UNIVERSITY OF OSLO
Department of Informatics

The design and evaluation of a QuA implementation broker based on peer-to-peer technology

Master thesis

Johannes Oudenstad

12th February 2007

Abstract

In the QuA component based middleware architecture, the implementation broker assists the service planner in service planning by performing resource discovery. Pluggable core services is a key feature in QuA, and the implementation broker role is one of those. However, at the start of this thesis, there was only one component available for this role; the Basic Implementation Broker. The Basic implementation broker is designed to perform resource discovery of local resources. A second implementation should not only be able to share offer space for resources among instances of QuA, for its ability to scale well, self organize and provide robustness to data loss from node failure would allow for a larger field of use for the component.

Peer-to-peer technology has evolved greatly since the rise and fall of Napster, and the scalability, robustness and self-organization properties make peer-to-peer technology a good basis for an architectural model for distributed systems.

This thesis aims to investigate the feasibility of using peer-to-peer technology in QuA resource discovery by designing and implementing an implementation broker component based on peer-to-peer technology. The component is also tested and evaluated in terms of scalability, robustness and ability to self organize a network of peer-to-peer broker components without any centralized control.

The design of the component is only technology generation specific, but the implementation described uses the FreePastry implementation of the Pastry technology. The component is fully operational as an implementation broker component in QuA.

The evaluation of the component show that the component is able to distribute responsibility for query resolution on resources as evenly as the underlying technology permits on participating nodes in a network of peer-to-peer broker components. Further, it is able to re-organize responsibility for resources among participating nodes both in the event of nodes joining and departing from the network. The replication scheme is also proven to be working, and through that robustness to data loss from node failure is also acheived.

Acknowledgements

While working with this thesis has been both fun and inspiring, it is impossible to hide that the hard work has taken its toll, and I would never have pulled through without support.

First of all, I would like to thank my supervisor. Professor Frank Eliassen is a big capacity in the field of distributed systems, and especially middleware. His experienced guidance has been invaluable in my work with this thesis.

I would also like to thank professor Eliassen for keeping his faith in me when times were rough after my cousin and dear friend Tor Runar Berg so tragically passed away January 29. 2006, at the age of 24.

To that end, my friends and family have shown invaluable support through the emotional storm.

The QuA project team consists of a group of friendly and open minded people. I would like to thank the master students, Ph.D. students and post doctoral reserchers involved in the project for making me feel welcome in the project, and for all professional and non-professional discussions and conversations.

Ph.D. student Eli Gjørven deserves a special appreciation. She has provided me with valuable input on the Java implementation of the QuA middleware whenever documentation has been inadequate, and has been a valuable opponent in discussions regarding implementation problems of the peer-to-peer broker related to the QuA middleware.

The student society at Simula is great, and all students at Simula should be proud of the good atmosphere at the student reading hall. Ole Christoffer Apeland and Tommy André Nyquist are very talented individuals and fellow students at the Network and Distributed systems department at Simula. Both of them have been helpful when it comes to use of linux tools.

Further I would like to express my gratitude to former Simula master student, now Ph.D. student, and good friend Martin Alnæs. When the hard drive of my laptop computer broke down, Martin readily offered me one of his laptops on loan as a replacement. Without that loan, I would have been without a computer for over almost two months.

The weekly game evenings with my close friends Tommy André Nyquist, Atle Einarsson, Martin Alnæs, Thomas Levy and Andreas Marienborg have been a good way to get my mind of the work with the thesis, and recharge my batteries.

There are a lot of people out there contributing to open source software. Keep up the good work! Among the tools used in the work with this thesis that are based partially, or completely, on the work of the open source community are the excellent Debian / Gnome based Ubuntu Linux distribution, the Gnome Dia diagramming tool, The Gimp image editing application, LaTeX, Eclipse, and the LaTeXeditor plugin for Eclipse called TeXLipse.

Last, but not least, I would like to thank all of my family, especially my mom, dad and sister for their love and support, and for having patience with me, through the time I have spent working on this thesis. My girlfriend has also shown a lot of patience with me during the last stage of this work. My mind has been 100% focused on this one task, and I have not had time for anything but this work for a long time. Mentally I have been in my own world. I promise that you will see more of me in times to come!

Preface

This thesis is the final part of a five year program for the degree of master of informatics at the University of Oslo. The thesis has been written as part of the QuA project at the Network and Distributed systems group (ND) at Simula with Professor Frank Eliassen as supervisor.

The work of a master thesis is worth 60 credits in the Norwegian study weighing system, which means that it is worth one year of studies. However, to complete a study for a master degree at the Department of Informatics, University of Oslo, one must complete the thesis and 60 credits worth of master level courses within two years.

Contents

Abstract **i**

Acknowledgements **iii**

Preface **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 3
1.2.1 Sub goals . . . 3
1.3 Research method . . . 3
1.4 Scope . . . 6
1.4.1 Type of resource discovery . . . 6
1.4.2 Security . . . 6
1.4.3 Level of abstraction . . . 6
1.4.4 Outdated resources . . . 6
1.5 Summary of results . . . 7
1.6 Thesis structural overview . . . 7

2 Background **9**
2.1 Background study motivation . . . 9
2.2 Middleware . . . 10
2.2.1 Object based middleware . . . 10
2.2.2 Component based middleware . . . 10
2.3 QuA . . . 11
2.3.1 QuA overview . . . 11
2.3.2 QuA capsule . . . 12
2.3.3 Service Mirrors . . . 13
2.3.4 Service planning . . . 14
2.3.5 Pluggable core services. . . . 15
2.3.6 Implementation Broker . . . 15
2.3.7 Implementation broker requirements . . . 17
2.3.8 Summary . . . 17
2.4 Resource discovery . . . 17
2.4.1 Jini . . . 18
2.4.2 ODP trading function . . . 23
2.4.3 Summary . . . 25
2.5 Resource discovery in peer-to-peer systems . . . 26

2.5.1 The Napster legacy 28
2.5.2 Unstructured decentralized overlays 29
2.5.3 Structured decentralized overlays 32
2.5.4 Summary 35
2.6 Pastry walkthrough 36
2.6.1 Structure 36
2.6.2 Routing state 36
2.6.3 Routing messages in the overlay 37
2.6.4 The Pastry API 37
2.6.5 Joining the overlay 38
2.6.6 Self repair and survivability 39
2.6.7 Network locality 39

3 Requirement specification 41
3.1 Functional requirements 41
3.2 Non-functional requirements 42

4 Design of a peer-to-peer broker 45
4.1 Conceptual view 45
4.2 Extending QuA properties 46
4.2.1 Filtering on properties 47
4.3 Mapping service mirrors to nodes 48
4.3.1 Normal configuration 49
4.3.2 Alternative configuration 50
4.3.3 Responsibility and joining and leaving nodes 53
4.3.4 Other P2P-based resource discovery systems 54
4.4 Modelling replication 56
4.5 The layers of the P2P-broker 58
4.5.1 Scenario 59
4.5.2 QuA layer 62
4.5.3 Glue layer 62
4.5.4 Peer-to-peer layer 62
4.5.5 Layer interfaces 63
4.6 Relating the design to requirements 63
4.6.1 Specification conformance 63
4.6.2 Consistency 64
4.6.3 Subtyping 64
4.6.4 Other issues 65

5 Implementation 67
5.1 Choosing technology 67
5.1.1 Peer-to-peer technology 67
5.1.2 Programming language 69
5.2 An overview of the implementation 69
5.2.1 The basic system 71
5.2.2 The message structure 72
5.3 The FreePastry implementation 72
5.3.1 Communicating with FreePastry 73
5.3.2 Bootstrapping the Pastry node 76
5.4 Implementing the broker functionality 77

5.4.1 Configuration parameters . . . 77
5.4.2 Communicating with the OverlayGlue . . . 78
5.4.3 Sending and receiving messages . . . 79
5.4.4 Storing service mirrors . . . 80
5.4.5 Implementing advertiseMirror . . . 80
5.4.6 Implementing getMirrorsFor . . . 81
5.4.7 Serializing service mirrors . . . 82
5.5 Gluing the broker to FreePastry . . . 83
5.5.1 Relaying messages . . . 83
5.5.2 Dispatching incoming messages . . . 83
5.5.3 Reliability of messaging . . . 84
5.6 Implementing replication . . . 85
5.6.1 PAST replication manager overview . . . 85
5.6.2 Replication at the glue layer . . . 86
5.6.3 Why using the basic PAST manager is insufficient . . . 88
5.6.4 The solution . . . 89
5.6.5 Re-querying for resources when response yields 0 service mirrors . . . 90
5.7 Possible improvements . . . 91
5.7.1 Forward method / Caching . . . 91
5.7.2 Difference method . . . 92
5.7.3 Fetching during replication . . . 92

6 Testing the broker **95**
6.1 Test environment . . . 95
6.2 What to test . . . 96
6.3 Distribution test . . . 96
6.3.1 Test setup . . . 97
6.3.2 Expected results . . . 97
6.3.3 Limitations of the test . . . 98
6.4 Joining nodes . . . 98
6.4.1 Test setup . . . 98
6.4.2 Expected results . . . 98
6.5 Departing nodes . . . 98
6.5.1 Test setup . . . 99
6.5.2 Expected results . . . 100
6.5.3 Limitations of the test . . . 100

7 Experiments **101**
7.1 Distribution . . . 101
7.2 Joining nodes . . . 104
7.3 Departing nodes . . . 105

8 Evaluating the broker **115**
8.1 Distribution and scalability . . . 115
8.2 Self-organization . . . 118
8.3 Robustness . . . 121

