBM Business Partners to Grid-enable applications for the next phase of e-business on demand, available

at http://www-304.ibm.com/jct09002c/isv/marketing/emerging/grid_wp.pdf. Access Date: 2006-08-15.

[28] Jetty WebServer, available at http://jetty.mortbay.org/jetty. Access Date: 2006-05-29.

[29] Jini Extensible Remote Invocation (JERI) Application Programming Interface, available at http://java.sun.com/products/jini/2.1/doc/specs/api/net/jini/jeri/package-summary.html. Access Date: 2006-08-15.

[30] JxtaJeri, available at http://user-wstrange.jini.org/jxtajeri/JxtaJeriProgGuide.html. Access Date: 2004-04-20.

[31] LimeWire, available at http://www.limewire.com. Access Date: 2006-08-15.

[32] Microsoft Corporation's COM: Component Object Model Technologies, available at http://www.microsoft.com/com/default.mspx. Access Date: 2006-05-29.

[33] Mnet, available at http://en.wikipedia.org/wiki/Mnet. Access Date: 2006-08-15.

[34] Napster, available at http://www.napster.com/. Access Date: 2006-05-29.

[35] Object Management Group's CORBA Component Model, available at http://www.omg.org/technology/documents/formal/components.htm. Access Date: 2006-05-29.

[36] Object Management Group's CORBA Website, available at http://www.omg.org/corba. Access Date: 2006-05-29.

[37] Open Grid Services Architecture (OGSA), available at http://www.globus.org/ogsa/. Access Date: 2006-05-30.

[38] The Peer-to-Peer Sockets Project, available at http://p2psockets.jxta.org. Access Date: 2006-08-15.

[39] Project JxTA, available at http://www.jxta.org/. Access Date: 2006-05-29.

[40] Rice University's FreePastry, available at http://freepastry.rice.edu/. Access Date: 2006-05-29.

[41] SETI@home, available at http://setiathome.berkeley.edu/. Access Date: 2006-05-29.

[42] Sharman Networks' KaZaA, available at http://www.kazaa.com/. Access Date: 2006-05-29.

[43] Skype, available at http://www.skype.com. Access Date: 2006-08-15.

[44] SpamWatch: A Peer-to-Peer Spam Filtering System, available at http://www.cs.berkeley.edu/~zf/spamwatch/. Access Date: 2006-08-15.

[45] StreamCast Networks' Morpheus, available at http://morpheus.com/. Access Date: 2006-05-29.

[46] Sun Microsystems' Java Enterprise Edition (Java EE), available at http://java.sun.com/javaee/. Access Date: 2006-05-29.

[47] Sun Microsystems' Java Remote Method Invocation (Java RMI), available at http://java.sun.com/products/jdk/rmi/. Access Date: 2006-05-29.

[48] Sun Microsystems' Jini Network Technology, available at http://www.sun.com/software/jini/. Access Date: 2006-05-29.

[49] Request For Comments (RFC) 1 - Host Software, available at http://www.faqs.org/rfcs/rfc1.html. Access Date: 2006-05-29.

[50] Request For Comments (RFC) 1094 - NFS: Network File System Protocol Specification, available at http://www.ietf.org/rfc/rfc1094.txt. Access Date: 2006-08-16.

[51] Request For Comments (RFC) 1112 - Host Extensions for IP Multicasting, available at http://www.ietf.org/rfc/rfc1112.txt. Access Date: 2006-08-15.

[52] Request For Comments (RFC) 2518 - HTTP Extensions for Distributed Authoring -- Web-based Distributed Authoring and Versioning (WebDAV), available at http://www.ietf.org/rfc/rfc2518.txt. Access Date: 2006-08-15.

[53] Request For Comments (RFC) 2526 - Reserved IPv6 Subnet Anycast Addresses, available at http://www.ietf.org/rfc/rfc2526.txt. Access Date: 2006-08-15.

[54] Microsoft's .NET Framework, available at http://msdn.microsoft.com/netframework/. Access Date: 2006-08-15.

[55] C. Adam and R. Stadler, "Implementation and Evaluation of a Middleware for Self-Organizing Decentralized Web Services," in *IEEE SelfMan 2006*, 2006.

[56] Free Riding on Gnutella, available at http://econ.gsia.cmu.edu/Ecommerce/Gnutella.pdf. Access Date: 2006-10-10.

[57] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless Network File Systems," in *15th Symposium on Operating System Principles*. Copper Mountain Resort, Colorado: ACM Press, 1995, pp. 109-126.

[58] A. Bakker, I. Kuz, M. V. Steen, A. S. Tanenbaum, and P. Verkaik, "Design and Implementation of the Globe Middleware," Report IR-CS-003, Vrije Universiteit 2003.

[59] Peer-to-Peer Requirements on the Open Grid Services Architecture Framework, available at http://www.gridforum.org/documents/GFD.49.pdf. Access Date: 2006-08-25.

[60] F. Baude, D. Caromel, and M. Morel, "From Distributed Objects to Hierarchical Grid Components," *Lecture Notes in Computer Science*, vol. 2519, pp. 1226-1242, 2003.

[61] R. J. Bayardo, A. Crainiceanu, and R. Agrawal, "Peer-to-Peer Sharing of Web Applications," in *12th International World Wide Web Conference*, 2003.

[62] F. Berman, G. Fox, and A. J. G. Hey, *Grid Computing: Making the Global Infrastructure a Reality*: John Wiley & Sons, Ltd, 2003.

[63] G. A. Bolcer, M. Gorlick, and A. S. Hitomi, "Peer-to-Peer Architectures and the Magi Open-Source Infrastructure," Endeavors Technology 2000.

[64] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332-383, 2001.

[65] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "One Ring to Rule Them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks," in *10th ACM SIGOPS European Workshop*, 2002.

[66] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A Large-Scale and Decentralised Application-Level Multicast Infrastructure," *IEEE Journal on Selected Areas in Communications*, 2002.

[67] A. Chien, B. Calder, S. Elbert, and K. Bathia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System," *Journal of Parallel Distributed Computing*, vol. 63, pp. 597-610, 2003.

[68] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A Case for End System Multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, pp. 1456-1471, 2000.

[69] B. Chun, D. Culler, and T. Roscoe, "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *ACM Computer Communication Review*, vol. 33, pp. 3-12, 2003.

[70] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," in *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, 2001.

[71] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a DHT for Low Latency and High Throughput," in *1st. Symposium on Networked Systems Design and Implementation (NSDI 04)*. San Francisco, California, USA, 2003.

[72] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, 2003.

[73] P. Druschel and A. Rowstron, "PAST: A Large Scale, Persistent Peer-to-Peer Storage Utility," in *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Elmau/Oberbayern, Germany, 2001.

[74] R. A. Ferreira, A. Grama, and S. Jagannathan, "Plethora: An Efficient Wide-Area Storage System," in *11th ACM/IEEE/IFIP International Conference on High Performance Computing*, 2004, pp. 252-262.

[75] A. Ferscha and M. Hechinger, "A Light-Weight Component Model for Peer-to-Peer Applications," in *24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*. Hachioji, Japan, 2004, pp. 520-527.

[76] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing*, 2005, pp. 2-13.

[77] M. J. Freedman and D. Mazières, "Sloppy Hashing and Self-Organizing Clusters," in *2nd International Workshop on Peer-to-Peer Systems*, 2003.

[78] K. Fu, F. Kaashoek, and D. Mazières, "Fast and Secure Distributed Read-Only File System," in *4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*. San Diego, USA, 2000.

[79] L. Gong, "Guest Editor's Introduction: Peer-to-Peer Networks in Action," in *IEEE Internet Computing*, vol. 6, 2002, pp. 37-39.

[80] A. Gordon, *Programming COM and COM+*: Prentice Hall, 2000.

[81] M. Hatala, G. Richards, T. Eap, and J. Willms, "The eduSource Communication Language: Implementing an Open Network for Learning Object Repositories

and Services," in *ACM Symposium on Applied Computing*. Nicosia, Cyprus, 2004.

[82] H. t. Hofte, *Working Apart Together: Foundations for Component Groupware*. Enschede, the Netherlands: Telematica Instituut, 1998.

[83] Epidemic Algorithms, available at http://wwwcs.upb.de/cs/ag-madh/WWW/Teaching/2004SS/AlgInternet/Submissions/09-Epidemic-Algorithms.pdf. Access Date: 2006-08-15.

[84] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, pp. 51-81, 1988.

[85] M. A. Jovanovic, F. S. Annexstein, and K. A. Berman, "Scalability Issues in Large Peer-to-Peer Networks: A Case Study of Gnutella," University of Cincinatti Technical Report 2001.

[86] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *29th Annual ACM Symposium on Theory of Computing*, 1997, pp. 654-663.

[87] J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," in *13th Symposium on Operating Systems Principles*. Pacific Grove, California, USA, 1991.

[88] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," in *32nd ACM Symposium on Theory of Computing (STOC 2000)*, 2000, pp. 163-170.

[89] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Y. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," in *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.

[90] M. Lewis and A. Grimshaw, "The Core Legion Object Model," in *5th IEEE International Symposium on High Performance Distributed Computing*, 1996.

[91] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and Replication in Unstructured Peer-to-Peer Networks," in *International Conference on Supercomputing*. New York, USA: ACM Press, 2002, pp. 84-95.

[92] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed Hashing in a Small World," in *4th USENIX Symposium on Internet Technologies and Systems (USITS 2003)*, 2003.

[93] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *1st International Workshop on Peer-to-Peer Systems (IPTPS 2002),* 2002, pp. 53-65.

[94] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-Peer Computing," Technical Report HPL-2002-57. HP Labs 2002.

[95] R. Mondéjar, P. García, and C. Pairot, "Bunshin: DHT para Aplicaciones Distribuidas," in *I Congreso Español de Informática (CEDI 2005)*. Granada, Spain, 2005.

[96] R. Mondéjar, P. García, C. Pairot, and A. F. Gómez-Skarmeta, "Enabling Wide-Area Service Oriented Architecture through the p2pWeb Model," in *15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006)*. Manchester, UK, 2006.

[97] A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*: O'Reilly, 2001.

[98] C. Pairot, P. García, A. F. Gómez-Skarmeta, and R. Mondéjar, "Towards New Load-balancing Schemes for Structured Peer-to-Peer Grids," *Future Generation Computer Systems - The International Journal of Grid Computing: Theory, Methods and Applications*, vol. 21, pp. 125-133, 2005.

[99] C. Pairot, P. García, R. Rallo, J. Blat, and A. F. Gómez-Skarmeta, "The Planet Project: Collaborative Educational Content Repositories on Structured Peer-to-Peer Grids," in *5th ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. Cardiff, UK, 2005, pp. 35-42.

[100] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *22nd International Conference on Distributed Computing Systems (ICDCS 2002)*. Vienna, Austria, 2002, pp. 611-618.

[101] C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," in *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1997)*. Newport, USA, 1997.

[102] J. A. Pouwelse, P. Garbacki, and D. H. J. Epema, "The Bittorrent P2P File-Sharing System: Measurements and Analysis," in *4th International Workshop on Peer-to-Peer Systems*, 2005.

[103] S. Ratnasami, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content Addressable Network," in *ACM SIGCOMM*, 2001.

[104] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling Churn in a DHT," in *USENIX Annual Technical Conference*, 2004.

[105] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasami, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: A Public DHT Service and its Uses," in *ACM SIGCOMM*, 2005.

[106] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, pp. 26-52, 1992.

[107] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[108] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility," in *18th ACM Symposium on Operating Systems Principles*, 2001.

[109] K. Seymour and H. Nakada, "GridRPC: A Remote Procedure Call API for Grid Computing," in *Proceedings of GRID 2002*, 2002.

[110] M. P. Singh, "Peering at Peer-to-Peer Computing," in *IEEE Internet Computing*, vol. 5, 2001, pp. 4-5.

[111] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *ACM SIGCOMM*, 2001, pp. 149-160.

[112] C. Szyperski, *Component Software. Beyond Object-Oriented Programming*: Addison Wesley, 1998.

[113] A. R. Tripathi and T. Noonan, "Design of a Remote Procedure Call System for Object-Oriented Distributed Programming," *Software-Practice and Experience*, vol. 28(1), pp. 23-47, 1998.

[114] M. Van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum, "Locating Objects in Wide-Area Systems," in *IEEE Communications*, 1998, pp. 104-109.

[115] M. Van Steen, P. Homburg, and A. S. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. January-March 1999, pp. 70-78, 1999.

[116] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, 2004.

[117] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," University of California UCB/CSD-01-1141, 2001.

[118] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, "Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination," in *11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001)*, 2001.

Annex A. Dermi's Insights

A.1 Application Programming Interface

In this section, we describe how developers can benefit from Dermi's services. We focus on how to use each of the invocation abstractions, access the decentralized object registry, and implement a distributed interceptor.

Nevertheless, before going further into how developers can make good use of Dermi's infrastructure, we first analyze Dermi's core classes, their interactions, and how they are organized in order to provide all of its services.

A.1.1 Core Classes

Dermi's API is mainly composed by a bunch of classes, whose relationships are shown in Figure A.1. There are other classes as well as the ones we show, but we concentrate on describing the functionalities of the ones we consider to be core classes.

A.1.1.1 RemoteEventListener

The *RemoteEventListener* interface is the base interface to be implemented by all Dermi remote stubs and skeleton objects. These objects are invisible to the developer and provide object remote accessibility by means of dynamic proxy interception. The interface defines the standard contract for any remote event triggered by the wide-area event bus. Therefore, it provides upcalls invoked when any event arrives (*eventArrived()*) or any anycall event arrives (*anycallEventArrived()*).

A.1.1.2 ERemote and EInterceptor

These interfaces are the ones that are inherited whenever any Dermi remote object's public interfaces are implemented. Subsequently, any Dermi remote object must define its public remotely accessible interface, which must extend the *ERemote* interface. On the other hand, for a Dermi object to be able to be intercepted via the **distributed interception** mechanism, it must implement the *EInterceptor* interface. The *ERemote* interface provides standard methods for getting the properties of the remote object (*getERef()*), as well as initializing (*init()*), closing (*close()*), and copying (*copy()*) the object.

The *EInterceptor* interface extends *ERemote* and also provides methods for adding and removing object type compatible interceptors (*addInterceptor()* and *removeInterceptor()*).

A.1.1.3 EventServer

The *EventServer* class provides the base publication functionalities of the event bus layer for Dermi. Its main aim is to provide remote connectivity throughout the underlying network substrate by means of sending notifications. It uses a *Session* object, which is the one that really interacts with the event bus, thus sending and receiving remote events. It can be seen as a wrapper class which is the superclass of all Dermi remote objects (its subclasses are *DermiRemoteObject*, *DermiProxy* and *DermiRemoteInterceptorObject*).

It basically uses two main methods:

- *dispatchEvent()*. This method dispatches an event, extracting all parameters from an input *Hashtable* object. The event is disseminated to all members of the specified multicast group. In fact, it invokes the *publish()* method from the *Session* class.

- *dispatchDirectEvent()*. This method also dispatches an event, extracting all parameters from an input *Hashtable* object. However, the difference between this method and the previous one is that the event is delivered to the specified node. Therefore, this is the base method for dispatching **direct calls**. In fact, it invokes the *publishDirect()* method of the *Session* class.