9 Conclusion and further work **123**
9.1 Conclusion 123
9.2 Further work 124
9.2.1 Design issues 124
9.2.2 Implementation issues 125
9.2.3 Testing issues 125

A Enclosed CD **127**

B Definitions and explanaitions of terms **129**

List of Tables

5.1 Message types in the P2PBroker 79
5.2 Message types related to reliability 83
5.3 Message types related to replication 87

7.1 Distribution of 1000 resources on 2 nodes. 101
7.2 Statistics of the distribution test with 2 nodes experiment. 104
7.3 Statistics of the distribution test with 10 nodes experiment. 104
7.4 Statistics of the distribution test with 300 nodes experiment. . . . 104
7.5 Statistics of the distribution test with 500 nodes experiment. . . . 105

List of Figures

1.1 The workflow of this thesis. 5

2.1 The QuA middleware. Figure taken from [17]. 11
2.2 The Service Mirror. The figure is taken from [18]. 13
2.3 The participants of Jini. 19
2.4 Roles in ODP trading. Figure taken from [6] 23
2.5 Clients and servers. Servers may be clients of other servers. . . 27
2.6 Peer-to-peer architecture. A node knows only of a limited number of other nodes. There is no single centralized entity to control operations. 27
2.7 Architectural overview of the Napster system. The clients of the index server connect to each others in a peer-to-peer way to exchange files. 29
2.8 A node joining a GnuTella 0.4 network. A ping message is flooded through the network via the bootstrap node. 30
2.9 Gnutella 0.6 : Leafnodes are only connected to superpeers. Superpeers are interconnected. 32
2.10 Routing state of a hypothetical Pastry node in a Pastry overlay that uses 16-bit node-IDs. Figure taken from [26]. 37

4.1 Basic concept of the P2P-broker. 46
4.2 Mapping of service mirrors to nodes in normal configuration. . 49
4.3 Mapping of service mirrors to nodes in alternative configuration. 50
4.4 Creation of strands from an attribute / value tree in Twine. Figure taken from [5]. 55
4.5 Replication in the P2P-broker. 56
4.6 Layer responsibility in advertisment example. 58
4.7 Layer responsibility in query example. 60
4.8 Interfaces of the P2P-broker 63

5.1 Overview of the implementaiton. 70
5.2 Overview of the message structure. 72
5.3 Added complexity of distributed search for service mirrors. . . 81
5.4 Added complexity of controlling reliability of FreePastry messaging. 85
5.5 Overview of the FreePastry replication manager architecture . . 86
5.6 The IdMap class 89

5.7 The fundamental changes to the replication architecture 91

7.1 Distribution test. 10 nodes sharing 1000 service mirrors. 102

7.2 Distribution test. 300 nodes sharing 81k service mirrors. 102

7.3 Distribution test. 500 nodes sharing 73k service mirrors. 103

7.4 Tracing of the resources "qua/Repository", "qua/MPEGdecoder" and "qua/Broker" through time where 100 nodes join an initial network of 11 nodes 106

7.5 Tracing of the resources “qua/videoBuffer", “qua/Component1" and “qua/printer" through time where 100 nodes join an initial network of 11 nodes 107

7.6 Retries when asking for resource with key [0x6331...], representing “qua/videoBuffer" while 100 nodes join network. 108

7.7 Retries when asking for resource with key [0xA51C...], representing “qua/Component1" while 100 nodes join network. . . . 109

7.8 Retries when asking for resource with key [0xBC77...], representing “qua/printer" while 100 nodes join network. 110

7.9 Tracing 3 resources in a network of 100 departing nodes, where one node dies every 60 seconds on average. 110

7.10 Retries when asking for resource with key [0xBC77...], representing “qua/printer" with nodes dying every 60 seconds on avg. 111

7.11 Retries when asking for resource with key [0x6331...], representing “qua/videoBuffer" with nodes dying every 60 seconds on avg. 111

7.12 Retries when asking for resource with key [0xA51C...], representing “qua/Component1" with nodes dying every 60 seconds on avg. 112

7.13 Tracing 3 resources in a network of 100 departing nodes, where one node dies every 30 seconds on average. 112

7.14 Retries when asking for resource with key [0xBC77...], representing “qua/printer" with nodes dying every 30 seconds on avg. 113

7.15 Retries when asking for resource with key [0x6331...], representing “qua/videoBuffer" with nodes dying every 30 seconds on avg. 113

7.16 Retries when asking for resource with key [0xA51C...], representing “qua/Component1" with nodes dying every 30 seconds on avg. 114

8.1 Clustering of nodes 117

8.2 Two Pastry nodes share the Id-space of any network evenly . . 118

Chapter 1

Introduction

1.1 Motivation

Peer-to-peer systems have received a lot of attention in research communities in the later years. File-sharing applications like the first Napster system, the GnuTella [7] based systems and others have been highly successful. The success of such peer-to-peer systems are based on a few key aspects of peer-to-peer technology. These are scalability, robustness and their ability to self-organize themselves and in general operate independently of any central entity.

Resource discovery in middleware is usually handled by a central entity acting as a trader of resources. Users or system components that have resources to offer, from now on called resource providers, advertise these to the trader. Users or system components that need a certain type of resource, from now on called resource consumers, query the trader for resources matching a set of requirements. Metadata describing the resources are used in the search for matching resources within the trader.

The Quality of Service Aware Architecture (QuA) [30] is a component based adaptive middleware architecture developed at Simula Research Laboratory (SRL, Simula). In QuA, services are planned and reconfigured by a component called the Service Planner. The Service Planner uses a component called Implementation Broker to find the components needed to build the services, and to find different configurations when reconfiguring the applications. In other words, the Implementation Broker is the trading function in QuA.

Middleware trading functions like the Implementation Broker are more often than not built as client-server architectures [6, 14, 30, 32]. The clients of the trading function are the resource providers and the resource consumers. Known problems with client-server architectures are scalability issues, and the single point of failure problem. In simple terms, a client-server architecture will be sufficient as long as the load on the server is relatively low, and as long as this single-server stays alive. There are ways of building in robustness and scalability in client-server architectures, but these often involve specialized and expensive hardware, and complex maintainance. Peer-to-peer technology has

over the last few years earned a reputation for being able to provide inexpensive ways of building scalable and robust distributed systems, that are able to self organize themselves to a large extent.

In a middleware resource discovery scenario with the trader built as a client-server architecture, the trading service often consists of a single node acting as the trader. In some cases it is possible to connect two or more traders, making them a federation of traders. A problem with this approach has been to find efficient ways of routing queries to the trader instance that actually has the resources, while making sure that all available resources in the federation can be reached.

QuA is a component based middleware with *pluggable core services* intended to be tailored to the context in which the middleware is to be deployed. At the time of starting the work of this thesis, only one implementation of the `implementation broker` core service was available. The "basic implementation broker" is a simple implementation of the role, allowing discovery of services on the local node. Enlargening the selection space of components for this role of the QuA middleware will allow for deployment of QuA in a wider range of contexts.

When QuA is used to run applications, we can usually limit the number of QuA capsules needed, and also the number of resources QuA need to be aware of in order to compose, run and reconfigure the application. However, it is possible to think of scenarios where QuA has to know about a great number of resources. One might imagine a situation where large distributed applications are to be planned, possibly using components written for different platforms, and with various dependencies to physical resources as well. If QuA was to be deployed in that kind of environment, one can imagine that a relatively large number of resources, like capsules with varying properties, different implementations of components and so on are kept within the system. In this case, it would be advantageous to have a reliable and scalable resource discovery scheme.

During the time the author has spent on this thesis, the QuA team has been working on an application called Personal Media Server (PMS) as a tool in their research. PMS is an application built to work as a binding between media streams and receivers that have QoS related restrictions. PMS has the potential of being deployed in an environment of peer users, where potentially everyone can be a video server and everyone can be a video client. In this scenario, a peer-to-peer based resource discovery scheme would be preferable.

Motivated by this, the work presented in this thesis is to investigate the feasibility of using peer-to-peer technology to handle resource discovery in QuA. Further, it will be interesting to see if it is possible to preserve key peer-to-peer properties like scalability, self-organization and robustness when applied in this context.

1.2 Problem statement

This thesis focus on the investigating the feasibility of using peer-to-peer techology to handle resource discovery in QuA. The main problem statement is as follows:

How can peer-to-peer technology be used in resource discovery in the QuA middleware? To what degree can peer-to-peer system's main advantages, scalability, robustness and self-organization, be leveraged in a QuA resource discovery component when deployed in a scenario with a high number of nodes and resources?

The problem statement contains two questions that need to be answered.

Answering the first question implies finding out how resource discovery is done in qua and other middleware, find out what the "state-of-the-art" is when it comes to peer-to-peer technology, and come up with a design and implementation of a QuA resource discovery component. A design of such a component would be an answer to how the technology could be used.

Answering the second question implies a successful design and implementation of a component and testing of that component to see if any of those properties can be leveraged in the given situation.

1.2.1 Sub goals

To help reach the answers to the questions in the problem statement, the following goals were created.

- Investigate the different types of peer-to-peer technology existing today, and find out how this technology can be used to build peer-to-peer resource discovery.
- Find out if it is possible to design and implement a broker based on peer-to-peer technology that will be functionally adequate in the QuA architecture.
- Through development and testing of the prototype, determine to which degree the positive properties of peer-to-peer technology can be preserved in the context of QuA resource discovery.

1.3 Research method

To be able to reach the goals set in section 1.2.1 and through them be able to answer the problem statement, we must decide which research methods to use. The research methods should reflect the goals and problem statement of the thesis.

The academic discipline of computing is rooted in mathematics, the experimental scientific method, and engineering [8]. From each of these roots, separate paradigms, or cultural styles, for approaching computing have emerged. Each of the three paradigms have four steps in the work-process identified by [8]. Further, the article points out that each paradigm follows an iterative workflow, iterating each step.

- The *theory* paradigm is rooted in mathematics. The four steps of the workflow are:
    1. Characterize objects of study. In practice, create definitions.
    2. Create theorems by hypothesizing relationships between the objects.
    3. Determine if the relationships are true, by proofs.
    4. Interpret the results.
- The *abstraction* paradigm, often referred to as *modelling* and *experimentation*, is rooted in the experimental scientific method often used by the natural sciences. The four steps of the workflow are:
    1. Form a hypothesis.
    2. Construct a model and make a prediction.
    3. Design an experiment and collect data.
    4. Analyze results.
- The *design* paradigm is rooted in engineering. In the design paradigm, problems are identified, and systems or devices are created to solve the problem. The four steps of the workflow are:
    1. State requirements.
    2. State specifications.
    3. Design and implement the system.
    4. Test the system.

[8] further points out that which one of these paradigms that is the most fundamental has been basis for much discussion on computing as a discipline. In the opinion of the authors of [8], all paradigms are just as fundamental as any of the others, and in practice they are all tangled up. Elements of any of the paradigms are to be found in the workflow of any of the other paradigms when working in our discipline. Rather, the competence of individuals lies in either of these paradimgs; applied mathematicians are skilled in the theory paradigm, computational scientists are skilled in the abstraction paradigm, and design engineers are skilled in the design paradigm.

In this thesis, we want to find out if it is feasible to develop a resource discovery component for the QuA middleware based on peer-to-peer technology. To do this, the work of this thesis has mostly been within the design paradigm.

To reach our goals set in section 1.2, we use the following research methods:

- In order to come up with a requirement specification for a peer-to-peer implementation broker component for QuA, a time-framed literature study to get a good knowledge of peer-to-peer systems and resource discovery in QuA and other middleware is used.

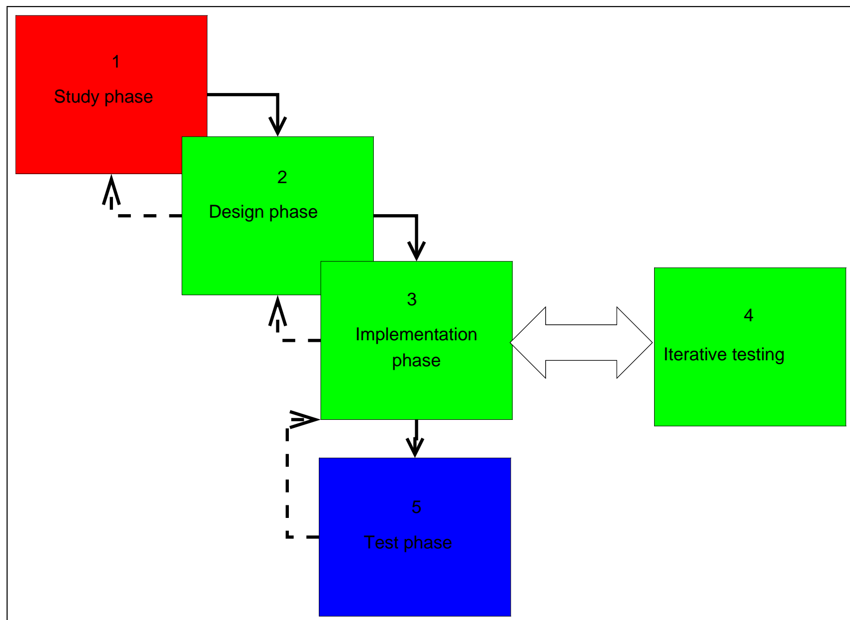

Figure 1.1: The workflow of this thesis.

- Designing and prototyping of a peer-to-peer based implementation broker component for QuA to find out if it is possible to make a functionally adequate QuA broker based on peer-to-peer technology.
- Creating test scenarios and performing experiments with the peer-to-peer broker component to see if any of the main peer-to-peer properties could be kept through the implementation.

Producing a full requirement analysis and a large specification has not been a priority in this thesis. Rather, the literature study is used to make a list of requirements upon which a design for our component can be made. The main priority of this thesis is to find out if an implementation broker component using peer-to-peer technology to share resources can successfully be implemented.

The work of this thesis follows an iterative workflow. Figure 1.1 illustrates the process. The work starts by studying relevant literature to get an overview of the problem. An overview of state-of-the-art technology in the fields of peer-to-peer technology and middleware resource discovery, as well as in QuA, is needed to start working on the design of a peer-to-peer implementation broker component. An iterative process between the study phase and the design phase will take place as relevant literature has to be consulted to come up with solutions to problems in the design. When an early design is ready, the implementation phase will begin. Constant iterative testing is a part of any implementation process, as it provides valuable input to continue the implementation, and possibly reveals problems with the design. If the latter is the case, an iterative process between the design phase and the implementation phase

will occur. When an operational implementation is ready, the work may go on to the testing phase, where we will design tests and perform experiments with the implementation of the peer-to-peer broker. As with the iterative testing, any flaws in the implementation discovered by the testing phase will send us back to the implementation phase.

In the figure, the phase marked with red colour relates to the literature study research method. The green phases are part of the prototyping research method. Likewise, the blue phase uses the test and experiments research method.

Each phase produces some of the text in this thesis. Phase 1 produces the chapters 2 and 3. Phase 2 produces chapter 4. Chapter 5 is the product of phases 3 and 4. Finally, chapters 6, 7 and 8 are the results of phase 5.

1.4 Scope

As the work of a master thesis is done within a limited amount of time, some limitations as to which fields to cover has to be set. The introduction and problem statement gives a direction that the thesis will take, but still there are some grey areas where a borderline has to be set that clarifies what this thesis will and will not cover. This section explicitly defines the scope of the thesis for some areas where the scope may not be entirely obvious. This section does not limit scoping to what is covered here, as the implicit scoping from sections 1.1 and 1.2 should be quite clear for the reader.

1.4.1 Type of resource discovery

The work of this thesis focuses on resource discovery in middleware. More specifically it focuses on resource discovery in *service planning* [30] realized through a particular trading function; the implementation broker role of QuA, a quality of service aware architecture developed at Simula Research Lab (SRL).

1.4.2 Security

Security is a big research field, and studying security in the context of middleware resource discovery could yield a master thesis on its own. Therefore, security has been excluded from the scope of this thesis.

1.4.3 Level of abstraction

As the scoping of type of resource discovery indicates, this research is limited to the middleware layer in level of abstraction.

1.4.4 Outdated resources

A problem with consistency of resources arises where a node in a distributed system advertises a resource that is dependent on that node's participation in the system, and then leaves the distributed system without withdrawing the

resource. This is a general problem with resource registration in distributed systems that relates to use of resources as well. A frequently used element in solutions to this problem is leases, or timeouts, used to clear out "dangling pointers". This thesis will not look into solutions for that problem.

1.5 Summary of results

Through literature study, this thesis has identified which peer-to-peer technologies that have the necessary infrastructure to support resource discovery in the QuA middleware. In short, the requirement that resources are discovered based on resource types, combined with the fact that all advertised resources of a type must be visible to any entity asking for resources of that type, rules out all but the structured peer-to-peer overlay technologies. Further, a design of a P2P-broker component is proposed and the design has been implemented. The implementation unveiled that the requirement for multiple resources to be associated with each key in the structured overlay has not been taken into account by the implementers of the FreePastry implementation of Pastry, or in the design of their replication manager. Finally, experiments show that the P2P-broker is able to preserve the known traits of peer-to-peer technology; scalability, self-organization and robustness.

1.6 Thesis structural overview

This thesis has the following structure;

Chapter 2 presents the necessary background material to understand the work of this thesis, and discusses related work.

Chapter 3 identifies and sums up the requirements for a QuA implementation implementation broker built on peer-to-peer technology, based on the findings in chapter 2.

Chapter 4 presents and discusses the design of the P2P-broker.

Chapter 5 reviewes and discusses the implementation of the P2P-broker that has been realized as part of the work of this thesis.

Chapter 6 discusses how to test the following aspects of the P2P-broker design and implementation; scalability, self-organization and robustness, and presents test-scenarios to cover each field.

Chapter 7 presents the experiments run based on the test-scenarios discussed in chapter 6, and the results from the experiments.

Chapter 8 evaluates the design and implementation of the P2P-broker based on the results from chapter 7.

Chapter 9 concludes and suggests further work.

Appendix A contains a CD with the source code of the implemented component, along with everything needed to start QuA with the implemented component.

Appendix B presents some of the most frequently used terms in this thesis, as well as an explanation of their intended use in the context of this thesis.

Chapter 2

Background

This chapter will describe the most relevant theory to understand the work of this thesis. The chapter is the product of a time-framed literature study.