A.1.1.4 DermiConnection

The *DermiConnection* object encapsulates a connection to all parts of the underlying substrate which Dermi uses to provide all of its functionalities. It wraps the connection to the overlay network routing layer, as well as the event server layer. The decentralized object location and routing layer is managed by the static class called *Registry*. The *DermiConnection* class provides a set of methods which are directly invoked by the *Session* class. The most important are the following:

- *subscribe()*.This method manages the subscription to a specified multicast group. Therefore, all object stubs and skeletons subscribe to their object's associated multicast group in order to stay notified of any events targeted to that object.

- *unsubscribe()*. This is the inverse process. It stops notification about events targeted to the specified multicast group.

- *publish()*. This method dispatches an event to all members of the specified multicast group.

- *publishDirect()*. This method dispatches an event to only one member of a specified multicast group. It is the internal method which models the **direct call** abstraction. Naturally, the destination's *NodeHandle* is required in order to be able to send the event.

- *anycall()*. This method dispatches an event to the sender's closest member in the multicast group. It is the core method for modeling the *anycall* abstraction.

- *addInterceptor()*. This method enables an interceptor to be added to a specified multicast group. This means that before sending an event to the multicast group, this event will sequentially traverse the added interceptors. This interceptor can change the destination event, if it wants.

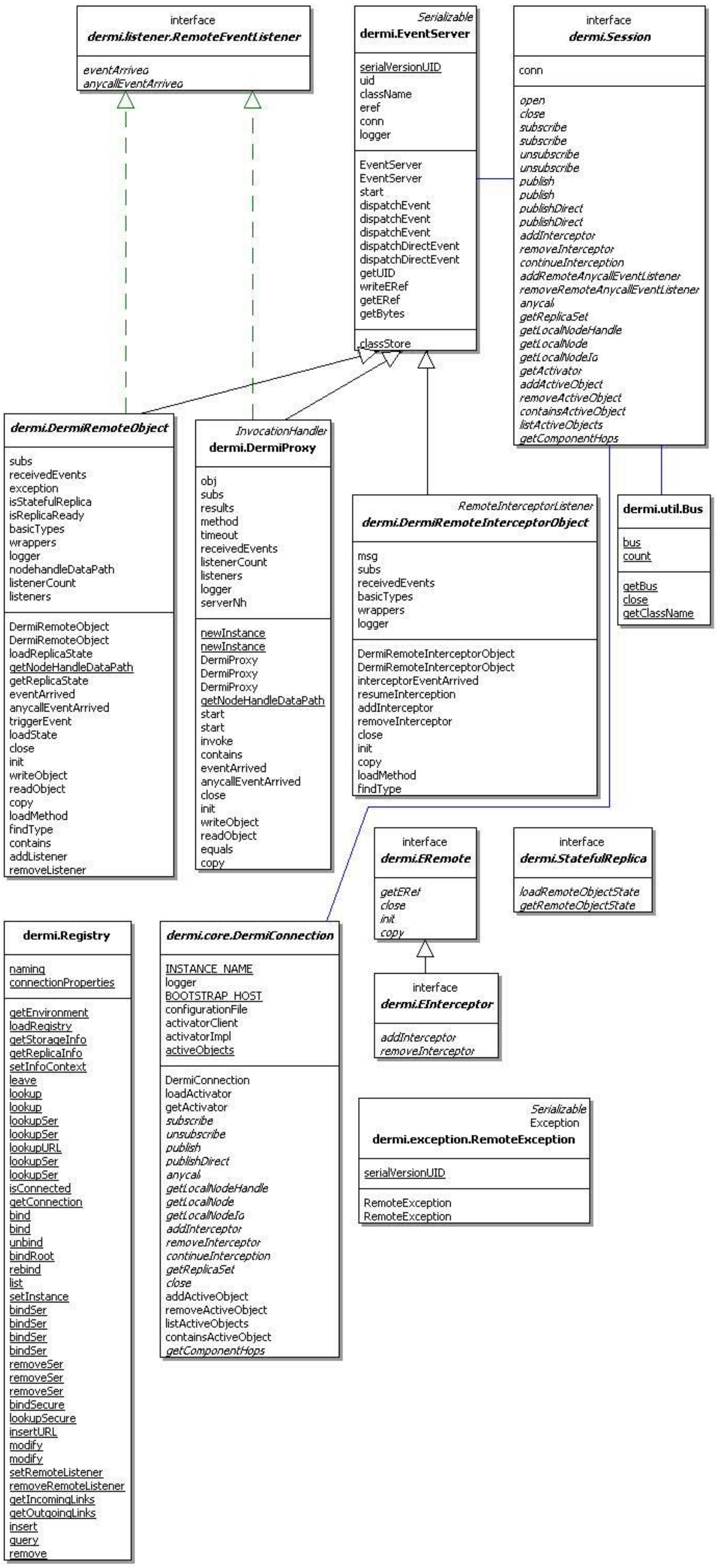


Figure A.1. Dermi's Class Diagram

- *removeInterceptor()*. Used for removing type compatible interceptors from a multicast group.

- *continueInterception()*. Method invoked when an event has traversed an interceptor and must still be intercepted or sent to the multicast group.

Note that *DermiConnection* is in fact an abstract class, whose methods are implemented in the *dermi.core.pastry.DermiConnection* class. This is so because it allows a Dermi connection to be implemented independent of the underlying network subtrate. Therefore, whenever we want to provide Dermi for another KBR layer, we need it to implement the *DermiConnection* interface for that substrate.

A.1.1.5 Session

The *Session* class is the base class used by the dynamic remote stubs and skeletons to access all of Dermi's services. It is basically a wrapper that allows access to all the features exposed by Dermi.

A *Session* is created via a *SessionFactory* class (not shown in the diagram). The idea is to specify the underlying connection class (i.e. *dermi.core.pastry.DermiConnection*) to be used in the *createSession()* method. As a consequence, the connection class is loaded and instantiated via reflection.

A.1.1.6 DermiRemoteObject

The *DermiRemoteObject* class is the core of a Dermi remote object. It extends the *EventServer* class and implements the *RemoteEventListener* interface. Every Dermi remote object implementation **must** extend this class if it is to be able to work as a Dermi object. The inner implementation of a Dermi remote object is found in the following methods:

- *Constructor*. In the constructor, we subscribe this object to its own multicast tree, and we determine whether it is a *hopped* fault tolerant object. If it is, new object replicas are activated throughout the neighbours.

- *loadReplicaState()*. This method is invoked whenever an object's remote state needs to be obtained by any of its replicas. It calls the *getReplicaState()* which calls the *loadRemoteObjectState()* method on the child class in order to anycall to the object group and recover the object's state.

- *eventArrived()*. This is the core method of this class. It is the upcall method invoked every time a remote event is targeted to this object. It first takes care of special remote methods, like *getBytes()* or *getRemoteObjectState()*. If it is not one of them, it processes the method by invoking it locally on this object (the object's child implementation class) using reflection. If everything goes well, it sends the invocation result back to the caller. If an exception occurs, it constructs a remote exception object and sends it back to the caller.

- *anycallEventArrived()*. This method is the same as the one above but deals with anycalls and manycalls. Since obtaining an object's state is considered an *anycall*, it first deals with this special case (*getRemoteObjectState()*). Afterwards, it discovers whether the remote method is an *anycall* or a *manycall* (by trying to invoke the global condition method). Subsequently, it deals with each case's particularities: invoking the local / global condition methods, and dealing with each of its responses (affirmative / negative). Depending on each case, the anycall / manycall will continue routing or will stop at this node. Similarly, if any kind of exception occurs, a remote exception response is sent back to the caller.

- *addListener() / removeListener()*. These methods manage the addition and removal of remote listeners. Therefore, each time an event matching the listener's subscription is received, all of its listeners are invoked the *onEvent()* method.

A.1.1.7 DermiRemoteInterceptorObject

The *DermiRemoteInterceptorObject* class is similar to *DermiRemoteObject*, but for the interceptor objects. These objects are required to implement this class and obtain full **distributed interception** services. Therefore, this class furnishes the basic abstractions so that this service can be provided.

- *Constructor*. The constructor initializes this object, and performs an automatic subscription as interceptor (*addInterceptor()*) in its specified multicast group. As a consequence, all events sent to that group will first traverse this object.

- *interceptorEventArrived()*. Once an interceptor event arrives, this upcall is invoked. The local method specified in the event's contents is invoked and the interception is resumed by calling *resumeInterception()*.

- *resumeInterception()*. This method changes the event's contents (if this is the interception's goal), and calls *Session*'s *continueInterception()* method, which continues the interception process to the next interceptor or delivers the event to the multicast group (if no more interceptors are present).

- *addInterceptor() / removeInterceptor()*. These methods manage subscription / unsubscription from this object as an interceptor in the specified multicast group.

A.1.1.8 Registry

This static class provides the **decentralized object location** facilities of our object middleware. It allows remote object metadata, and serializable object insertion / lookup primitives. It can be seen as Dermi's Naming Service. The *Registry* class contains a reference to a *Naming* object. This *Naming* object is an interface which uniformizes access to any DHT-like underlying layer. Therefore, it defines the basic methods for looking up and inserting data into a DHT. Specific implementations of this interface allow Dermi to work with PAST (*dermi.registry.past.Naming*) or Bunshin (*dermi.registry.bunshin.Naming*).

The most important methods provided by the *Registry* class are the following:

- *lookup()*. This method returns a remote reference (*stub*) for the remote object associated with the specified identifier. This is the usual way of obtaining references to the remote object, and thus of calling its methods.

- *lookupSer()*. This method returns a serializable object that has been previously inserted into the DHT. This object can be of any kind, with the only requirement that it implements Java's *Serializable* interface.

- *bind()*. By using the *bind()* method, we can insert remote object references into the DHT. What is in fact inserted is not the remote object nor its reference, but a set of properties recreating the remote object reference in the near future. Any bound object references can later be restored by calling *lookup()*.

- *bindSer()*. We use this method for inserting any kind of serializable object into the DHT. These objects can later be restored by calling the *lookupSer()* method.

- *unbind()*. This method enables a remote object's reference to be deleted from the DHT.

- *removeSer()*. This method enables a serializable object to be deleted from the DHT.

- *rebind()*. This method overwrites the specified remote object reference if it already exists or it creates a new one if it does not.

- *list()*. This method lists all children nodes from a specified root object. For instance, if the *p2p://deim.etse.urv.cat* root is specified, a list of all bound children nodes is returned (*p2p://deim.etse.urv.cat/deskshot*, *p2p://deim.etse.urv.cat/jounin*, …).

- *bindSecure()*. This method securely binds resources, by storing an encrypted password.

- *lookupSecure()*. This method allows any securely bound object in the DHT to be restored. If the password given is incorrect, the resource is not returned.

A.1.1.9 DermiProxy

The *DermiProxy* class is invisible to the developer because it performs all of the remote object reference communication when invoking remote objects. In earlier versions of Dermi, *stubs* for remote objects had to be pregenerated using the *dermic* tool. This was inherited because of the way in which Java RMI worked. However, by taking advantage of the features of Java version 1.4 onwards, we eliminated the need to pregenerate all object stubs by creating a **dynamic proxy** object which intercepted all object calls and performed all remote communication in such a way that it was totally transparent to the developer.

Therefore, every time we obtain a reference to a remote object, what we really obtain is a reference to a *DermiProxy* object by invoking the *java.lang.reflect.Proxy.newProxyInstance()* reflection method. This object acts as a transparent bridge which intercepts all of its proxied object's calls. As a consequence, every time an object's method is called, the *invoke()* method on the proxy is invoked first. This method analyzes if the method called is a remote one or a local one. In the latter case, it simply calls it. However, if it is a remote method call, it first extracts the method's annotation information to discover if it is a *direct call, multicall, anycall* or *manycall*. Moreover, the synchrony type is also discovered (*synchronous* or *asynchronous*). After the object's granularity and synchrony types have been discovered, an event of the appropriate type is dispatched and targeted to the remote object. This event is received by the remote object, which in turn invokes its method, and dispatches its result back.

Meanwhile, the *DermiProxy* object is waiting for a result (or a timeout), and when the result is received by invoking its *eventArrived()* method, the thread awakes and processes the result value. This value is returned as a result to the caller, without the local method being called, because it has already been remotely called .

A.1.1.10 RemoteException

This is the remote exception superclass in Dermi. It is thrown every time a remote object invocation fails for some reason. There are many subclasses of this exception, all of which show different error conditions: *ConnectionException, NotBoundException, NotSatisfiedException, TimeoutException, RegistryNotLoadedException, TypeNotFoundException, UnmatchedAnycallMethodException, …*

A.1.2 Programming Dermi

When we designed Dermi's API, we wanted to facilitate the learning process for developers as much as possible. Therefore, the coding steps are natural for those developers who are used to programming with Java RMI. The guidelines are summarized below:

A.1.2.1 Defining a Remote Object

To define a new remote object we must first define the remote object's exposed interface (see Figure A.2). This remote interface contains all the methods that the object exposes and which can be remotely called by Dermi. The interface is required to extend Dermi's *ERemote* interface, as well as to be annotated as *@DermiRemoteInterface*. For each method, we specify the kind of method invocation (*Direct Call, Hopped Call, Multicall, ...*), and its synchrony type (*Synchronous* or *Asynchronous*). All remote methods should throw a *RemoteException* so that object clients can be notified about remote failures.

```
package dermi.samples.simple;

import dermi.*;
import dermi.exception.*;
import dermi.annotation.*;

/**
 * Interface to a Dermi Remote Object (Simple Object)
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DermiRemoteInterface
public interface Simple extends ERemote {
  @RemoteMethod (granularity = Granularity.MULTICALL, type =
                 SynchronyType.SYNCHRONOUS)
  public void setAge (String age) throws RemoteException;

  @RemoteMethod (granularity = Granularity.MULTICALL, type =
                 SynchronyType.SYNCHRONOUS)
  public String getAge() throws RemoteException;

  @RemoteMethod (granularity = Granularity.MULTICALL, type =
                 SynchronyType.SYNCHRONOUS)
  public String merge (Integer x, String y, Integer z)
                                          throws RemoteException;
}
```

Figure A.2. Interface of a Dermi Remote Object
All the methods exposed by the remote object are defined in its interface. Java 1.5's annotations are used to specify that this is a Dermi interface (@DermiRemoteInterface), and for each method, we specify its call type (Granularity.MULTICALL) and its synchrony type (SynchronyType.SYNCHRONOUS).

Secondly, we must implement the remote object's methods. As we can observe in Figure A.3, the implementation only requires that Dermi's *DermiRemoteObject* class (which manages the underlying remote communication between objects) be extended, and obviously that the object's interface defined above be implemented. Two constructors for object initialization are also required by Dermi.

```
package dermi.samples.simple;

import dermi.*;
import dermi.exception.RemoteException;

import java.util.*;

/**
 * Implementation of Simple object's methods
 * The implementation completely hides remote state propagation
 * This will be taken into account transparently by the object's skeleton
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
public class SimpleImpl extends DermiRemoteObject implements Simple {

  String age = "12";

  // Dermi constructor: REQUIRED
  public SimpleImpl() {
  }
  // Dermi constructor: REQUIRED
  public SimpleImpl (Properties env) throws RemoteException {
    super (env);
    System.out.println ("[Simple] object created.");
  }

  // Remote methods implementation
  public void setAge (String age) throws RemoteException {
    System.out.println ("[setAge] called.");
    this.age = age;
  }

  public String getAge() throws RemoteException {
    System.out.println ("[getAge] called.");
    return age;
  }

  public String merge (Integer x, String y, Integer z)
                                                  throws RemoteException {
    System.out.println ("[merge] called.");
    return age + " + " + y + " + " + x + " + " + z;
  }
}
```

Figure A.3. Implementation of a Dermi Remote Object
The DermiRemoteObject class needs to be extended, the object's interface implemented, and two constructors for object initialization created.

A.1.2.2 Defining *anycall* methods

Developers can easily mark any remote object's methods as *anycalls*, by following the same rules that are used to define any Dermi object: using annotations. To mark a method as an anycall procedure, we must mark it with the ***Granularity.ANYCALL*** tag, and its condition method as ***Granularity.ANYCALL_CONDITION*.** In our example, the object that returns the data unit **(getDataUnit)** will be called if and only if the condition method (**getDataUnitCondition**) returns true, as specified in Figure A.4. Otherwise, the message is routed to another group member.

Figure A.5 shows a class diagram for a sample Dermi remote object implementation using the *anycall* abstraction. Notice how it inherits and implements the proper Dermi classes / interfaces. Both *SetiClient* and *SetiServer* classes use the remote object.

```
/**
 * This is the interface for the Seti anycall demo object
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DermiRemoteInterface
public interface Seti extends ERemote {

  // Method signature for the anycall method pair must be the same,
  // except for the return result, which needs to be boolean for the
  // condition method

  /**
   * This method returns data unit from one of the client's nearest server
   * @param system String Example parameter
```

```
     * @throws RemoteException If something goes wrong ;-)
     * @return String Data unit returned
     */
    @RemoteMethod (granularity = Granularity.ANYCALL)
    public String getDataUnit (String system) throws RemoteException;

    /**
     * This method is automatically called by the skeleton to check whether
     * the condition can be satisfied for each server
     * @param system String Example parameter
     * @throws RemoteException If something goes wrong ;-)
     * @return boolean true if condition satisfied (the server has
     *  available data units)
     */
    @RemoteMethod (granularity = Granularity.ANYCALL_CONDITION)
    public boolean getDataUnitCondition (String system)
                                                 throws RemoteException;
}
```

Figure A.4. Definition of anycall methods in a remote object's interface

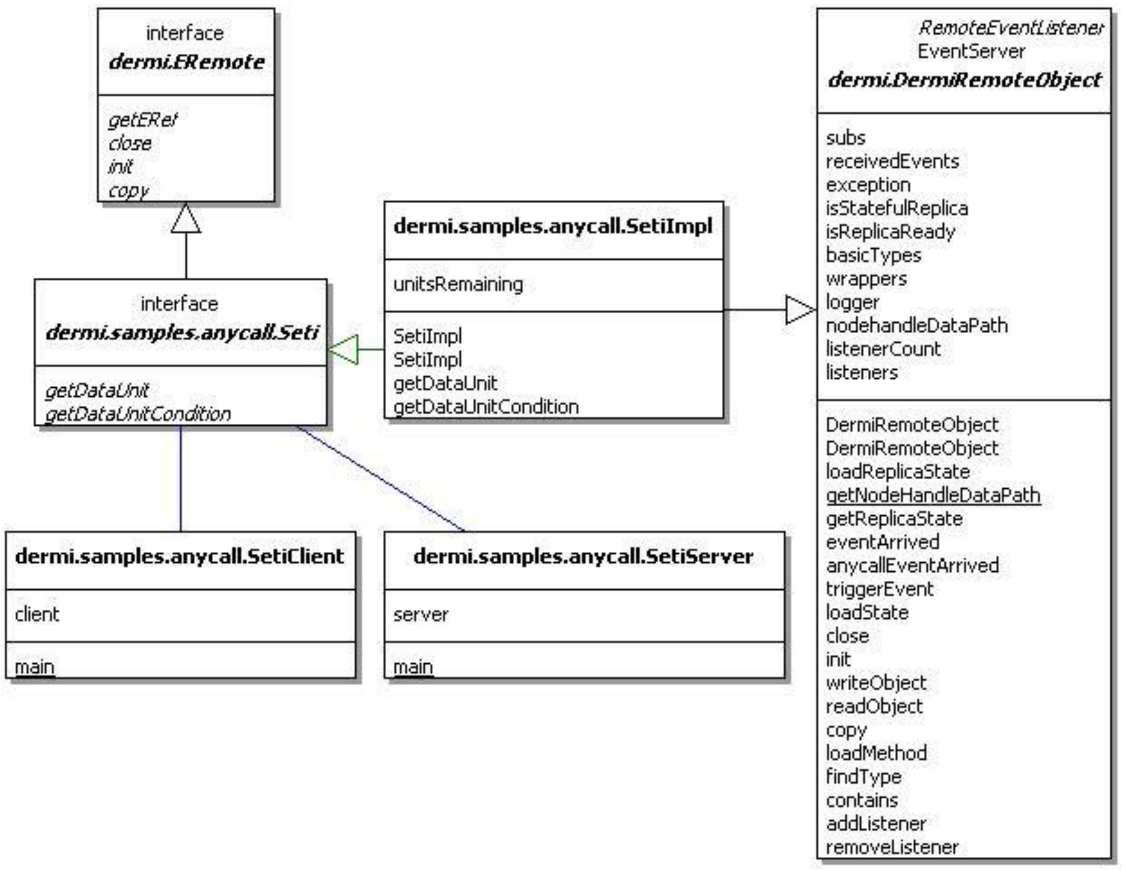


Figure A.5. Sample anycall Dermi application class diagram

A.1.2.3 Defining *manycall* methods

The *manycall*'s API follows the same strategy as anycall: it marks the methods with a special annotation tag (***Granularity.MANYCALL*), and defines the local and global condition methods with ***Granularity.MANYCALL_LOCAL_CONDITION*** and ***Granularity.MANYCALL_GLOBAL_CONDITION*** (see Figure A.6). The implementation part naturally hides all communication details from the developer.

```
/**
 * This is the interface to the Voting object
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DermiRemoteInterface
public interface Voting extends ERemote {
  @RemoteMethod (granularity = Granularity.MANYCALL)
  public Integer vote (Integer currentVotes, Integer maxVotes,
                       String party) throws RemoteException;

  @RemoteMethod (granularity = Granularity.MANYCALL_LOCAL_CONDITION)
  public boolean voteCondition (Integer currentVotes, Integer maxVotes,
                       String party) throws RemoteException;

  @RemoteMethod (granularity = Granularity.MANYCALL_GLOBAL_CONDITION)
  public boolean voteGlobalCondition (Integer currentVotes,
                    Integer maxVotes, String party) throws RemoteException;
}
```

Figure A.6. Definition of manycall methods in a remote object's interface

A.1.2.4 Working with the Decentralized Object Registry

From the developer's perspective, accessing the DOLR layer via the decentralized location service is accomplished via the *Registry* object. This object provides the *bind*, *lookup* and *remove* methods typical of any naming service. Therefore, we can easily bind object references, look them up and remove them through Dermi's API. This process is illustrated as follows:

- Once the remote object has been coded (*interface* and *implementation*), it is ready to be used. The first time, we initialize the object, and bind it on the decentralized object registry, so that it can be subsequently used by other clients. This process is shown in Figure A.7.

```
...
// Load Dermi's connection properties
Properties env = Registry.getEnvironment ("dermi-config.xml");

// Create remote object (the first time)
SimpleImpl server = new SimpleImpl (env);
// We can use the object now
server.setAge ("29");

// Now that the object is created, we can bind in on the DOLR
Registry.bind ("p2p://simple_dermi_object", server);
...
```

Figure A.7. Binding an object into the decentralized object location service
We first obtain connection properties by calling ***Registry.getEnvironment()***. These properties are passed to the object when it is initialized for the first time. After object initialization, we can call its methods. Whenever we want to insert the object's reference into the location service, we invoke ***Registry.bind()***.

- Whenever any other node in the network (another object probably) wishes to work with the *p2p://simple_dermi_object* we created before, it simply looks it up on the registry and starts working with it as if it was a local object (see Figure A.8).

```
...
// Load the registry first
Registry.loadRegistry ("dermi-config.xml");

// Look up an object in the registry
Simple client = (Simple) Registry.lookup ("p2p://simple_dermi_object");

// Execute remote object's methods
client.setAge ("30");
...
```

Figure A.8. Looking up an object from the decentralized object location service
First we must connect to the overlay network, and load the registry using the ***Registry.loadRegistry()*** method. Then we obtain the remote object's reference (*stub*) so as to be able to call its remote methods. Once they have been obtained, we can call the object's methods. Note that we are working with a *stub* which will transparently marshal/unmarshal calls to the remote object itself.

A.1.2.5 Implementing Distributed Interception Objects

From the developer's perspective, implementation of an interceptor object follows similar rules as those followed when a standard Dermi remote object is implemented. However, the interceptor must implement the ***EInterceptor*** interface. Its implementation must extend the ***DermiRemoteInterceptorObject*** class and, of course, implement the interceptor interface. If we wish to intercept methods from another object, the interceptor's methods signature will have to be the same as the original method if only the input parameters are to be intercepted. The output parameter can be changed so that the original parameters can be transformed into a *Vector*.

```
/**
 * This is the implementation of a log interceptor object. Its methods
 * will be called each time a call is made to the destination object (in
 * this case the Simple object)
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @author Pedro Garcia    <pedro.garcia@urv.net>
 * @version 1.2
 */
public class LogInterceptorImpl extends DermiRemoteInterceptorObject
                                               implements LogInterceptor {
  ...

  public Vector setAge (String p0) throws RemoteException {
    System.out.println ("[setAge: adding two more years]");

    // If we wished to change the value of parameters, simply change them
    // and pile them up in order in the returned Vector
    Vector v = new Vector();
    v.add (new String ("" + (Integer.parseInt (p0) + 2)));
    return v;
  }

  public void getAge() throws RemoteException {
    System.out.println ("[LogInterceptor - getAge]");
  }

  public Vector merge (Integer p0, String p1, Integer p2) throws
                                                        RemoteException {
    System.out.println ("[LogInterceptor - merge]");

    // If we wished to change the value of parameters, simply change them
    // and pile them up in order in the returned Vector
    Vector v = new Vector();
    v.add (p0);
    v.add (new String ("merge intercepted ;-)"));
    v.add (p2);
    return v;
  }
}
```

Figure A.9. Implementing a distributed interceptor object
The object must extend the DermiRemoteInterceptorObject class, and implement all methods to be intercepted.

Once the remote interceptor object has been implemented, we can instantiate it and bind it in realtime with the already running object. Therefore, all intercepted calls will first traverse the interceptor object, and then be routed towards the destination object. In order to commit a realtime binding, we must instantiate the interceptor, and pass the reference to the object we wish to intercept.

A.1.2.6 Defining Replicated Objects

From the developer's point of view, the marking of an object as *stateful* implies that it must implement the ***StatefulReplica*** interface. This interface provides two methods: ***loadRemoteObjectState()***, called whenever the object's state is to be loaded from the DHT layer, and ***getRemoteObjectState()***, called whenever the object's state is to be sent to the DHT layer.

When the object replica is initialized, state is recovered from the DHT layer by calling ***loadReplicaState()*** in the constructor. This method performs an *anycall* to the same object's multicast group, which calls ***getRemoteObjectState()*** on the remote object, and with the returned value, it populates state information about the local object.

```
...
// Load the registry first
Registry.loadRegistry ("dermi-config.xml");

// Look up the Simple object in the registry
Simple obj = (Simple) Registry.lookup ("p2p://simple_dermi_object");

// Instantiate and bind the interceptor object with the Simple object
LogInterceptorImpl server = (LogInterceptorImpl) Registry.loadInterceptor (
"dermi.samples.interception.LogInterceptorImpl", obj);

// The interceptor is loaded. All calls done to the Simple object will
// automatically traverse our interceptor now

...
```

Figure A.10. Binding an interceptor to an already running object
The ***loadInterceptor()*** method allows dynamic interceptor binding.

```java
public class SpriteImpl extends DermiRemoteObject implements Sprite,
                                                              StatefulReplica {
  private int x;

  ...

  // Dermi constructor: REQUIRED
  public SpriteImpl (int x, Properties env) throws RemoteException {
    super (env);
    // Default x value
    this.x = x;

    // Load replica state
    super.loadReplicaState();
  }

  /**
   * Method for loading state into this object
   * @param data SpriteValues Object state
   */
  public void loadRemoteObjectState() throws RemoteException {
    SpriteData data = (SpriteData) super.loadState();
    this.x = data.getX();
  }

  /**
   * Method for obtaining this object's state
   * @return SpriteData
   */
  public Serializable getRemoteObjectState() {
    return new SpriteValues (x);
  }
}
```

Figure A.11. Object replication example
All stateful objects must implement the *StatefulReplica* interface, and implement ***loadRemoteObjectState()*** and ***getRemoteObjectState()*** methods. When initializing the object, we must force the loading of the object's state.

Annex B. p2pCM’s Insights

B.1 Application Programming Interface

In this section we not only analyze p2pCM’s core classes and their interactions, but also how they are organized in order to provide all of their exposed services. We also focus on how p2pCM components can be implemented and outline all the necessary steps for this goal to be accomplished.

B.1.1 Core Classes

p2pCM’s core class diagram is shown in Figure B.1. We will briefly describe each of them and outline the main methods which provide p2pCM functionalities.

B.1.1.1 ComponentInterface

The *ComponentInterface* interface extends Dermi’s *ERemote* interface. This is the interface to be extended by all p2pCM remote component interfaces. Not only must components implement *ERemote*’s methods, but also the *queryInterface()* method, and the *activate()* and *passivate()* methods. The *queryInterface()* method must be explicitly implemented by the component’s implementation. However, since the *Component* base class already provides a default implementation for the *activate()* and *passivate()* methods, component implementation classes may or may not override them.

B.1.1.2 Component

The *Component* class is the base class to be extended by all p2pCM remote component implementations. This class is basically a Dermi remote object which offers all component services to p2pCM developers. Therefore, it inherits from *DermiRemoteObject* and provides default implementation methods for activation and passivation, as well as persistent (DHT) storage / retrieval and de-serialization. Component’s life cycle management is performed through a *ComponentControl* instance.

- *storePersistentState()*. This method can be called to serialize the component's state in the underlying DHT as a *Serializable* object.
- *loadPersistentState()*. This method can be used to restore the component's persistent state from the DHT.
- *activate()*. This method implements the default activation policy for components. Therefore, it follows this algorithm:
  - First the node's five most suitable replicas are obtained.
  - If there are fewer than two replicas, it is impossible to activate new components (node is isolated)
  - The node which is overwhelming this component is identified, and a new object replica is activated there, unless an instance is already active within that node.