The structure of this chapter is as follows. Section 2.1 gives a motivation for the choice of areas to study in this background study. It also narrows the scope of the background study. In section 2.2 some important concepts of component middleware in general will briefly be described, before QuA is put under the lens and studied in section 2.3. The focus will be on resource discovery in 2.4 and peer-to-peer technology is studied with focus on resource discovery in section 2.5. Section 2.6 is devoted to the Pastry peer-to-peer technology, as this technology is particularly important to this thesis.

2.1 Background study motivation

The work of this thesis spans over 3 separate research areas. It covers *resource discovery* by the use of *peer-to-peer* technology, through the implementation broker role of *QuA middleware*. This is also reflected in the problem statement of section 1.2.

QuA is a component middleware architecture, but the focus of the work of the QuA project is middleware QoS support through *service planning*. Understanding middleware architectures in general is not required to understand the work of this thesis. However, a short summary of some central concepts used in QuA has been included. A brief overview of the QuA architecture is given, and the way QuA solves resource discovery is explained. Based on this, the scope of the background search on resource discovery schemes in section 2.4 is narrowed.

Resource discovery is one of the two core areas of research in this thesis. This is a big field, and the search for relevant background material was limited to resource discovery approaches similar to that of QuA. In other words type based discovery through a trading function. Some well known trader-based resource discovery schemes are discussed. Other ways of discovering resources are defined to be out of scope of this thesis, as explained in the introduction to section 2.4.

To be able to choose a suitable peer-to-peer technology on which to base the design of peer-to-peer resource discovery in QuA, existing state-of-the-art technologies in that area has to be investigated. Section 2.5 gives an introduction to peer-to-peer technology in general, and describe some of the most influential works in this genre. Through classification of the discussed peer-to-peer systems and based on their properties, some technologies are ruled out as alternatives for the implementation of a peer-to-peer based implementation broker. The structured overlay Pastry will be given most attention as this is the technology chosen in the modelling and implementation of the peer-to-peer QuA broker.

2.2 Middleware

Middleware is a layer of software between the operating system layer and the application layer. The motivation for creating middleware is to mask heterogeneity of underlying networks, hardware, operating systems and programming languages for the application developer. In addition, the middleware provides some sort of uniform programming and computational model for use of programmers when programming distributed applications. Examples include remote method invocation (RMI), remote event notification, remote SQL access and distributed transaction processing [9]. To hide operating system and hardware heterogeneity, middleware provides programming abstractions that are independent of underlying operating systems. RMI and Remote Procedure Call (RPC) implementations mask networking differences, as well as creating location- and access-transparency when developing distributed applications. Interface definition languages (IDLs) provide a way of agreeing upon ways of communicating for parts of the system written in different programming languages.

2.2.1 Object based middleware

The common object request broker architecture (CORBA) is a widely known middleware developed by the object management group (OMG). CORBA provides all of the functionality mentioned above, as well as support for a number of services including security, naming, events and notification [10].

Java RMI is a middleware for invoking methods on remote objects. However, when using Java RMI one is restricted to the use of only one programming language, namely JAVA.

2.2.2 Component based middleware

While Object Oriented Programming (OOP) is a well known programming technique, components and component based programming is more of a technology in the coming. Building applications with components can be seen as building with LEGO. Each component has its own well defined input and output interfaces, and event sources and event sinks. Components are glued together by bindings to create something bigger than the building blocks. In this way, the idea is that reuse of software will be easier, and that the construction of new applications will be faster due to the reuse of components.

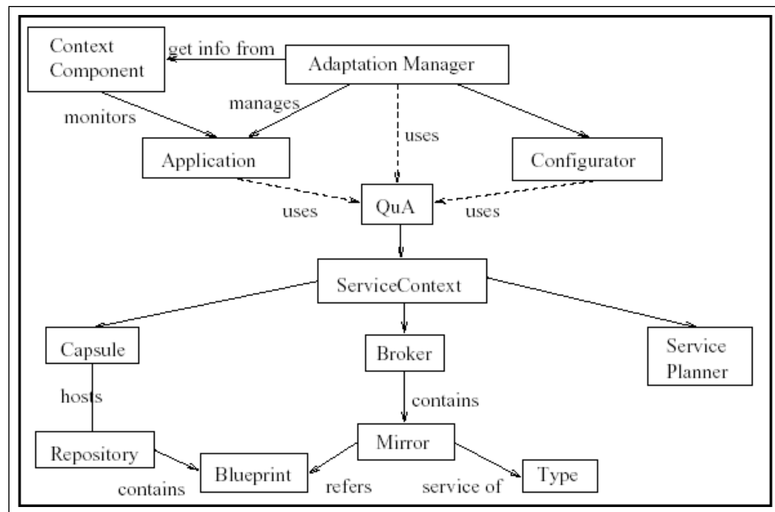

Figure 2.1: The QuA middleware. Figure taken from [17].

Component based middleware is middleware that seeks to make the job of component software developers easier, as well as making software more reliable. The job of software developers are made easier because of the increased possibilities of reuse of code. More reliable software is expected because of the possibilities of handling issues like security, distribution and transactions in the middleware layer. This feature is really a general middleware feature, but it is more visible in object and component based middleware.

The probably best known component technologies are Enterprise Java Beans (EJB), Common Object Model Plus (COM+) and the CORBA component model (CCM).

2.3 QuA

To be able to construct a peer-to-peer based resource discovery component for QuA, we must obtain an understanding of how QuA works, what type of resource discovery that is needed in the QuA middleware, and how a resource discovery component fits into the QuA framework. In this section, we will study QuA with the specific goal of finding out how the service planning phase depends upon stable resource discovery.

2.3.1 QuA overview

QuA is a middleware component architecture developed at Simula Research Laboratory. The established middleware architectures do not have support for QoS sensitive applications. This forces the application developers to program platform-specific knowledge into application components, causing the components themselves to be platform dependent [3]. Platform independence is a main goal for component software architectures. QuA aims at ensuring platform independence even for QoS aware components by building in QoS

support in the middleware architecture. A central part in the approach of the QuA middleware is a process called *service planning*. Clients wishing to start a service need only to specify the logical type of the service, and the quality requirements they have. The middleware platform is responsible for finding and configuring the best possible service of that type in relation to the specified QoS demands.

QuA is designed with a small core and pluggable service components. This allows for easier maintenance of the middleware, and deployment of the middleware on devices with limited resources. Further, QuA is a reflective middleware, realized through a Meta-Object Protocol called QuAMOP [3]. This allows introspection on services and the middleware. The mirror based reflection provided by service mirrors allows for introspection in all phases of the life-cycle of a component [18].

Self-adaptive systems are systems that adapt themselves to changing requirements and environments [18]. Separation of these conserns from the actual application implementations can be done through the use of middleware that supports adaptation. QuA supports self-adapting software through its adaptation manager, and service re-planning. The adaptation manager collects information from a pluggable context component that monitors the application in question. When the adaptation manager decides that change is needed, it calls the Service Planner for a re-planning of the service. Figure 2.1 shows a diagram of the architecture of the QuA middleware.

Evolution of software happens as users find new ways of using it, which leads to the need to evolve the system to cope with the new user requirements. [17] states that software evolution can be divided into two main categories; substitutional and non-substitutional evolution. Substitutional evolution happens when a service evolves into a new service, but its type still conforms to the type of the old service. Non-substitutional evolution is evolution that does not satisfy this property. Both types of evolution is supported by QuA [17].

2.3.2 QuA capsule

`QuA capsules` provide a local run-time environment for instances of the QuA platform. A `QuA capsule` hosts one or more repositories where blueprints reside. `Capsules` may have different capabilities. For instance, one capsule may only be able to run Java components, while another may only be able to run SmallTalk component. Similarly, available physical resources may also vary from capsule to capsule. As mentioned above, some service implementations may have specific requirements to which types of platforms they run on. For the `service planner` to be able to resolve a `service mirror`, it may need to find `capsules` that are capable of hosting certain components. Bindings to QuA capsules can therefore be advertised to the broker as a service. When such dependencies are discovered in the service planning process, the service planner will ask the broker for capsule services that match the components requirements, instruct that capsule to host the component in question, and create the required bindings. In this way, the service planning process is a centralized process capable of creating distributed services.

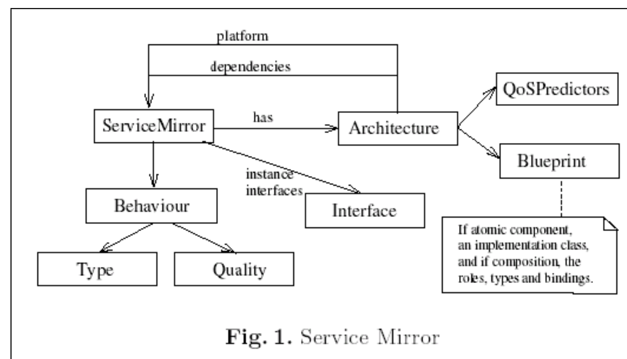

Figure 2.2: The Service Mirror. The figure is taken from [18].

2.3.3 Service Mirrors

In QuA, resources are modelled as services which are described by service mirrors [18]. This description is divided into three parts; a behaviour part, an interface part, and an architecture part. Figure 2.2 shows a conceptual view of a service mirror.

The behaviour part defines the functional and non-functional or *qualitative* behaviour of the service. The type part specifies the functional behaviour and defines a set of *QoS dimensions* that can be used to define the qualitative behaviour of the service. The set of QoS dimensions is also called QoS model. In the behaviour part, qualitative requirements can be expressed in the QoS dimensions defined for the type. In practice this means that the QoS dimensions are tightly coupled with the type, and that qualitative requirements and preferences can be specified differently for each implementation or configuration of the type.