  Naturally, this method can be overridden to implement more complex activation policies.
- *passivate()*. This method implements the default passivation policy for components. It unloads the component from memory and serializes its state into the DHT. This process is performed whenever the component's instance is not utilized within a threshold specified in seconds.

B.1.1.3 ComponentControl

This class models the component's life cycle manager. It provides the necessary callbacks for component activation and passivation. It is a default implementation of a life cycle manager, and it keeps track of the number of invocations. If it detects a low memory condition or a low invocation interval, it calls the *passivate()* method on the *Component*. On the contrary, if it detects a high invocation interval, new component replicas are spawned by calling the *activate()* method on the *Component*. This component's life cycle behaviour can be easily overriden by extending the class.

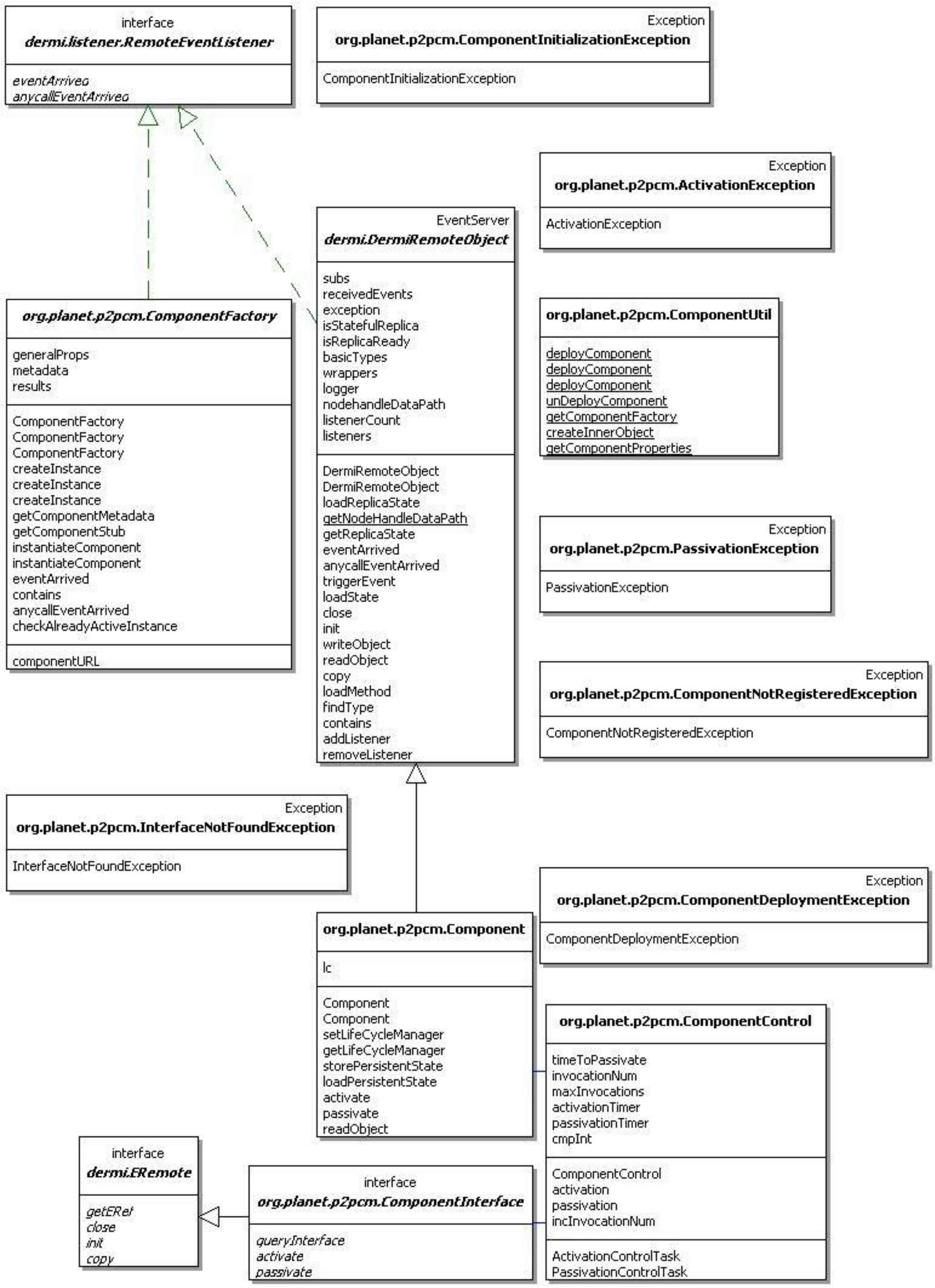


Figure B.1. p2pCM Class Diagram
Notice how p2pCM uses Dermi as its remote object foundation.

B.1.1.4 ComponentFactory

The *ComponentFactory* class provides the methods necessary for instantiating new component instances. All specific component factories must extend this class so that new component instances of their type can be created. The main method provided is *createInstance()*:

- *createInstance()*. Two variants of this method are provided to create component instances: one with initialization arguments and the other without. The default behaviour is applied when a component instance is created. Basically, if a previous instance is found to be already running on the network (checked by using the *checkAlreadyActiveInstance()* anycall method), its *stub* reference is returned (*getComponentStub()*). Otherwise, a brand new component instance is created locally (*instantiateComponent()*), and its reference is returned.

B.1.1.5 ComponentUtil

The *ComponentUtil* class provides static methods that allow common tasks among any component's life cycle. It basically manages component deployment / undeployment from the DHT.

- *deployComponent()*. Based on a parameter-specified URL or path to a packed component JAR file, it extracts its metadata and deploys the component on the DHT, under the specified p2p URI. As a consequence, this component becomes usable by p2pCM, and can be instantiated.

- *undeployComponent()*. Deletes any trace of the component from the DHT. If attempts are made to instantiate the component when it is not deployed, a *ComponentNotRegisteredException* is thrown.

B.1.1.6 Exceptions

Some exceptions can be thrown by p2pCM, all of which extend Dermi's *RemoteException* base exception class. These are:

- *ComponentInitializationException*. Thrown whenever something goes wrong while trying to instantiate a component.

- *ActivationException*. Thrown whenever something bad happens when trying to activate a new component instance.

- *PassivationException*. Thrown whenever something bad happens when trying to passivate a new component instance.

- *InterfaceNotFoundException*. Thrown whenever the specified queried interface is not implemented by the component.

- *ComponentDeploymentException*. Thrown whenever something goes wrong while deploying a component.

- *ComponentNotRegisteredException*. Thrown when trying to lookup a component that has not been deployed yet.

B.1.2 Programming p2pCM

This section describes how developers should build reusable components with p2pCM, from design and implementation through deployment and use.

B.1.2.1 Defining a Remote Component

The definition of a p2pCM remote component involves several steps, which include the implementation of the component's factory class, the design of the component's interfaces, and finally the implementation of the component itself. Finally, the component must be deployed on the DHT if it is to be usable.

From the developer's perspective, developing the component's factory requires the following steps:

- A class should be created that extends p2pCM's **ComponentFactory** class.

- This class should be annotated with the ***@DCMFactory*** tag, and component's metadata should be specified (identification, is it stateful?, is it replicated?, persistence type, and deployment URI)

- The factory constructor should be implemented (simply calling the superclass constructor)

- The ***createInstance*** methods should be overridden (if necessary) to allow component instantiation with and without initialization parameters.

An example of a typical component factory implementation is shown in Figure B.2.

```
/**
 * Simple component's factory class. Note that it must extend
 * <b>ComponentFactory</b>
 * It must provide a standard set of annotations which indicate the
 * component's metadata:
 *     - id          --  Component's unique ID
 *     - stateful    --  Whether it is stateful or stateless
 *     - persistent --  Which kind of persistence is applied
 *     - replicated --  Whether it is replicated or not (singleton)
 *     - url         --  URL where the component's metadata entry will be
 *                       bound in the decentralized registry
 *
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DCMFactory (
    id = "39fb8620-38e1-45c7-9275-4fe3133d19ac",
    stateful = false,
    persistent = PersistenceType.CONTAINER,
    replicated = true,
    url = "p2p://results.deim.etse.urv.cat"
    )
public class ResultsFactory extends ComponentFactory {
  /**
   * ResultsFactory's constructor -- REQUIRED
   * @param componentURL String Component's URL locator
   * @throws RemoteException If something goes wrong ;-)
   */
  public ResultsFactory (String componentURL) throws RemoteException {
    super (componentURL);
  }

  /**
   * This method creates an instance of this component (without arguments),
   * and returns an implementation of the specified interface
   * @param interf String Interface name
   * @param instance String Instance name
   * @return ComponentInterface Returns an implementation of
   *                            ComponentInterface
   * @throws InterfaceNotFoundException Thrown if specified interface
   *                                    cannot be found
   * @throws RemoteException Thrown if something else goes wrong ;-)
   */
  public ComponentInterface createInstance (String interf, String instance)
                     throws InterfaceNotFoundException, RemoteException {
    return super.createInstance (interf, instance);
  }

  /**
   * This method creates an instance of this component (with arguments),
   * and returns an implementation of the specified interface
   * @param interf String Interface name
   * @param instance String Instance name
   * @param args Object[] Component arguments
   * @return ComponentInterface Returns an implementation of
   *                            ComponentInterface
   * @throws InterfaceNotFoundException Thrown if specified interface
   *                                    cannot be found
   * @throws RemoteException Thrown if something else goes wrong ;-)
   */
  public ComponentInterface createInstance (String interf, String instance,
Object...args) throws InterfaceNotFoundException, RemoteException {
    return super.createInstance (interf, instance, args);
  }
}
```

Figure B.2. Implementation of a component's factory class
It is important to extend the ***ComponentFactory*** class, and appropriately annotate the component's metadata.

From the developer's point of view, a component's interface and implementation are practically seen as a Dermi object, with some particularities:

- The component's interfaces must extend p2pCM's ***ComponentInterface*** class, and be annotated as ***@DermiRemoteInterface***

- Interface methods must be annotated as Dermi methods

```
@DermiRemoteInterface
public interface Result extends ComponentInterface {
   @RemoteMethod (
         granularity = Granularity.MULTICALL,
          type = SynchronyType.SYNCHRONOUS
          )
   public void addResult (String data) throws RemoteException;

   @RemoteMethod (
         granularity = Granularity.ANYCALL,
          type = SynchronyType.SYNCHRONOUS
         )
   public String getResult() throws RemoteException;
}
```

Figure B.3. A p2pCM component interface
Component interfaces must be tagged as ***@DermiRemoteInterface***, and extend ***ComponentInterface***. The methods must be annotated as Dermi methods.

- The implementation class must extend p2pCM's ***Component*** class, and implement the component's interfaces. It must also be annotated as ***@DCMImplementation***.
- The ***queryInterface()*** method must be overriden in order to return the specified view of the component depending on the component's interface queried for.

An example of a component's interface is shown in Figure B.3. An implementation is shown in Figure B.4

```
/**
 * This is the component's base class. In this case, all of the component's
 * interfaces are implemented
 * It is in fact a special Dermi Remote Object that must extend the
 * Component class, and be labelled with @DCMImplementation annotation tag.
 *
 * @author Carles Pairot   <carles.pairot@urv.net>
 * @version 1.2
 */
@DCMImplementation
public class ResultsImpl extends Component implements Result, Filter {
  // Dermi constructors: REQUIRED
  ...


  // Component's methods implementation
  ...

  /**
   * The queryInterface method must return a suitable component
   * implementation for the specified interface
   * @param interf String The desired interface implementation
   * @return ComponentInterface The desired implementation instance
   * @throws InterfaceNotFoundException If the component does not provide
   *                      an implementation for such specified interface
   */
  public ComponentInterface queryInterface (String interf)
                                        throws InterfaceNotFoundException {
    try {
      if (interf.equals ("org.planet.p2pcm.ComponentInterface") ||
          interf.equals ("org.planet.p2pcm.samples.simple.Result") ||
          interf.equals ("org.planet.p2pcm.samples.simple.Filter")) {
        return (ComponentInterface) this.copy();
      }
      else {
        throw new InterfaceNotFoundException ("Interface " + interf + " not
                                              implemented.");
      }
    } catch (RemoteException re) {
      throw new InterfaceNotFoundException ("Unable to query for interface
                                " + interf + ": " + re.getMessage());
    }
  }
}
```

Figure B.4. A p2pCM component implementation
Component implementations must extend ***Component*** class, and be annotated as ***@DCMImplementation***. The method ***queryInterface()*** must be overriden to deal with different component views.

B.1.2.2 Component Deployment / Undeployment

Once the component factory has been implemented, we should implement the component itself (previous section) and, after that, we should deploy it on the network, so that it can be worldwide visible and instantiable. To do so, we should call the ***ComponentUtil.deployComponent()*** method, and pass the packaging JAR file that contains the component's classes. There is a ***ComponentUtil.undeployComponent()*** method which allows component undeployment from the network. Figure B.5 illustrates this process.

```
// This is how components are deployed: we must specify a JAR file which
// contains all component's classes
ComponentUtil.deployComponent (new URL ("file:./class_store/components/result-
simple.jar"));

// Test component
...

// If you wish to undeploy a component, you can proceed this way
ComponentUtil.unDeployComponent ("p2p://results.deim.etse.urv.cat");
...
```

Figure B.5. Deploying a p2pCM component
Once the component's classes are packed in a JAR file, we can deploy it using ***ComponentUtil.deployComponent()***.

B.1.2.3 Component Instantiation

Once we have implemented the component's factory, its interfaces and the implementations, we are ready to instantiate the component. To do so, we must follow these guidelines:

- Look the component's factory up in the decentralized registry

- Create an instance of the component, specifying the desired view (interface) to be returned. We must specify the instance identifier as well and, optionally, any initialization parameters

- Once this has been done, we can work with the component's view

- If we desire to work with another component's view (interface), we can invoke the ***queryInterface()*** method

This behaviour is shown in Figure B.6.

```
...
ComponentFactory fac = null;

// Get component's factory
try {
  fac = ComponentUtil.getComponentFactory (
                                        "p2p://results.deim.etse.urv.cat");
} catch (ComponentNotRegisteredException e) {
  ...
}

// Instantiate a "SETI" instance for the component Result
try {
  Result res = (Result) fac.createInstance (
                         "org.planet.p2pcm.samples.simple.Result", "SETI");

  // Call component's methods
  res.addResult ("new_data_unit_00001");

  // Get a new view on the Result component (the interface Filter)
  Filter filt = (Filter) res.queryInterface (
                                  "org.planet.p2pcm.samples.simple.Filter");

  // Call methods on the new view
  Object[] a = filt.getBestGaussians();
  ...
} catch (InterfaceNotFoundException e) {
  // Queried interface is invalid
  ...
}
...
```

Figure B.6. A p2pCM component instantiation
It is mandatory to obtain the component's factory first. We can then instantiate a new component instance, which will be created locally if it is the first occurrence in the network, or we will be returned a stub to one that is already running.

Figure B.7 shows a sample p2pCM application's class diagram. The *Slides* component is defined. It implements two views: the *Screen* and the *Controls* interfaces. They can both be queried, and we can observe that both are implemented by the *SlidesImpl* class, which models the component's base implementation. Note how interfaces inherit from *ComponentInterface* and how the *SlidesFactory* extends *ComponentFactory*, and *SlidesImpl* does the same with the *Component* class. The *TestSimpleComponent* class deploys the component and uses it.

B.1.2.4 Stateful Components

Stateful components must implement the ***StatefulReplica*** interface, which provides the ***loadRemoteObjectState()*** and ***getRemoteObjectState()*** methods used to load and get the object's state, respectively. On replica activation, Dermi anycalls (***getRemoteObjectState()***) to the multicast group asking for the closest up-to-date object copy in the network. This copy returns its state to the recently activated object, thus achieving initial state loading (***loadRemoteObjectState()***). This mechanism is the same as the one for stateful replica remote objects in Dermi.

From the developer's perspective, if the *PersistenceType.CONTAINER* mode is chosen, state storage and recovery is completely transparent. Once a component instance that has been previously passivated on the DHT is re-created, p2pCM will first try to load state from an already running instance. If none is found, then the state will be obtained from the DHT itself. The programmer remains completely unaware of this process and needs to code nothing to achieve it (but must remember to implement the ***StatefulReplica*** on the component).

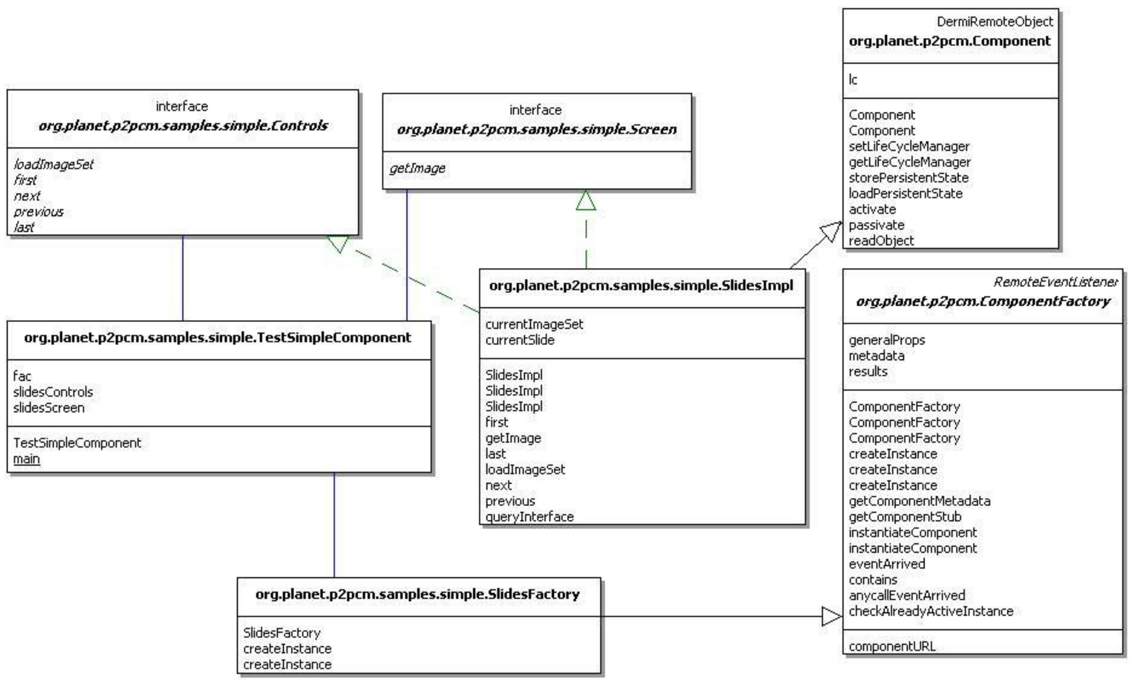


Figure B.7. Sample p2pCM Application Class Diagram

Annex C. SNAP's Insights

C.1 Application Programming Interface

As with other annexes, in this section we describe SNAP's core classes and their interactions and give an overview of how developers can use SNAP to construct wide-area distributed applications.

C.1.1 Core Classes

Although developers do not need SNAP's API to program and deploy applications on our infrastructure, SNAP exposes some of its internal methods to developers so that applications can dynamically activate new replicas, perform redirections to other applications, and check database liveness. There is also a simple introspection API which shows all active SNAP applications, and for each of them reveals which nodes host their instances. This introspection API will be further investigated in future work.

SNAP's API is shown in Figure C.1 and it corresponds to the ***org.planet.snap.IApplication*** interface. Figure C.2 shows an example of how the introspection API is used.

```
@DermiRemoteInterface
public interface IApplication extends ERemote {

  // Returns the URL of this running web application instance
  @RemoteMethod (granularity = Granularity.ANYCALL)
  public String getAppInstance (String appp2pUrl) throws RemoteException;

  // Anycall condition method for the one above (returns true if web
  // application's signature verifies
  @RemoteMethod (granularity = Granularity.ANYCALL_CONDITION)
  public boolean getAppInstanceCondition (String appp2pUrl)
                                             throws RemoteException;

  // Redirects to the SNAP application specified in the p2pUrl parameter
  @RemoteMethod (granularity = Granularity.MULTICALL,
                 type = SynchronyType.SYNCHRONOUS)
  public Properties redirectToSnapApp (String p2pUrl, String webAppDir,
                                       IApplication snapSkel)
                      throws ApplicationDeploymentException, RemoteException;
  // Starts a new database instance on the specified node
  @RemoteMethod (granularity = Granularity.DIRECTCALL,
                 type = SynchronyType.SYNCHRONOUS)
  public void startNewDatabaseInstance (dermi.core.NodeHandle nh,
                      String dbGroup, int clusterSize) throws RemoteException;

  // Deploys a new SNAP application replica on the specified node
  @RemoteMethod (granularity = Granularity.DIRECTCALL,
                 type = SynchronyType.SYNCHRONOUS)
  public Object deploySnapAppReplica (dermi.core.NodeHandle nh, String p2pUrl,
     String webAppDir) throws ApplicationDeploymentException, RemoteException;

  // Checks wether the specified node contains a database instance alive
  @RemoteMethod (granularity = Granularity.DIRECTCALL,
                 type = SynchronyType.SYNCHRONOUS)
  public boolean isDatabaseAlive (dermi.core.NodeHandle nh)
                                                  throws RemoteException;

  // Introspection API

  // Gets a listing of all available SNAP web applications
  public Collection<String> getApplications() throws RemoteException;

  // Returns a stub which models a gateway to the specified application
  public ERemote getApplication (String p2pUrl) throws RemoteException;

  // Returns all IP + Port listing of physical nodes in which this SNAP web
  // application is currently running (returns the nodes of the web
  // application cluster)
  @RemoteMethod (granularity = Granularity.MULTICALL,
                 type = SynchronyType.SYNCHRONOUS)
  public Collection<String> getInstances() throws RemoteException;

  // Returns all IP + Port listing of physical nodes in which this SNAP web
  // application holds an active database instance
  @RemoteMethod (granularity = Granularity.MULTICALL,
                 type = SynchronyType.SYNCHRONOUS)
  public Collection<String> getDatabaseInstances() throws RemoteException;
}
```

Figure C.1. SNAP's API. Interface *IApplication*

Figure C.3 shows SNAP's class diagram. As we can see, SNAP uses both p2pCM and Dermi classes as its foundation elements. We now describe each class and their main functionalities.

```
...

// Get list of available SNAP applications
Collection c = app.getApplications();

// Iterate through them and print their name (p2pUrl)
Iterator<String> it = c.iterator();

String appName = "";

while (it.hasNext()) {
  appName = it.next();
  out.println ("Application found: " + appName);
}

// Get the application instance for the last application
```

```
IApplication iapp = (IApplication) app.getApplication (appName);

// Print number of instances
out.println ("# of deployed instances for application " + appName + ": ");
out.println (iapp.getInstances().size());

// Print location of instances
out.println ("IP + Port instances: " + iapp.getInstances());

// Print location of database instances
out.println ("Deployed IP + Port DB servers: " + iapp.getDatabaseInstances());

...
```

Figure C.2. SNAP's Introspection API usage

C.1.1.1 IApplication

This interface has already been introduced at the beginning of this section. Its purpose is to provide SNAP's remote interface (it extends Dermi's *ERemote* interface). Therefore, all of SNAP's exposed methods are in fact Dermi remote methods.

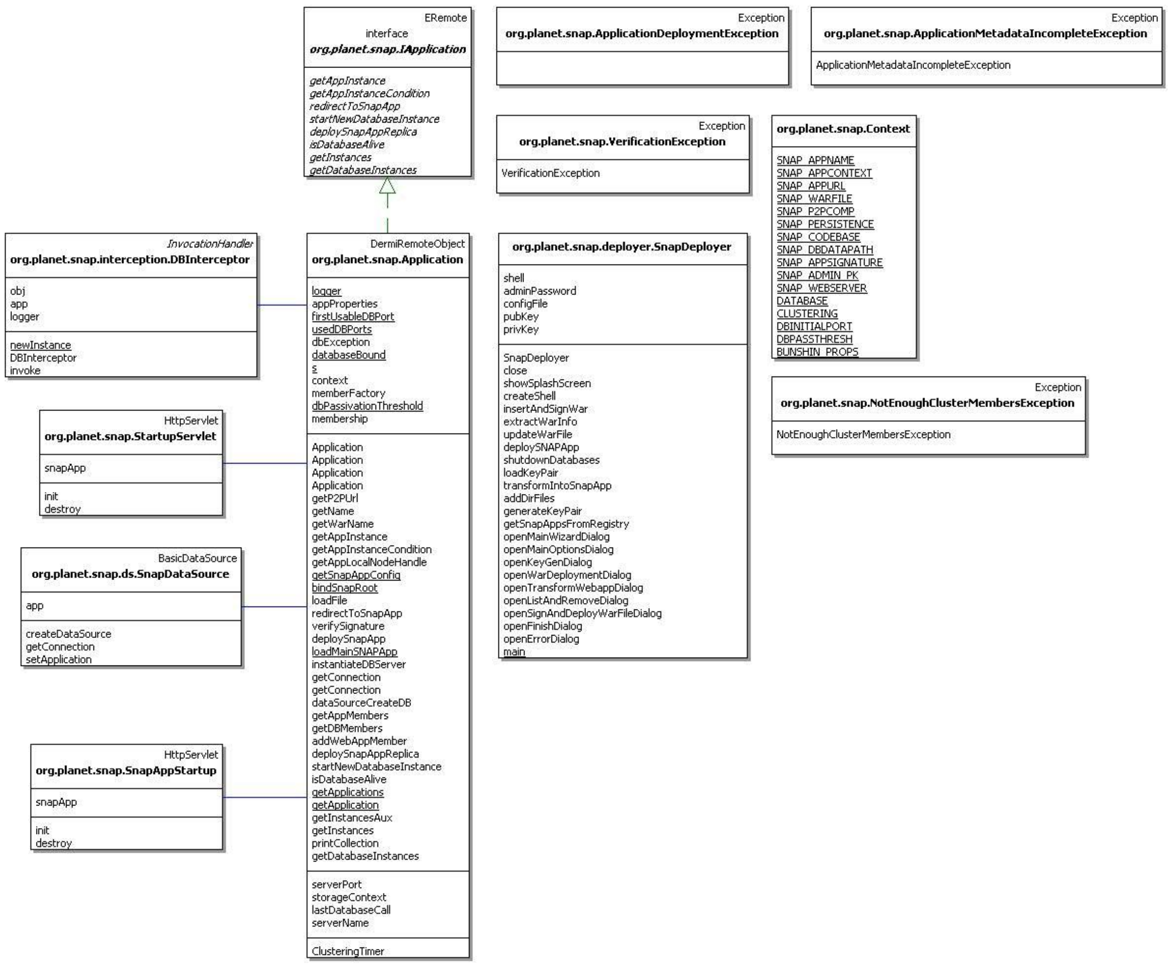


Figure C.3. SNAP's Class Diagram

C.1.1.2 Application

The *Application* class is the implementation of the above remote interface in the form of a *DermiRemoteObject*. It provides the following core methods:

- *getAppInstance() / getAppInstanceCondition()*. This is an *anycall* method which returns a reference to a local SNAP application instance. If the *getAppInstanceCondition()* method verifies (the application is correctly signed), an URL pointing to the web application is returned.

- *redirectToSnapApp()*. This method is used to redirect from a p2p-URI style identifier to a physical HTTP web application, which will normally be the closest one if it is already running elsewhere. If it is not, it is deployed on the local webserver instance by calling the *deploySnapApp()* method.

- *startNewDatabaseInstance()*. This method allows a new database server to be instantiated, specifying that maximum cluster size has been reached.

- *deploySnapAppReplica()*. This method is used to deploy a replica of a SNAP application onto a specified node.

- *isDatabaseAlive()*. This method checks whether a database instance is alive on a specified node.

- *ClusteringTimer*. This inner task periodically checks the application / database clustering status and balances it if it notes that more fault tolerance nodes are required.

- *Introspection API methods*. These methods have already been described at the beginning of this section.

C.1.1.3 Context

This is a simple utility class that holds all of SNAP's constants.

C.1.1.4 SnapAppStartup

This is an *HttpServlet* subclass, which serves as the initial SNAP infrastructure load point. Therefore, any SNAP application is required to load this servlet on startup, by specifying it on its *web.xml* deployment descriptor file.

C.1.1.5 StartupServlet

This is an *HttpServlet* subclass that creates SNAP's main application handler. It is SNAP's main web application running on each node. It is responsible for managing all other SNAP applications running on the same webserver instance.

C.1.1.6 SnapDataSource

The *SnapDataSource* class extends the *java.sql.BasicDataSource* class. This is SNAP's way of providing transparent DataSource access to J2EE applications. Therefore, it overrides the *createDataSource()* method, and returns a connection to the local HSQLDB database instance.

Moreover, the *getConnection()* method is also overridden to return a dynamic proxy object which encapsulates a *java.sql.Connection* object. This dynamic proxy class is the *DBInterceptor* class.

C.1.1.7 DBInterceptor

The database dynamic proxy class provides a transparent way of performing additional operations while calling standard *java.sql.Connection* methods from a J2EE web application. It intercepts all statement creation routines and, therefore, captures all *java.sql.Statement* calls to *execute()*, *executeQuery()*, and *executeUpdate()* methods. This way, SNAP keeps track of when the last database call was performed, and with this information it can passivate database instances that have not been used during a threshold time period.

C.1.1.8 SnapDeployer

The *SnapDeployer* class models SNAP's deployment tool behaviour.

C.1.1.9 Exceptions

Finally, a bunch of exception classes are used within SNAP. These include:

- *ApplicationDeploymentException*. Thrown when an application cannot be properly deployed.
- *ApplicationMetadataIncompleteException*. Thrown whenever the application's metadata is insufficient for deployment.
- *NotEnoughClusterMembersException*. Thrown when it is impossible to activate new application instances because there are not enough neighbour nodes available.

- *VerificationException*. Thrown when an already deployed application does not match the administrator's signature.