Where the term *component* is used in other component technologies to describe all phases of a component life-cycle; design, deployment and runtime, the QuA component model has a reserved use for this term [3] [30]. In QuA, the term *component type* is what we have at design-time. A *blueprint* is an implementation of a component-type. QuA reserves the term component for the collection of run-time objects created from blueprints.

The architecture part of a service mirror contains a blueprint whose implementation conforms to the type described in the behaviour part, and QoS-predictors that describe the application developer's knowledge of the QoS of the service implementation. Blueprints come in two forms. They can be compositions, which define roles, types and bindings. Blueprints of this type have pointers to service mirrors describing the types that the composition consists of. They can also be atomic, which means that they have no further dependencies. The architecture part also specifies any platform dependencies the blueprint may have. For example, a blueprint consisting of a few Java classes needs a Java platform to run the code. Context dependencies are also specified by the architecture part. A service mirror that does not have its context

dependencies satisfied by the current context, may be discarded in the service planning phase as it is not a feasible implementation of the type in that context.

Finally, the interfaces part contains references to the objects that provide the interfaces specified by the service type, given that the service mirror represents a running service.

As service mirrors are meta-information on services, they are available independently of the state of a service, and this is accessible through the QuA middleware, service mirrors provide mirror-based reflection to services in QuA. The contrasting way to realize reflection on components would be through use of reflective interfaces in the components themselves. This would restrict the use of the reflective facilities to run-time only. With service mirrors, the reflective facilities will be reachable even at pre-runtime, which gives us the opportunity to reason about how non-instantiated services would compare to running services in a given context. This is a prerequisite for service planning which is described next.

2.3.4 Service planning

Service planning is the process where QuA configures a service. Initial configuration happens when a client specifies a service mirror that at least contains a behaviour specification and hands it over to the *service planner* component of QuA. The behaviour specification specifies, as we have seen above, the functional and non-functional aspects of the service. In this case it specifies the QuA-type of the service the client needs, and the QoS demands the client has for the service expressed as a utility function [17] [18]. The task of the service planner in the service planning process is to evaluate alternative implementations that fit the behaviour specification. To find alternative implementations that fit a service description defined in the behaviour part, the service planner connects to an implementation broker and requests the set of mirrors that fit the description. Next, the planner resolves any unresolved dependencies in all candidate mirrors. A mirror is said to be fully resolved if it has an architecture with no dependencies, or it has an architecture whose dependency mirrors are all fully resolved [18]. For each fully resolved mirror, the QoS predictors are calculated. The output of these predictors are used as input for the utility function in the behaviour specification. From the utility function, the service planner derives the utility of each of the mirrors, and ranks them based on the utility. The mirror with the best utility is normally the result of the service planning process.

When the adaptation manager decides that a service needs to adapt, normally to cope with context changes, it calls the service planner for a replanning of the running service. The service planner then performs a new service planning based on the behaviour specification of the running service. If it finds a service configuration that has a better utility in this context than the currently used configuration, the running service will be adapted into the new service. In the case that service evolution has taken place, the new service will be contained in the implementation broker, and will be automatically discovered by the broker and handed over to the service planner for evaluation. In this way, QuA automaticly supports service evolution.

2.3.5 Pluggable core services.

A central goal for the QuA architecture is that it should be able to run on a vast array of different machines and platform setups. To achieve this goal, the QuA architecture relies on, as mentioned above, *pluggable* core services. Central components in the QuA framework such as the `service planner` and the `implementation broker` are pluggable services. A core service can be seen as a role that can be fulfilled by any implementations that conforms to the specification of that role. This gives us the opportunity to pick and choose implementations of these services that best fits the environment in which QuA is to be deployed. In this way, QuA can be tuned to optimize its performance in different environments.

For this to have any effect, however, more than one implementation of each core service must be available. At the time of writing this thesis, only one implementation of the `implementation broker` is available. The implementation in question is known as the `Basic Implementation Broker`, and has only the most basic functionality needed to operate in its role. Some of the motivation for investigating the possibilities of a peer-to-peer based broker service was to enlarge the selection space for the `implementation broker` role. This is also reflected in section 1.1.

2.3.6 Implementation Broker

As stated in sections 1.2 and 1.4, the `implementation broker` role of the QuA middleware will be used to investigate the feasibility of using peer-to-peer technology to do trader-based resource discovery.

The QuA `implementation broker` is a typical *trader-based* discovery service as described in section 2.4. The resources traded in the broker are the service mirrors discussed above. Component developers and application developers alike must advertise the service mirrors describing their composition or blueprint to the broker. The broker has a responsibility of hosting all mirrors advertised to it. When services are planned, all implementations that reside in the `broker` or `brokers` that the `service planner` is connected to, and that conforms to the given type of the service, are considered.

As stated above, QuA can be configured through the use of pluggable core services like the `broker` to tuned to the environment. However it is possible for the `planner` to have connections to multiple brokers, possibly of different implementations, that may have different service offer spaces. For instance, the QuA capsule may host one instance of the `basic implementation broker` to handle resources that only should be able to be discovered locally, and still participate in a peer-to-peer network sharing other resources by hosting an instance of a `P2P broker`.

In the service planning phase, the planner asks the broker for service mirrors matching a type description, and the broker is responsible of returning the service mirrors that match the type. Some non-functional properties may also be specified to the broker, in which case it is supposed to filter out the mirrors that do not fit the description.

The property model of QuA is rather limited. Each service mirror is through the behaviour part associated with a set of properties. Which property types a service mirror supports is dependent upon the type of the service. However, there is not specified any classes of property types, and the value range of a property type must be known in advance for component developers if they are to specify use for them. There is no way of figuring out the value range of properties at run time. In practice this limits the range of options for property filtering, and thus the value of the property model. In the current implementation of the "Basic Implementation broker" component of QuA, the only option for property matching is an "equals" match. The value field for each specified property is matched for equality to see if the property is satisfied.

Any implementation of the `implementation broker` component of the QuA framework has to implement the *implementation broker interface*. That interface has the following methods:

1. `advertiseMirror(ServiceMirror sm)`
2. `removeMirrors(ServiceMirror sm)`
3. `Set serviceMirrors = getMirrorsFor(ServiceMirror sm)`

The first method tells the broker that the caller wants to make a resource available. All the information on that resource is contained in the `service mirror`. Upon receiving the mirror, the broker will categorize it on the service type, or `QuA type` of the mirror, and save the data with later lookup on service mirrors of that type in mind.

The second method tells the broker to remove all resources conforming to the type specified in the `service mirror`. The `service mirror` supplied is only supposed to contain references to the QuA type it wants to remove. Notice that there is no defined way to remove a particular resource, only all conforming to a given service description at once, although it is possible to build that into the desired functionality without changing the interface because a `service mirror` and not a QuA type is the parameter to the method.

The third method is used when an entity asks the broker for service mirrors conforming to a given specification in the form of a service mirror. The provided service mirror contains the type specification of the resources, and a map of properties that need to be satisfied for any resource to fit the resource description.

According to the specification of QuA, the non-functional properties that can be associated with a resource in QuA may be either static or dynamic of nature. If we use a printer modelled as a service and advertised as a service mirror as an example, a static property could be the number of pages per minute the printer is able to print, and a dynamic property could be the length of the printer queue. When the implementation broker service receives a call to the `getMirrorsFor` call with a service mirror that contains a map of properties, it will filter out services based on the type and the static properies. Dynamic properties will be ignored. The `service planner` will however be able to take dynamic properties into account when planning a service. Unfortunately, there is no way of dynamically figuring out if a property is static or dynamic in the

current implementation of QuA. Still, as it is the service planner's responsibility to create the query to send to the implementation broker, the implementation broker does not have to worry as it will be the service planner's fault if any dynamic property is specified. This can be disregarded for the purpose of this thesis.

2.3.7 Implementation broker requirements

As an alternative implementation of the `implementation broker` role of QuA, a peer-to-peer broker has to conform to its specification. To be able to narrow the scope of the background study of comparable resource discovery mechanisms and peer-to-peer technology, some requirements are pointed out here. A more formal requirement specification for the P2P-broker is given in chapter 3.

The broker has to:

- Conform to the interface of the implementation broker role of QuA, given above.
- Be consistent when dealing with resources.
- Be able to filter resources based on a list of properties in the discovery phase of operation.

Furhter, the broker is expected to:

- Share the load of storing `service mirrors` between peers.
- Be relativley lightweight in terms of physical resource consumption.

2.3.8 Summary

This master thesis aims to investigate the feasibility of doing peer-to-peer based resource discovery in QuA. Resource discovery in QuA is realised through the `implementation broker` role, which is specified through an interface. The resource discovery function of QuA, the `broker`, is used by the `service planner` in service planning and replanning. We have identified a few requirements for a QuA resource discovery component that may help us in the search for relevant related work in the field of resource discovery and peer-to-peer systems.

2.4 Resource discovery

In the real world, the term resource can be used about natural resources, mental resources like inner strength or intellectual strength, man made resources such as libraries and services, or resources in form of money and wealth. In the computing world, we often see resources in terms of capabilities of computers and other hardware, or in terms of services performed by pieces of software.

When discussing discovery of resources, a good place to start would be to consider what people do to discover resources, and relate that to use in networking and computing. People's methods for searching for resources can generally be divided into two groups. Since the search for resources often is related to the refinement of the resource and eventually offering a new resource

or service, we will use the following example to clarify the two patterns of resource discovery.