C.1.2 The Database Engine: HSQLDB-WAN

So far, we have hardly mentioned the database engine SNAP uses to store persistent data. In fact, the engine we use is a modified version of the HSQLDB [26] engine, but adapted to wide-area environments. Therefore, we want all members of a database cluster belonging to the same SNAP applications to share their data, and all changes performed to a member of the group to be propagated to all others.

This is the ideal setting in which to develop a p2pCM component which is integrated into the database engine that performs all communication logic within the same cluster group. We called this variation of the original database engine HSQLDB-WAN.

Figure C.4 shows HSQLDB-WAN's p2pCM integration class diagram. It shows all of the components that enable remote communication between database instances belonging to the same cluster group:

- *SQLReplicator / SQLReplicatorImpl / SQLReplicatorFactory*. This is a p2pCM component, which provides methods for propagating any SQL write statements results to all other database instances of the same cluster.

- *Membership / MembershipImpl / MembershipData / MembershipFactory*. This is another p2pCM component which helps to keep track of the active members of the database cluster. This component exposes methods to join, leave, and get the current *alive* members of the cluster. It periodically stores this information on the DHT (by means of the *MembershipData* state class), and recovers it if no other component instances for that cluster are active. The component's implementation class (*MembershipImpl*) also holds a *DeadCheckerTask* inner class which removes *dead* members (allowing later activation of new cluster members to replace them).

The *org.hsqldb.Database* class uses both these components to propagate membership changes and SQL updates to all cluster members by calling the appropriate component methods.

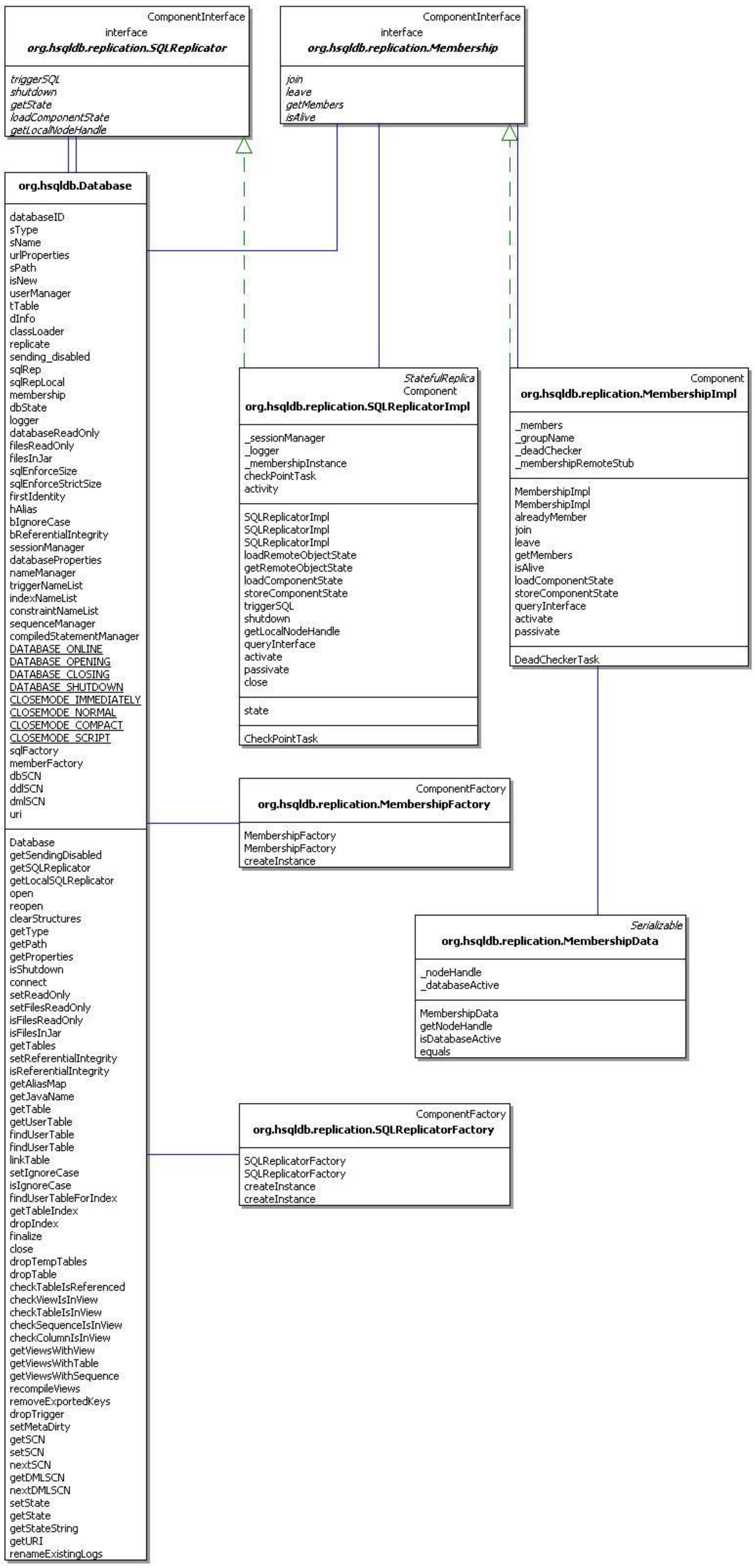


Figure C.4. HSQLDB-WAN Class Diagram

C.2 Programming SNAP

One of the most important features of SNAP which has already been highlighted at the beginning of this chapter is that it is easy to use. The idea is to facilitate as much as possible the transitioning process of any client-server based J2EE application to a SNAP application.

Therefore, already existing J2EE applications can **easily be ported to SNAP** through an easy and automated process of signing and packaging (via the *SNAPDeployer* tool), which also creates a new XML deployment descriptor file (*snap-war.xml*). The administrator can easily deploy static web application contents in the SNAP network too, without needing to change a line of code.

When dealing with applications which work with relational databases, we have also tried to make the transition to SNAP as transparent as possible. Developers can choose to use direct *Java Database Connectivity (JDBC)* connections (thus making slight changes in the way the JDBC connection is obtained), or *DataSources* (where they only have to update the DataSource's *Java Naming and Directory Interface (JNDI)* name in the application's *web.xml* file), without touching a line of code.

The idea is to make it easier for developers to use SNAP. In fact, unless they wish to access native replicated file warehouse features, or p2pCM components, already existing J2EE applications seamlessly adapt to SNAP with an automatic procedure of packaging, signature, and deployment.

Now we briefly describe how applications can easily be ported and deployed into SNAP, and that from the developer's perspective there are few requirements.

C.2.1 Web Application Adaptation

Adapting an already existing web application to work with SNAP is a simple process. There are several kinds of web applications and adaptation modes.

C.2.1.1 Static Web Applications

If your web application is a static one (i.e. includes only static HTML pages, say a home web page), adaptation is just a matter of replacing the default web page for the one included in SNAP's templates directory (**index.jsp**). Please note that the default original web page must be renamed **orig_index.html**. The process is that SNAP will first load the **index.jsp** page, which will initialize SNAP's environment, thus automatically redirecting to **orig_index.html**, which corresponds to the original static web application index.

The advantages of this approach is that SNAP will provide load balancing and failover to your static web applications, so they will be worldwide accessible even if the hosting server fails.

More specifically, what the new **index.jsp** page does is this:

```
<%@page import="dermi.*, org.planet.snap.*, java.util.*, java.io.*,
                java.net.*;" %>
<%

// Create and register application into SNAP
Application app = new Application (
        InetAddress.getLocalHost().getHostName(), request.getServerPort(),
        Application.getSnapAppConfig (application.getRealPath (
                                                      File.separator)));

application.setAttribute ("snap_webapp", app);

%>

<FRAMESET cols="0%, 100%" FRAMEBORDER="NO" BORDER="0" FRAMESPACING="0">
    <FRAME src="about:blank" SCROLLING="NO" noresize>
    <FRAME src="orig_index.html" noresize>
</FRAMESET>
```

Once this is done you are ready to deploy your static web application onto SNAP.

C.2.1.2 J2EE Web Applications

Persistence mode: Access to Database via DataSources

In order to adapt a dynamic web application which performs relational database queries working with container provided *DataSources*, we must add the following lines to the application's **web.xml** descriptor file, indicating the initial load of the SNAP's associated instance:

```
  <servlet>
      <servlet-name>startup</servlet-name>
      <servlet-class>org.planet.snap.SnapAppStartup</servlet-class>
      <load-on-startup>0</load-on-startup>
  </servlet>
```

By using SNAP's datasource configuration, we guarantee that persistence data is replicated among a determinate number of servers, to guarantee transparent failover and load balancing.

DataSource configuration will also be modified on the application's **web.xml** descriptor file by using the default **java:comp/env/jdbc/SnapDS** JNDI name. However, the DataSource name and configuration can be modified by opening Jetty's Configuration File (**etc/jetty.xml**). By following thish approach not a single line of code needs to be changed in order to port the application to SNAP.

```
<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<!-- Snap Datasource properties.                                      -->
<!-- + It uses a modified HSQLDB database with replication.           -->
<!-- + No host is specified, and neither port is.                     -->
<!-- + Only username and password arguments.                          -->
<!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -->
<Call name="addService">
  <Arg>
    <New class="org.mortbay.jetty.plus.DefaultDataSourceService">
      <Set name="Name">DataSourceService</Set>
      <Call name="addDataSource">
                  <Arg>jdbc/SnapDS</Arg>
                  <Arg>
                  <New class="org.planet.snap.ds.SnapDataSource">
                         <Set name="Username">sa</Set>
                         <Set name="Password"></Set>
                  </New>
                  </Arg>
                  </Call>
           </New>
     </Arg>
</Call>
```

Once you have changed the DataSource name, you are ready to deploy your web application onto SNAP.

Persistence mode: Direct JDBC Connection to Database

If your application does not use a JDBC connection pool via DataSources, then you must adapt it only to modify the connection phase so that persistent data is stored and replicated using SNAP's modified HSQLDB bundled version.

The first thing to do is to modify the application's **web.xml** file, as in the previous step (only to load the startup servlet).

By changing the way connections to the database are made, we benefit from SNAP's database replication service, thus guaranteeing that persistence data is replicated among a determinate number of servers, and transparent failover and load balancing is achieved.

How can a JDBC connection to the underlying SNAP database be made? Take a look at the following code snippet:

```
<%@page import="dermi.*, org.planet.p2pcm.*, org.planet.snap.*,
                java.util.*, java.io.*, java.net.*, java.sql.*"
%>

<%
Application app = (Application) application.getAttribute ("snapApp");

if (app == null) {
  throw new NullPointerException ("Application not initialized!");
}

// To obtain a SNAP-compatible JDBC connection
Connection con = app.getConnection ("sa", "");
...
%>
```

First of all, a reference to the underlying SNAP application instance must be obtained. Next, to get a JDBC connection, simply call the **getConnection (user, password)** method from the SNAP application instance. From now on, the rest of the web application code remains the same.

Once this is done you are ready to deploy your web application into SNAP.

C.2.2 Web Application Deployment

In order to be able to deploy any kind of web application onto SNAP, this application must be previously approved by the network's administrator.

The ideal case is to have the application delivered to the administrator, which will in turn sign it and make it available to the SNAP community. Note that if an application is deployed in a SNAP node but it has not been signed by the administrator, it will not work, since this security aspect is checked every time an application is accessed.

However, let us supose the application has been sent to the administrator (and YOU are the administrator), and you need to deploy it. The first thing to do is to define a file named **snap-war.xml**, which is to be located in the **META-INF** directory of the web application.

This file contains SNAP's metadata for this application. More specifically, it defines the persistence type, the clustering factor (the number of nodes where the application will dynamically be replicated), and database properties. A sample *snap-war.xml* file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>
    SNAP Web-Application descriptor
  </comment>
  <entry key="appName">SNAP TestSuite Application</entry>
  <entry key="appContext">snaptestsuite</entry>
  <entry key="appp2pUrl">p2p://this_entry_is_not_used_now</entry>
  <entry key="persistence">database</entry>
  <entry key="clustering">3</entry>
  <entry key="dbInitialPort">9999</entry>
  <entry key="dbDataPath">/WEB-INF/data/</entry>
  <entry key="dbPassivationThreshold">10</entry>
</properties>
```

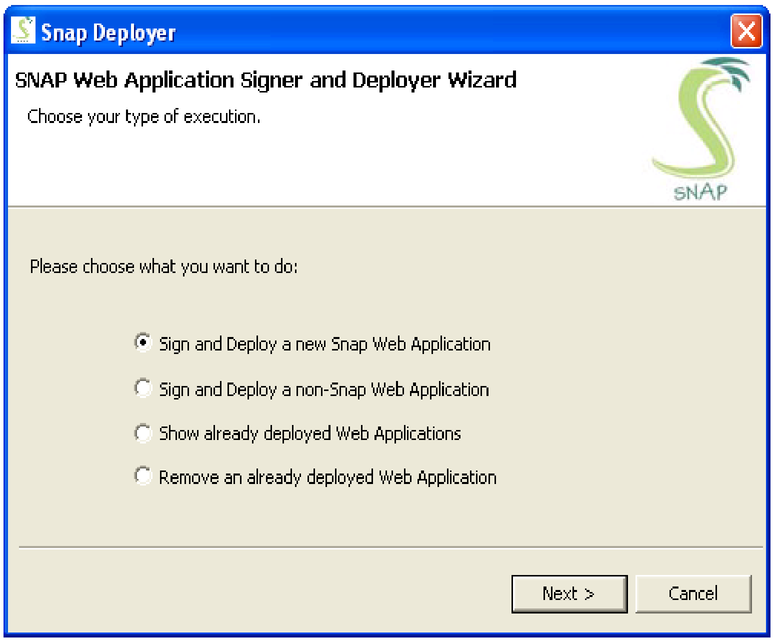
This is what the entries mean:

- **appName** – The application's name
- **appContext** – The application's context (in Jetty webserver)
- **appp2pUrl** – Unused entry – to be filled in by SNAPDeployer
- **persistence** – Persistence type (*database*) or delete the entry for none
- **clustering** – Clustering factor: the number of nodes where the application will dynamically be replicated
- **dbInitialPort** – Database initial port (subsequent database activations are to be bound in incremental ports)
- **dbDataPath** – Path where the database files are to be stored (relative to the application's context)
- **dbPassivationThreshold** – Minutes of inactivity for database instance passivation (to free up resources)

All entries should be filled in before trying to deploy the web application. Subsequent versions of SNAP will have this process integrated with the deployment tool. However, this step currently needs to be done manually.

The next step is to start the **SNAPDeployer** tool, in order to deploy the application onto the SNAP network. If everything is OK, the deployer will connect to the network, and show its splash screen.

Next, a step-by-step wizard will guide you through the deployment process. It is important to make a distinction in the options that the wizard presents you with (see Figure C.5).


Snap Deployer

SNAP Web Application Signer and Deployer Wizard

Choose your type of execution.

Please choose what you want to do:

[✓ Sign and Deploy a new Snap Web Application]

Sign and Deploy a non-Snap Web Application

Show already deployed Web Applications

Remove an already deployed Web Application

Figure C.5. SNAP Deployer option window

- **Sign and Deploy a new Snap Web Application** – This option deploys an already .WAR packed application onto SNAP.
- **Sign and Deploy a non-Snap Web Application** – This option deploys a directory containing a web application onto SNAP.
- **Show already deployed Web Applications** – This option shows information about all active and deployed web applications.
- **Remove an already deployed Web Application** – This option undeploys a SNAP web application.

The next deployment step generates an administrator's public/private key pair (if it is the first time), or reuses a **keystore.rsa** file which contains the administrator's public/private key pair (this file should be kept in a safe place).

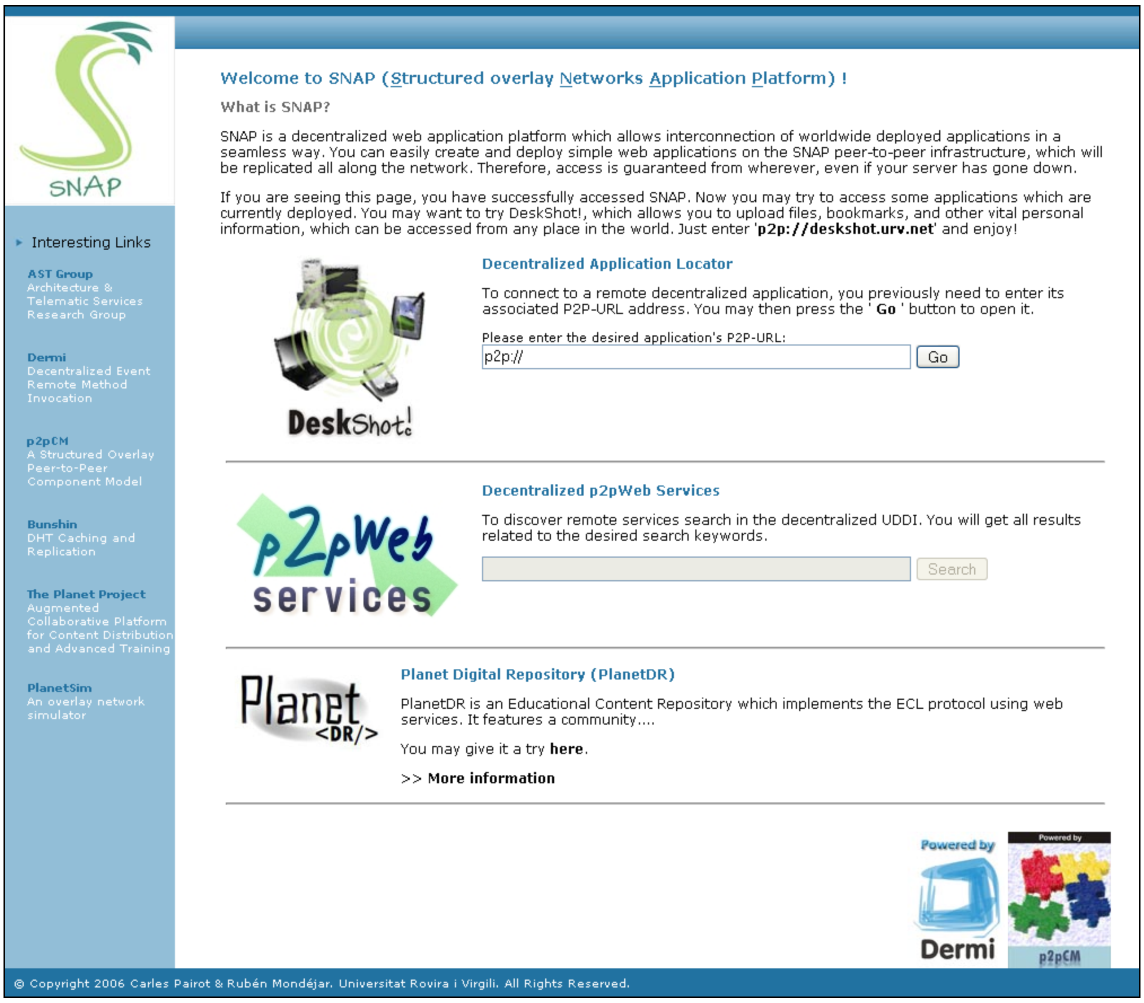
Finally, and after asking about the application's metadata (it is especially important to assign a correct **p2p://** identifier, as this will be used to **locate** the application via SNAP's decentralized application locator), the application will be signed, repacked and uploaded to the decentralized SNAP network.

C.2.3 Accessing SNAP applications

Once the application has been deployed on the SNAP network, it is ready to be used. But first it must be instantiated. A web application is instantiated automatically the first time it is called. The steps are summarized as follows:

- Open a web browser window and connect to your local machine (if you are hosting a SNAP node), or to a remote machine which is joined to the SNAP ring. For example: http://localhost:33333

- SNAP's home will appear on your screen, and you may now enter the p2p URL of your recently deployed application. For example: *p2p://cpairot-webpage.urv.net*


SNAP
Interesting Links
AST Group
Architecture & Telematic Services Research Group
Dermi
Decentralized Event Remote Method Invocation
p2pCM
A Structured Overlay Peer-to-Peer Component Model
Bunshin
DHT Caching and Replication
The Planet Project
Augmented Collaborative Platform for Content Distribution and Advanced Training
PlanetSim
An overlay network simulator
DeskShot!
p2pWeb services
Planet <DR/>
Powered by
Dermi
Powered by
p2pCM

Welcome to SNAP (Structured overlay Networks Application Platform) !

What is SNAP?

SNAP is a decentralized web application platform which allows interconnection of worldwide deployed applications in a seamless way. You can easily create and deploy simple web applications on the SNAP peer-to-peer infrastructure, which will be replicated all along the network. Therefore, access is guaranteed from wherever, even if your server has gone down.

If you are seeing this page, you have successfully accessed SNAP. Now you may try to access some applications which are currently deployed. You may want to try DeskShot!, which allows you to upload files, bookmarks, and other vital personal information, which can be accessed from any place in the world. Just enter '**p2p://deskshot.urv.net**' and enjoy!

Decentralized Application Locator

To connect to a remote decentralized application, you previously need to enter its associated P2P-URL address. You may then press the ' **Go** ' button to open it.

Please enter the desired application's P2P-URL:

p2p:// [Go]

Decentralized p2pWeb Services

To discover remote services search in the decentralized UDDI. You will get all results related to the desired search keywords.

[Search]

Planet Digital Repository (PlanetDR)

PlanetDR is an Educational Content Repository which implements the ECL protocol using web services. It features a community....

You may give it a try **here**.

>> **More information**

© Copyright 2006 Carles Pairot & Rubén Mondéjar. Universitat Rovira i Virgili. All Rights Reserved.

Figure C.6. SNAP's Home Page

- Now SNAP checks whether this application has already been instantiated elsewhere on the network. Since it is the first time, SNAP instantiates it on the local web server node. Therefore, it loads the SNAP infrastructure for that application, dynamically activates the database (if the application requires), and starts replicating the application instance among its k replica nodes, so as to provide fault tolerance. This process of application instance replication means that the application is deployed locally on other web server nodes, and it forms a *cluster* of replicated web applications. If one of the servers crashes, SNAP allows the closest one to respond, and the application is still accessible.

- After this process (the first time it obviously takes a little longer), SNAP redirects the browser to the local web application instance, and we can start working with the application.

- Imagine that the next day we want to continue working with the application. We follow the same steps and try to access *p2p://cpairot-webpage.urv.net*. This time, loading the application does not last as long, since it is already active. Therefore, we are redirected to the *closest* webserver and start working with the application running there.

Glossary

.NET Framework – The Microsoft .NET Framework is a component of the Microsoft Windows operating system. It provides a large body of pre-coded solutions to common program requirements, and manages the execution of programs written specifically for the framework. The .NET Framework is a key Microsoft offering, and is intended to be used by most new applications created for the Windows platform. *See DCOM.*

Anycast – Anycast is a network addressing and routing scheme whereby data is routed to the *nearest* or *best* destination as viewed by the routing topology. Anycast is generally used as a way to provide high availability and load balancing for stateless services such as access to replicated data.

API (Application Programming Interface) – An Application Programming Interface (API) is the interface that a computer system, library or application provides in order to allow requests for services to be made of it by other computer programs, and/or to allow data to be exchanged between them.

Application Level Multicast – Application Level Multicast is a way to provide *multicast* when it is not possible to use IP Multicast. Routing is not performed at the network level, but at the application level. This way, events and messages are relayed from origin to destination by an application specific component, called the distributed information bus. *See Multicast, IP Multicast, Event Bus.*

Aspect-Oriented Programming (AOP) – In software engineering, the programming paradigms of Aspect-Oriented programming (AOP) and aspect-oriented software development (AOSD) attempt to aid programmers in the separation of concerns, specifically crosscutting concerns, as an advance in modularization. AOP does so using primarily language changes, while AOSD uses a combination of language, environment, and methodology.

BFS (Breadth-First Search) – Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Britney Problem, the – The Britney Problem is a common issue in p2p systems that use distributed hash tables (DHTs) to organize their namespaces. In many DHTs a given key, such as *britney.mpg*, is mapped to a single node that will store this file and serve all requests to it. What occurs is that popular keys get mapped to single nodes, causing hotspots that get a tremendous number of requests and therefore cannot serve all client peers interested in the file. *See p2p, DHT.*

Broadcast – Broadcasting refers to transmitting a packet that will be received (conceptionally) by every device on the network. In practice, the scope of the broadcast is limited to a broadcast domain. *See Multicast.*

CBSD (Component-Based Software Development) – A software engineering discipline which tries to settle the basis for the design and development of reusable software component-based distributed applications. It claims that software components, like the idea of hardware components, used for example in telecommunications, can ultimately be interchangeable and reliable.

Churn – Within a peer-to-peer network, the churn rate refers to the number of peers leaving the system during a given period—usually an hour, rather than a year. It is a significant problem for large-scale systems, as they must maintain consistent information about peers in the system in order to operate most effectively. For example, with a high churn rate it may be impossible to index file locations, making some files inaccessible even though they are available to some peers.

Client/Server Architecture – Client-server is a network architecture which separates the client from the server. Each instance of the client software can send requests to a server or application server. There are many different types of servers; some examples include a file server, terminal server, or mail server. While their purpose varies somewhat, the basic architecture remains the same.

Clustering (Computer Cluster) – A computer cluster is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer. Clusters are commonly, but not always, connected through fast local area networks. Clusters are usually deployed to improve more speed and/or reliability than that provided by a single computer, while typically being much more cost-effective than single computers of comparable speed or reliability. J2EE application server vendors define a computer cluster as a group of machines working together to transparently provide enterprise services (support for JNDI, EJB, JSP, HttpSession, and component failover, and so on). *See Federation, J2EE.*

Component Life Cycle – The time between a component's creation (also known as *instantiation*) and the moment the component is no longer used (and is *destructed*). In between, the component may live through many other stages, like *passivation*, where the component's state is persisted to secondary memory storage, *activation,* where the component's state is recreated from secondary memory storage, and others. *See Reusable Component.*

Component Model – A distributed component-oriented model is an architecture for defining components and their interactions. It must provide a packaging technology for deploying binary component executables. Moreover, it needs a container framework for injecting life cycle, thus permitting activation and passivation of component instances. Other services include security, transactions, persistence, and events. *See Reusable Component.*

Container – An entity that provides life-cycle management, security, deployment and runtime services to any distributed component. *See Reusable Component*.

CORBA (Common Object Request Broker Architecture) – Common Object Request Broker Architecture (CORBA) is a standard for software componentry, created and controlled by the *Object Management Group (OMG)*. It defines APIs, communication protocol, and object/service information models to enable heterogeneous applications written in various languages running on various platforms to interoperate. CORBA therefore provides platform and location transparency for sharing well-defined objects across a distributed computing platform.

CSCW (Computer Supported Cooperative Work) – Computer Supported Cooperative Work (CSCW) is a discipline that addresses how collaborative activities and their coordination can be supported by means of computer systems.

DCOM (Distributed Component Object Model) – Distributed Component Object Model (DCOM) is a Microsoft proprietary technology for software components distributed across several networked computers to communicate with each other. It extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. It has been deprecated in favor of Microsoft .NET. *See .NET Framework.*

DHT (Distributed Hash Table) – Distributed hash tables (DHTs) are a class of decentralized distributed systems that partition ownership of a set of *keys* among participating nodes, and can efficiently route messages to the unique owner of any given key. Each node is analogous to an array slot in a hash table. DHTs are typically designed to scale to large numbers of nodes and to handle continual node arrivals and failures. *See Hash Function*.

Distributed Systems – A collection of (probably heterogeneous) automata whose distribution is transparent to the user so that the system appears as one local machine. This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent.

DNS (Domain Name System) – The Domain Name System (DNS) stores and associates many types of information with domain names, but most importantly, it translates domain names (computer hostnames) to IP addresses. It also lists mail exchange servers accepting e-mail for each domain. In providing a worldwide keyword-based redirection service, DNS is an essential component of contemporary Internet use. *See IP.*

DSHT (Distributed Sloppy Hash Table) – A Distributed Sloppy Hash Table (DSHT) is an indexing abstraction based on DHTs which creates self-organizing clusters of nodes that fetch information from each other to avoid communicating with more distant or heavily-loaded servers. The *sloppy* hash table refers to the fact that any overlay network of this kind is made up of concentric rings of DHTs, each ring representing a wider and wider geographic range. The DHTs are composed of nodes all within some latency of each other. It avoids hot spots (the *sloppy* part) by only continuing to query progressively larger sized rings if they are not overburdened — i.e. if there are many hits to the top-most two rings, it will just ping the close ones, and when it reaches a hit that is overloaded it stops progressing upward. This therefore decreases hot spots and at the same time limits the amount of global knowledge. *See DHT.*

ECL (eduSource Communication Language) – The eduSource Communication Language (ECL) is a protocol which defines a standard communication language for educational content repository interoperability. The ECL protocol enables these repositories to communicate with each other and enables other repositories and services to become a part of eduSource. The protocol is independent from existing protocols and enables developers to build universal tools and services that will enable their users to connect and use services provided by any repository connected to the eduSource network.

Event Bus (Distributed Information Bus) – The Distributed Information Bus is an architecture that allows extensibility in distributed systems. This approach allows for asynchronous communication between senders and receivers. The event bus is the application which makes this communication possible. *See Distributed Systems, Publish/Subscribe Event System*.

Federation – An analogous concept to clustering, but driven at a higher level. A federated network consists of a number of interconnected clusters, which communicate via multicast or direct sockets. *See Clustering.*

Free riding – In economics, collective bargaining, and political science, free riders are actors who consume more than their fair share of a resource, or shoulder less than a fair share of the costs of its production. The free rider problem is the question of how to prevent free riding from taking place, or at least limit its negative effects.

Grid Computing – Grid computing is a computing model that provides the ability to perform higher throughput computing by using many networked computers to model a virtual computer architecture that can distribute process execution across a parallel infrastructure. Grids use the resources of many separate computers connected by a network (usually the Internet) to solve large-scale computation problems.

GUID (Globally Unique Identifier) – A Globally Unique Identifier (GUID) is a pseudo-random number used in software applications. While each generated GUID is not guaranteed to be unique, the total number of unique keys (2^{128} or 3.4028×10^{38}) is so large that the possibility of the same number being generated twice is very small.

Hash Function – A hash function is a way of creating a small digital *fingerprint* from any kind of data. The function chops and mixes the data to create the fingerprint, which is often called a *hash value*. The hash value is commonly represented as a short string of random-looking letters and numbers. A good hash function is one that yields few hash collisions in expected input domains. In hash tables and data processing, collisions prevent data from being distibuished, making records more costly to find.

HTTP (Hypertext Transfer Protocol) – Hypertext Transfer Protocol (HTTP) is a method used to transfer or convey information on the World Wide Web. It is a patented open internet protocol whose original purpose was to provide a way to publish and receive HTML pages. *See WWW.*