A hunter that needs e.g. a deer resource starts a search for it in what we can describe as all likely places. This has to be considered an active search for resources. The equivalent in networking would be either for the service provider to broadcast its presence so that consumers may find that resource by listening for its packets, or for the consumer to broadcast its resource needs, and for providers to answer.

When the hunter has killed the deer, he can start refining his resource. Even- tually he may have a new resource that he can provide to others, for example meat or hides. As it is unlikely that someone needing his resources suddenly drops by in his lodge, he needs to find a consumer himself. However, he does not want to search the forrest for other cabins and hope for someone to buy his resource. He needs a more efficient way to find a consumer. A marketplace is a place where resources can be traded. Resource providers announce their resources at the marketplace. Consumers travel to the marketplace and browse options to find what they need. In middleware, the traditional approach to the resource discovery problem is to agree upon a sort of common marketplace where resources can be announced and traded. This is often defined as a middleware service, and often called a trading service or trader.

Since this thesis is related to the work on the middleware component architecture QuA, the focus of this section will be on resource discovery in terms of the second approach to resource discovery; resource discovery through a marketplace. The marketplace is often referred to as a trader or a broker. In the search for similar systems to the `implementation broker` role of QuA to present in this section, three aspects were emphasized;

- The system should consist of a trader or trading function.
- The system should trade resources based on type conformance and matching of properties.
- The system should be a middleware service.

In this section, we will take a look at two systems for discovering resources or services, that both match these requirements; Jini and the Open Distributed Processing (ODP) trading function.

2.4.1 Jini

Jini [32] [23] is a distributed system created with the goal of easing the sharing of resources between users. In Jini, a resource may be either a piece of software, a hardware resource, or a combination of both. Enabling participants to add and delete resources flexibly is thought to make a more dynamic network that better reflects the dynamic nature of the workgroup. Jini is tightly coupled with Java. This means that it is platform independent in the same way as Java, but it also means that knowlege of Java is needed to provide and use resources. Services do not have to be programmed in java, but at least java wrapper classes that conform to the interfaces of the resource type implemented must be provided.

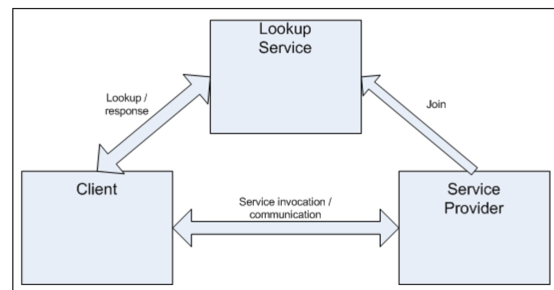

Figure 2.3: The participants of Jini.

Key concepts

We will now take a look at some key concepts used in Jini that are relevant to this discussion of Jini.

Service is the most central concept in Jini. A service is defined as *"an entity that can be used by a person, a program, or another service"* [32]. This is a vague definition, but it means that almost anything can be modelled as a service in Jini. Examples of services mentioned are computation, storage, communication channels, software filters, hardware devices and even other users. Communication with a Jini service happens through what is known as a *service protocol*. The service protocol is constituted by a set of Java interfaces.

Many services in Jini are *leased*. A lease is a time framed agreement of use of the service. When a client needs a particular service, a lease time is negotiated with the service provider. If the lease is not freed up or renewed by either part before the lease time is out, the lease ends, and the service is no longer associated with the client. The key advantage with lease base access is that it provides a mechanism for freeing up allocated resources in the case of network or node failure. This is a commonly used mechanism in distributed computing, often referred to as soft-state.

Java remote method invocation, or Java-RMI, is a central part of the Jini infrastructure. It allows for mobility of code, which is an important feature that makes Jini easier to use. This will be further explained later in this subsection.

Security is defined to be out of scope in this thesis. However, the obeservant reader may notice that the mobility of code implies a serious security threat. This is handled by Jini through access control lists and the notion of a principal that traces back to a particular user of the system [32]. Interested readers are encouraged to find out more by reading the relevant Jini specifications.

Architectural overview

Figure 2.3 shows the architecture of Jini. Participants in a Jini network can take any of the roles shown in the figure. A participant can have more than one role

at the same time. It is actually possible to operate all three roles at the same time, but this is a special case.

The central part of the architecture is the lookup service. It acts as a trader of services. Clients connect to a lookup service to find services it needs. This process is supported through the lookup protocol. Service providers connect to a lookup service to advertise its services so that others can use it. This process is called joining, and is supported through the join protocol. The lookup service itself is discovered through the discovery protocol. As figure 2.3 shows, once a client has found a service, the client communicates directly with the service provider.

Discovering the lookup service

For a Jini entity to enter a Jini network, it must find a lookup service. The lookup service acts as a trader in the Jini system. To find a lookup service, entities need to use the discovery protocol[1]. There are three parts of the discovery protocol [33].

- When a lookup service wishes to announce its arrival in the network, it will use a multicast announcement protocol to mark its existence. Clients and service providers are listening to such messages and may respond with a request to connect to the lookup service.

- When a client or a service provider wants to connect to a lookup service that they know of in advance, they may use the unicast discovery protocol. Lookup services are running a TCP server and waiting for connections from clients and service providers. When connecting to the lookup service with this protocol, the entities connecting to it must specify a hostname and a port number. If the connection is accepted, the lookup service will reply by sending a service registrar object. This is used to communicate with the lookup service.

- Clients and service providers that do not know of any lookup services, or wants to find additional ones, use the multicast request protocol. This protocol consists of four steps. First, the discovering entities set up a TCP-server that accepts references to lookup-services. This is called a multicast response service. Second, lookup services listen for requests for references to lookup services. Listening entities are said to be instances of the multicast request service. Third, the discovering entity multicasts a message requesting references to lookup services. Fourth, and finally, the lookup services respond to the discovering entity, and by means of the unicast discovery protocol transmits a service registrar object through which the discovering service can communicate with the lookup service.

[1]Through this protocol, Jini is able to operate even when the participating entities do not know of a lookup service in advance. In other words, it allows participants to dynamically discover the trading service. This gives the Jini system a great deal of flexibility, and is cause for some [11] to categorize it as a system for resource discovery in ubiquitous computing. However, as we will se next, once connection to the lookup service has been established, services are traded in the same way as in a trading function in object or component based middleware.

Joining with a service

When a service provider wants to provide a service to other users, it has to register it with a lookup service. This process is known as joining the lookup service. The service provided consists mainly of three parts. One part is the service backend. The backend is located at the service provider. A second part is known as the service object. This object represents the service at the clients that use the service, and implements the set of Java interfaces that constitutes the type of the service. The service object may be a plain proxy-object, forwarding all requests to the service backend, but in some cases, the service object contains the whole service, so that little or no communication is needed with the service backend [23]. Service objects may also be implemented as a combination of these. Alternatively, the service object may be replaced by a RMI stub if the service is implemented as an RMI remote object [34]. The third part of provided services is a set of non-functional attributes describing the service, known as service attributes. The service attributes are used in the lookup service to find services that match a client's requests during a lookup.

To register a service, the service provider creates a serviceItem object that consists of the service object and the service attribute list. The service provider uses the service registrar object received from the lookup service in the discovery phase to communicate with the lookup service, and register the service by calling a Register(ServiceItem item, Long leaseDuration) call. The leaseDuration describes how long the provider wants the service to be available. The lookup service returns the actual lease time set. Leases can be renewed, and Jini supports lease managers that can help with this. If this is the first time the service is registered, the service also gets a serviceID. This is a universally unique identifier, and the service provider has to remember this. The next time the service is registered, the service provider has to specify this to the lookup service.

When methods in the service object are invoked that are not exclusively implemented in the service object, method calls are forwarded to the service backend. This can be done by using Java-RMI, or the Jini Extensible RMI (JERI), or by any other way of communication. What is used is left to the service implementor to decide. This means that only the service object and the service attributes need to be implemented in Java, and allows some flexibility with regard to e.g. legacy systems.

Looking up a service

Clients needing a particular service will contact the lookup service with a request for a service matching a specification. In the discovery phase, the client received a service registrar object. The client will use this to contact the service provider, and invoke a method Lookup(ServiceTemplate template). ServiceTemplate objects describe the service wanted by the client. It has three parts, of which none are mandatory [23]:

1. A serviceID.
2. A set of interfaces that has to be implemented by the service.

3. A set of attributes that has to be supported by the service.

The serviceID field is used when the client has used a service before, knows its particular serviceID, and wants to use it again.

The lookup service uses the ServiceTemplate object to find the service that best fits the category defined. The service object for that service is returned to the client. Alternatively, the lookup service can return all services that match the requirements, and let the user decide which one to use.

A central feature in Java-RMI that Jini uses extensively is the ability to copy not only data, but objects and classes over the network. This makes life easier for the clients, as to them, all method invocations always appear to be local, even for services they have dynamically discovered. All services in Jini are represented locally at the user by a proxy object, be it a service object or a service registrar object [23]. This also means that the client does not need to know the communication protocol to talk to the service backend. This knowledge is included in the proxy object, which is a Java object that resides in the local runtime environment at the client.

Users or administrators may also register with the lookup service as listeners to events, where the event is that a new service of some type has arrived [34].

Clarification on entry objects

In Jini, properties of a service are specified by entry objects. The purpose of entry objects is twofold:

- The lookup service uses them to filter service based on matching of entry objects.
- Human users use them to identify which service they want to use in the case where more than one service matches a service description.