IP (Internet Protocol) – Internet Protocol (IP) is a network layer protocol in the internet protocol suite and is encapsulated in a data link layer protocol. As a lower layer protocol, IP provides the service of *communicable* unique global addressing amongst computers.

IP Multicast – The single word *Multicast* is typically used to refer to IP Multicast, which is a delivery method in IP networks for efficiently sending datagrams to multiple receivers at the same time on networks by way of a multicast destination address. *See Multicast, Application Level Multicast.*

Java EE (Java Platform, Enterprise Edition, J2EE) – Java Platform, Enterprise Edition or Java EE (formerly known as Java 2 Platform, Enterprise Edition or J2EE up to version 1.4) is a programming platform —part of the Java Platform— for developing and running distributed multitier architecture Java applications, based largely on modular software components running on an application server. The Java EE platform is defined by a *specification*. Like other Java Community Process specifications, Java EE is also considered informally to be a standard because providers must agree to certain conformance requirements in order to declare their products as *Java EE compliant*; albeit with no ISO or ECMA standard. *See JDBC, JNDI.*

JDBC (Java Database Connectivity) – JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. *See API, J2EE.*

JERI (Jini Extensible Remote Invocation) – Jini Extensible Remote Invocation (JERI) provides programmatic access to each layer of an RMI call via an API and allows an RMI service deployer to choose the RMI implementation most suitable in a deployment scenario. JERI also defines a uniform mechanism to make remote objects available to answer remote method calls (object exporting), which was not standardized in prior RMI releases. The result of the new features is that RMI calls can now adhere to any security requirement.

Jini – Jini is a network architecture for constructing distributed systems where scale, rate of change and complexity of interactions within and between networks are extremely important and cannot be satisfactorily addressed by existing technologies. Jini technology provides a flexible infrastructure for delivering services in a network and for creating spontaneous interactions between clients that use these services regardless of their hardware or software implementations.

JMS (Java Message Service) – The Java Message Service (JMS) API is a Java MOM API for sending messages between two or more clients. JMS is a specification developed under the Java Community Process as JSR 914. *See MOM.*

JNDI (Java Naming and Directory Interface) – The Java Naming and Directory Interface (JNDI) is an API for directory services. It allows clients to discover and lookup data and objects via a name and, like all Java APIs that interface with host systems, is independent of the underlying implementation. Additionally, it specifies a service provider interface (SPI) that allows directory service implementations to be plugged into the framework. The implementations may make use of a server, a flat file, or a database; the choice is up to the vendor. *See API, J2EE.*

KBR (Key-Based Routing Substrate) – The Key-Based Routing (KBR) substrate is a common layer in all structured p2p overlay networks which allows for efficient message delivery based on the message key. Therefore, the message moves closer to the destination node following an approaching path that depends on the key’s value. *See p2p, DHT.*

Kerberos – Kerberos is a computer network authentication protocol which allows individuals communicating over an insecure network to prove their identity to one another in a secure manner. Kerberos prevents eavesdropping or replaying attacks, and ensures the integrity of the data. Its designers aimed primarily at a client-server model, and it provides mutual authentication — both the user and the service verify each other's identity. Kerberos builds on symmetric key cryptography and requires a trusted third party.

LAN (Local Area Network) – A Local Area Network (LAN) is a computer network covering a small local area, like a home, office, or small group of buildings such as a home, office, or college. Current LANs are most likely to be based on switched Ethernet or Wi-Fi technology running at 10, 100 or 1,000 Mbit/s. The defining characteristics of LANs in contrast to WANs (wide area networks) are their much higher data rates, their smaller geographic range, and the fact that they do not require leased telecommunication lines. *See WAN.*

LOM (Learning Object Metadata) – Learning Object Metadata (LOM) is a data model, usually encoded in XML, used to describe a learning object and similar digital resources used to support learning. The purpose of learning object metadata is to support the reusability of learning objects, to aid discoverability, and to facilitate their interoperability, usually in the context of online learning management systems (LMS).

LPC (Local Procedure Call) – A Local Procedure Call (LPC) is the same as a Remote Procedure Call where all communication happens on the same computer. *See RPC.*

MDS (Monitoring and Discovery System) – The Globus Monitoring and Discovery System (MDS) is a collection of Web services that monitor and discover the resources and services available in a grid. In a grid context, resource discovery is the systematic process of determining which grid resource is the best candidate at completing a job in the shortest amount of time with the most efficient use of resources.

MOM (Message Oriented Middleware) – Message-Oriented Middleware (MOM) comprises a category of inter-application communication software that generally relies on asynchronous message-passing as opposed to a request/response metaphor. Most MOM depends on a message queue system, although some implementations rely on broadcast or on multicast messaging systems.

MP3 (MPEG-1 Audio Layer 3) – MPEG-1 Audio Layer 3, more commonly referred to as MP3, is the most popular digital audio encoding and lossy compression format, designed to greatly reduce the amount of data required to represent audio, yet still sound like a faithful reproduction of the original uncompressed audio to most listeners.

Multicast – Multicast is the simultaneous delivery of information to a group of destinations using the most efficient strategy to deliver the messages over each link of the network only once and only creating copies when the links to the destinations split. *See IP Multicast, Application Level Multicast*.

Naming service – A distributed object service which allows objects to be named by means of binding a name to an object reference. The name binding may be stored in the naming service, and a client may supply the name to obtain the desired object reference.

NAT (Network Address Translation) – The process of Network Address Translation (NAT) involves re-writing the source and/or destination addresses of IP packets as they pass through a router or firewall. Most systems using NAT do so in order to enable multiple hosts on a private network to access the Internet using a single public IP address. According to specifications, routers should not act in this way, but many network administrators find NAT a convenient technique and use it widely. Nonetheless, NAT can complicate communication between hosts.

NodeHandle – An overlay network node's physical information handle. This *NodeHandle* object typically consists of the node's IP address and port, and its *NodeId*.

NodeId – The node identifier in an overlay network. Keys closest to the node's *NodeId* are mapped into it.

ObjectWeb Consortium – The ObjectWeb consortium is an international consortium mainly devoted to producing open source middleware, EAI, e-business, clustering, grid computing. ObjectWeb is a not-for-profit, international consortium dedicated to the development of high-quality open-source components for distributed applications (Web applications, grid computing, clusters, business integration, nomadic systems, etc).

OMG (Object Management Group) – Object Management Group (OMG) is a consortium, originally aimed at setting standards for distributed object-oriented systems, and now focused on modeling (programs, systems and business processes) as well as model-based standards in some 20 vertical markets. Founded in 1989 by eleven companies (including Hewlett-Packard Company, Apple Computer, American Airlines and Data General), OMG mobilised to create a cross-compatible distributed object standard. The goal was a common portable and interoperable object model with methods and data that work using all types of development environments on all types of platforms. At its founding, OMG set out to create the initial Common Object Request Broker Architecture (CORBA) standard which appeared in 1991. OMG has also created the standard for Unified Modeling Language (UML). It has further expanded into Model Driven Architecture (MDA), and related set of standards. *See CORBA.*

p2p (Peer-to-Peer) – Peer-to-Peer (p2p) comprises a class of systems and applications that employ distributed resources to perform a critical function in a decentralized manner. The resources encompass computing power, data (storage and content), network bandwidth, and presence (computers, human, and other resources). The critical function can be distributed computing, data/content sharing, communication and collaboration, or platform services. Decentralization may apply to algorithms, data, and meta-data, or to all of them. This does not preclude retaining centralization in some parts of the systems and applications if it meets their requirements.

Plaxton Distributed Search Technique – A randomized lookup algorithm based on prefix matching used to locate objects in a distributed network in *O(log N)* probabilistic time.

Proximity Neighbour Selection (PNS) – Proximity Neighbour Selection (PNS) is a technique used in overlay networks which organizes nodes' internal routing tables according to network proximity. Therefore, one node's neighbours are chosen depending on the latency between them.

Publish/Subscribe Event System – Publish/Subscribe (or pub/sub) is an asynchronous messaging paradigm that allows for better scalability and a more dynamic network topology. Publish/Subscribe is a sibling of the Message Queue paradigm, and is typically one part of a larger Message-Oriented Middleware solution. JMS, for example, supports both the Publish/Subscribe and Message Queue models. In a Publish/Subscribe system, publishers post messages to an intermediary broker and subscribers register subscriptions with that broker. *See MOM, JMS.*

Remote Object Middleware (Distributed Object Middleware) – This middleware consists of software modules that are designed to work together but which reside in multiple computer systems throughout the organization. A program in one machine sends a message to an object in a remote machine to perform some processing. The results are sent back to the calling machine.

Rendezvous Point – The rendezvous point or *root* of a multicast group is the responsible node where all messages are directed when trying to multicast to that group. It can be considered the *root* of the multicast tree. Messages are therefore disseminated efficiently from the rendezvous point to the other members of the tree. In some cases, the rendezvous point cannot even belong to the multicast group itself.

Reusable Component – A unit of software application composition with contractually specified interfaces and explicit context dependencies that can be developed, acquired, added to the system and composed of other independent components, in time and space.

RIAA (Recording Industry Association of America) – The Recording Industry Association of America (or RIAA) is the trade group that represents the recording industry in the United States. Its members consist of a large number of private corporate entities such as record labels and distributors, who create and distribute about 90% of recorded music sold in the US.

RMI (Java Remote Method Invocation) – The Java Remote Method Invocation API, or **Java RMI**, is a Java application programming interface for performing the object equivalent of Remote Procedure Calls. *See RPC.*

RPC (Remote Procedure Call) – Remote Procedure Call (RPC) is a protocol that allows a computer program running on one computer to cause a subroutine on another computer to be executed without the programmer explicitly coding the details for this interaction. When the software in question is written using object-oriented principles, RPC may be referred to as remote invocation or remote method invocation. *See LPC, RMI.*

Secure Hash Algorithm – The SHA (Secure Hash Algorithm) family is a set of related cryptographic hash functions. The most commonly used function in the family, *SHA-1*, is used in a wide variety of popular security applications and protocols, including TLS, SSL, PGP, SSH, S/MIME, and IPSec. SHA-1 is considered to be the successor to MD5, an earlier, widely-used hash function. Both are reportedly compromised. In some circles, it is suggested that SHA-256 or greater be used for critical technology. The SHA algorithms were designed by the National Security Agency (NSA) and published as a US government standard. *See Hash Function.*

SOA (Service-Oriented Architecture) – The term Service-Oriented Architecture (SOA) expresses a perspective of software architecture that defines the use of services to support the requirements of software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation. SOA is usually based on Web services standards that have gained broad industry acceptance. These standards also provide greater interoperability and some protection from lock-in to proprietary vendor software. However, SOA can be implemented using any service-based technology. *See Web Service.*

Stateful – Used to refer to remote objects or components which need to store state information between any of their remote method calls.

Stateless – Used to refer to remote objects or components which do not need to store state information between any of their remote method calls.

Sun RPC (ONC RPC) – ONC RPC, short for *Open Network Computing Remote Procedure Call*, is a widely deployed remote procedure call system. ONC was originally developed by Sun Microsystems as part of their Network File System project, and is sometimes referred to as *Sun ONC* or *Sun RPC. See RPC.*

TCP (Transmission Control Protocol) – The Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite. Using TCP, applications on networked hosts can create *connections* to one another, over which they can exchange data in packets. The protocol guarantees reliable and in-order delivery of data from sender to receiver. TCP also distinguishes data for multiple connections by concurrent applications.TCP supports many of the Internet's most popular application protocols and resulting applications. *See IP.*

TTL (Time to live) – Time to live (TTL) is a limit on the period of time or number of iterations or transmissions that a unit of data (e.g. a network packet) can experience before it should be discarded. In theory, time to live is measured in seconds, although every host that passes the packet must reduce the TTL by at least one unit. In practice, the TTL field is reduced by one on every hop.

UDP (User Datagram Protocol) – The User Datagram Protocol (UDP) is one of the core protocols of the Internet Protocol (IP) suite. Using UDP, programs on networked computers can send short messages sometimes known as *datagrams* to one another. UDP does not provide the reliability and ordering guarantees that TCP does. Datagrams may arrive out of order or go missing without notice. Without the overhead of checking if every packet actually arrived, UDP is faster and more efficient for many lightweight or time-sensitive purposes. *See TCP, IP.*

URI (Uniform Resource Identifier) – A Uniform Resource Identifier (URI) is a compact string of characters used to identify or name a resource. The main purpose of this identification is to enable interaction with representations of the resource over a network, typically the World Wide Web, using specific protocols. URIs are defined in schemes that create a specific syntax and associated protocols. *See WWW.*

WAN (Wide Area Network) – A Wide Area Network (WAN) is a computer network covering a wide geographical area, involving a vast array of computers. The best-known example of a WAN is the Internet. WANs are used to connect local area networks (LANs) together, so that users and computers in one location can communicate with users and computers in other locations. Traditionally, WANs have been implemented using one of two technologies: circuit switching or packet switching. Typical communication links used in WANs are telephone lines, microwave links and satellite channels. *See LAN.*

WAR (Web Archive) – A Web Archive (WAR file) is a ZIP file used to distribute and package the necessary components for a J2EE web application. *See J2EE.*

Web Service – A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface that is described in a machine-processable format such as WSDL. Other systems interact with the Web service in a manner prescribed by its interface using messages, which may be enclosed in a SOAP envelope. These messages are typically conveyed using HTTP, and normally comprise XML in conjunction with other Web-related standards. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. *See HTTP, XML, SOA.*

WWW (World Wide Web) – The World Wide Web (WWW) is a global, read-write information space. Text documents, images, multimedia and many other items of information, referred to as *resources*, are identified by short, unique, global identifiers called Uniform Resource Identifiers (URIs) so that each can be found, accessed and cross-referenced in the simplest possible way. *See URI.*

XML (eXtensible Markup Language) – The eXtensible Markup Language (XML) is a World Wide Web Consortium (W3C) recommended general-purpose markup language for creating special-purpose markup languages capable of describing many different kinds of data. Its primary purpose is to facilitate the sharing of data across different systems, particularly systems connected via the Internet. Languages based on XML are defined in a formal way, allowing programs to modify and validate documents in these languages without prior knowledge of their particular form. Another view is that XML is a wide standard to encode structured information.

XML RPC – XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism. It is a very simple protocol, defining only a handful of data types and commands, and the entire description can be printed on two pages of paper. This is in stark contrast to most RPC systems, in which the standards documents often run into thousands of pages and require considerable software support in order to be used. *See RPC, XML, HTTP.*