Entry objects are objects of classes that implement the `Entry` interface. Each class implementing the `Entry` interface can be seen as a property type. When someone is doing a lookup for a service, they can specify a set of entry objects representing properties that the service must satisfy. When the lookup service is matching these attributes against attributes of candidate service offers, they match each attribute by doing an object equality check. That is, the Java `equals` method is not used. Instead, objects are matched for equality in serialized form. This means that the only check that can be performed at the lookup service is an equality check. If an offered service matches a property type with more than one value, it has to specify one entry object of the correct class for each matching value. Likewise, as pointed out in [23], if the value range is not agreed upon by all, the service provider must create an entry object specifying that it matches each possible way of specifying a value, as there is no standardized way in Jini to agree upon the value range for attribute types. For example, if a printer is localized in room "B331", but that room is known by some as the "coffee maker room", the service provider has to specify an entry object for each of the possible names for the room.

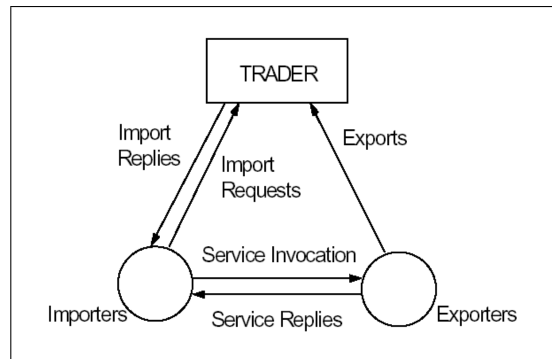

Figure 2.4: Roles in ODP trading. Figure taken from [6]

2.4.2 ODP trading function

Use of the ODP trading function [6] is the way to discover services in the ODP-Reference Model (ODP-RM). The motivation for the trading function is to provide a way to utilise services over *"heterogeneous software, heterogeneous hardware platforms and heterogeneous network environments"* in a *"distribution transparent"* way [6]. It is also argued that the the ever-changing nature of sites and applications in large distributed systems calls for the need to allow *late binding* between service consumers and providers. Late binding in this context means binding at run time, as opposed to binding pre run time. To allow late binding, software components must have a way of finding service providers dynamically, and a trading function will provide this.

Functional overview

An object implementing the trading function is called a *trader* [6]. Figure 2.4 shows how trading with an ODP trader works. Service *exporters* advertise service offers to the trader, which accepts them and keeps the service offer information. The information contained in a service offer include characteristics on the service, and the location of an interface at which the service is available. When a service *importer* needs a service, it sends a *service request* to the trader. The service request contains a set of requirements for the wanted service. Upon receiving a service request, the trader searches its service database for services that match the service request. A list of services that match the request is returned to the importer.

The specification of the ODP trader has been developed with that in mind that it should be able to operate in many different environments. In some cases, the trader will have to hold services for a large population of users, and in other cases the number of users and potential services will be smaller, but the available physical resources might be limited, so that the trader needs to have a small footprint [6]. In support of traders with limited resources, ODP traders are allowed to form *federations* [6].

[6] further points out that in order to prevent scalability issues, too much information about other parts of a distributed system should not be held in one part of the system. The reason for this is both the cost alone of holding

the amounts of data that could amass if no effort to limit it was taken, but also that the cost of keeping such information up to date would be an impossible task. Loading more small traders that cooperate in federations, where traders only are responsible for services close to the trader is assumed to solve the problem. The partitioning of the service space is the key to scalability according to [6]. However, there will still be a problem with a lot of messages being passed between individual traders, as a trader will never know which other trader has a particular service advertised to it, and a query for a service may propagate through all traders in a federation before the service is found. This will put a lot of stress on the network. [35] addresses this problem to some extent. Their answer is to let all traders know the number of services kept at all neighbouring traders and always passing a query on to the trader with the highest number of services first, assuming that the probability of finding it there is higher. The work of [35] is done with the Object Management Groug (OMG) CORBA trader, but the systems are quite similar. [6] states that the ODP trader is in "technical alignment" with the OMG *trading object service*.

Importers and exporters communicating with a federation of traders only ever interacts with one trader. Access to the other traders in the federation is done transparently to the user via communication between the federated traders.

Services in ODP

A service is in [6] defined as " *a function provided by an object at a computational interface*". Further it is a set of capabilities available at that object. Those capabilities can either be a single atomic operation, a set of operations, a sequence of operations or non-operational (e.g. provide or consume a stream of data) [6]. All services are instances of a *service type*. Further, each service type has an *interface signature type* associated with it. The interface signature type determines the computational behaviour of the service. Service implementations that are of the same interface signature type have the same operational signature, but may differ in some non-behavioural aspects. To be able to express such aspects, *service properties* can be associated with a service. Each service property has a *service property type* that it conforms to. A service property type consists of an identifying name, a property value type and a mode that specifies whether the property is mandatory or optional, read-only or modifiable. This means that what makes up the signature of a service type is the combination of interface signature type and the set of service property types.

Service offers

As mentioned above, when a service exporter is advertising a service to the trader, it creates and sends a service offer. The service offer consists of three parts.

- The service type identifying the type of service offered.
- The interface to which a binding can be established.
- And third, property values for the properties contained in the property type set of the service type.

Not all properties need a value, only those that are marked as mandatory. The trader assigns unique identifiers to each service offer.

The trader also supports *proxy offers*. The difference between a proxy offer and a regular service offer is that instead of containing an interface at which the importer may bind to the service, it contains an interface to which the trader forwards the import request to obtain the offer interface [6].

Service requests

Importers create service requests and send them to the trader when they need a service. A service request consists of the required service type and a set of required property values. These are seen as the matching criteria for the service. Additionally, a service request may contain preference requirements for the trader to take into account if multiple services match the request minimum requirements, some scoping information on the search, capability requirements of the importer and a request for values of service properties in case more than one service offer is returned to the importer for it to choose from. The trader uses this information and match it to the service offers advertised to it to find appropriate services.

2.4.3 Summary

We have identified two systems for resource discovery that are similar to the approach in QuA. Jini proves to be able to operate in the role of resource discovery in ubiquitous systems as well, as it has mechanisms for discovering the trading function dynamically. Once a binding has been established to the lookup service, it operates in a similar way to that of the `QuA broker`. We have also seen that it is possible to specify locations of lookup services in advance so that the need to dynamically discover the trading service disappears. The ODP-trader is part of a middleware framework and operates similarly to the `QuA broker`. We have seen that the way resource discovery is handled is to create a description of the resource, or service, and advertise these descriptions to a central entity where lookups for resources matching a specific resource or service type can be made by resource consumers. Note that the actual resource is seldom moved, except for in rare circumstances like Jini services that are small programs capable of running independent of any communication with the service provider can be a direct part of a service description in Jini. Although these resource descriptions are not actual resources, we can sometimes see them spoken of as resources, probably due to the fact that an item making it possible of using a resource can be seen as a resource itself.

The main difference between the systems is the way resources are modelled. In QuA, they are modelled as `service mirrors`. In Jini, resources are modelled as `service items`. In ODP, resources are modelled as `service offers`. The differences here are mainly caused by differences in the middleware architectures of the systems.

Properties are also modelled differently. QuA only has a mapping of property type and values where the QuA type of a service define the set of property

types. The ODP trader also has a tight coupling of the set of property types beloning to a service type. A property type consists of a name, a value type, and a mandatory / optional flag. Jini attributes are specified by entry objects; Java objects that conform to an entry interface.

The possibilities for matching properties are small in QuA and Jini. Both these systems only support equality matching of properties. In addition, the value range of properties can be difficult to agree upon for service providers and service consumers, as there is no mapping between a property type and a value range. The ODP trader has such a mapping via the value type, and has therefore the possibility of supporting more advanced matching mechanisms.

Finally we have seen that none of these systems emphasize scalability to large number of users, nor survivability. Jini provides self-organisation to a certain extent by allowing dynamical discovery of traders. Both systems have functionality for traders to work in some sort of federation, and this could help scalability issues to some extent, but will not help in the case of survivability. These are the issues that peer-to-peer systems address, and that we will try to address in making a peer-to-peer based `implementation broker` for QuA.

2.5 Resource discovery in peer-to-peer systems

System architectures are ways of dividing responsibilities between entities participating in distributed systems. Probably the most well known way to organize a distributed system is to build a client-server architecture. In client-server architectures, clients are the masters, or requesters in the communication protocols. The servers are the slaves, and replies to the requests of the clients. In simple terms, the client is the active party in communications, and the server is the passive party. An example of a client-server architecture is shown in figure 2.5.

As the users of a client-server architected system grows, the load on the servers grows in equal proportion. The servers will have to be hosted on computers with extreme capacity. In global-scale cases, variants of the client-server architecture with multiple servers working in "teams" will be needed. This can prove to be complicated to configure for system administrators. In addition, system resources may be swallowed by overhead cost in protocols needed to keep server state consistent. Examples and more details on different client-server architectures are found in [12].

As opposed to client-server architectures are peer-to-peer architectures. In peer-to-peer architectures, all system entities share an equal amount of responsibility for the system to operate. The basic concept is that all participants in the system are peers in all respects. They will perform the same job in any given situation. By creating distributed algorithms one can construct systems where the peers work together to do certain jobs without being reliant upon any central servers to coordinate their actions. Figure 2.6 shows the topology of a peer-to-peer network.

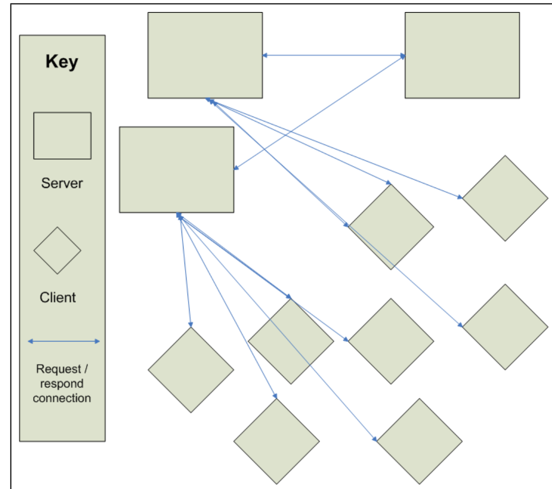

Figure 2.5: Clients and servers. Servers may be clients of other servers.

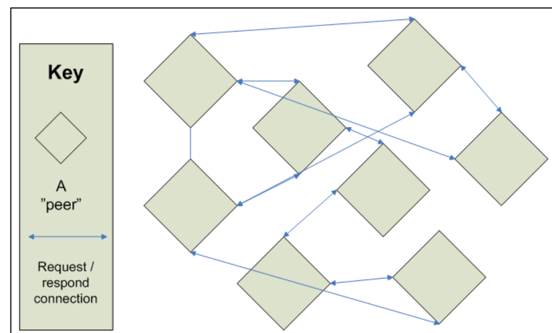

Figure 2.6: Peer-to-peer architecture. A node knows only of a limited number of other nodes. There is no single centralized entity to control operations.

The load on the resource discovery mechanism in a distributed system will grow with the number of resources offered in the system, and the number of resources offered is likely to grow with the number of connected users in the system. Resource discovery mechanisms in peer-to-peer networks aim to distribute the resources and thereby the load of the system on all participating entities. This is what makes it interesting to investigate the impact of peer-to-peer technology in the setting of QuA resource discovery.

Categorizing P2P systems

In the literature, there exists several ways of categorizing peer-to-peer systems. For example, [13] defines three categories, 1st Generation, 2nd Generation and 3rd Generation peer-to-peer systems. Here Napster is an example of a first generation system, Gnutella and Bit Torrent are examples of the second generation, and examples of the third generation would be Pastry and Tapestry. [13] also defines third generation peer-to-peer systems as peer-to-peer middleware as they offer a generalized API which application developers can build applications upon. [21] categorizes P2P systems in unstructured and structured P2P overlay networks[2]. Here, the 1st and 2nd generation systems from [13] roughly fit into unstructured overlays, and 3rd generation systems from [13] fit into structured overlays. There are other ways of categorizing these systems. This chapter roughly follows [21], but sees Napster as more of a special case as it is not truly a peer-to-peer system.

2.5.1 The Napster legacy

File-sharing applications is the "killer app" that put the spotlight on peer-to-peer systems and peer-to-peer technology. Although it was not a true peer-to-peer system, the creators of Napster used peer-to-peer technology to make an affordable way sharing billions of files from millions of users. Unfortunately for its creators, the success of this killer-app also meant the death of Napster, as lawyers of the music and movie industry in U.S.A. took Napster to the courthouse and won a big victory. Napster had to shut down the servers that kept track of what each of the peers in the system had to offer.

Napster's architecture consisted of a number of peer nodes, which were the world-wide users of the system, and a few centralized servers. The job of the servers was only to keep track of mappings between connected peers and their resources. This resulted in a simple resource discovery scheme. Resource providers submitted a list of resources to the server on start-up of the napster program. Whenever a consumer searched for a resource, they connected to a server and submitted a query. The server responded with a list of possible matches, and a list of locations the consumer could contact to get the resource. The server did not hold any resources itself, and all downloads of resources happened between the peers. Figure 2.7 shows an overview of the Napster architecture.

Napsters resource discovery scheme was not decentralized, but the actual exchange of resources happened directly between peers. The success of Napster had brought great attention to the peer-to-peer field. And the fact that the

[2]An overlay network is a network of logical links built upon an existing physical or logical network. For example, peer-to-peer networks are usually overlays on top of the internet.

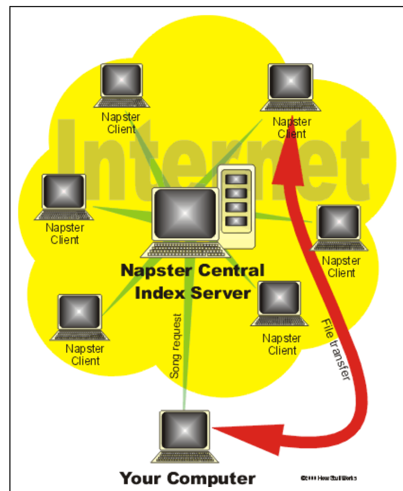

Figure 2.7: Architectural overview of the Napster system. The clients of the index server connect to each others in a peer-to-peer way to exchange files. The illustration is found on the internet.

only centralized entity in the system, the index server, was what brought Napster down in court meant that both researchers, the open source community and commercial actors pushed to completely decentralized solutions.

2.5.2 Unstructured decentralized overlays

Unstructured decentralized peer-to-peer overlays can be divided into two main groups. These are *pure peer-to-peer*, and *hybrid peer-to-peer* systems. Examples of pure P2P protocols are the early Gnutella 0.4 protocol [7], and Freenet. Examples of hybrid P2P protocols are the improved GnuTella 0.6 [29] protocol, the FastTrack protocol used in e.g. KaaZaa, and the eDonkey protocol.

Pure P2P

As an example of pure peer-to-peer protocols, we will take a closer look at the GnuTella 0.4 protocol. In this protocol, the nodes are organized in a flat structure. All nodes know a limited number of other nodes at any given time. At startup, a joining node has to find one initial node to connect to. This is often referred to as a bootstrapping node. This could be entered into the program manually, but there also existed[3] servers that kept a cache of currently connected nodes in the network. When a joining node has found a bootstrapping node, it sends a ping message to this node. The bootstrapping node forwards that message to all the nodes it has links to. This flooding of the message kept on until the time to live field (TTL) of the message had counted down. Each node receiving a ping message responds with a pong message stating whether or not the connection is accepted. The pong messages are routed back to the originator of the ping message via the same path through the use of globally

[3]To the author's knowledge, no Gnutella 0.4 networks are any longer operative.

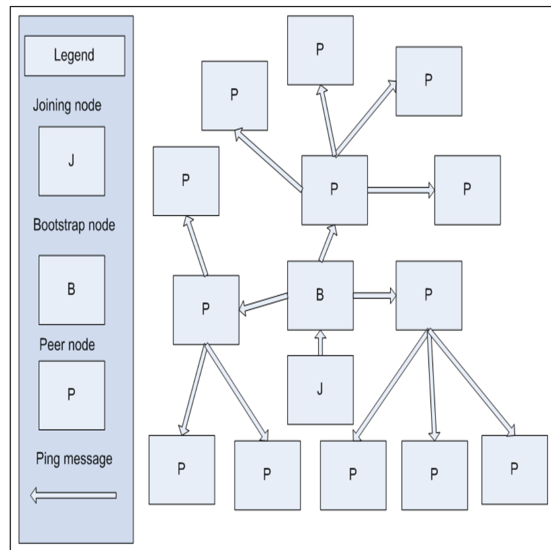

Figure 2.8: A node joining a GnuTella 0.4 network. A ping message is flooded through the network via the bootstrap node.

unique identifiers (GUIDs). 2.8 shows the flooding of a ping message in a small area of a gnutella 0.4 network.

To discover resources in GnuTella 0.4, one has to send a **query** message. This message is flooded from the message originator to all links. Every node that receives the message sends it to each of their links and so on if the hop count has not reached the TTL for the message, and the current node has not received this exact message before. The latter is to avoid loops. Each receiving node also checks if it has the resources queried for, and if it does, it sends a **query hit** message back to the **query** message originator. This message follows the same path the **query** message took, just like the ping-pong message scheme. When the querying node receives a **query hit** message, it can choose to set up a direct connection to the node that sent the response message.

The problem with the Gnutella 0.4 approach is that even though it is completely decentralized, and totally independent of central entities, it scales rather poorly. The reason for this is the node discovery (ping-pong) and resource discovery (query- query hit) messaging schemes where **ping** and **query** messages are flooded in the network. This results in each message duplicating into an extreme amount of messages as it flows through the network. GnuTella tries to control this by using a TTL field in the messages. But if each node has, say, 4 connections on average, even with a TTL of 5, each message will multiply into 484 messages. [25] is a widely cited paper that contains a mathematical proof on why gnutella cannot scale. The TTL field, and number of connections per node has to be set to low values to keep the number of messages in a system at a low level. However this presents a new problem, as it does not allow for a node to “see” all other nodes in the network. This effectively partitions the

network. Partitioning of the network in this way is not a big problem when searching for "popular" resources, as these probably are located at that many nodes around the network that the probability of not finding them because of the low TTL is negligible. When searching for less popular resources, the probability of finding them can be quite low. Anyhow, no guarantees can be made as to finding resources existing in the system.

Hybrid P2P

To improve on the GnuTella 0.4 protocol, a number of measures were taken in GnuTella 0.6. The most important was to introduce the concept of *UltraPeers* [29] [21] [4]. The introduction of UltraPeers changes the overlay topology to what shown in figure 2.9. The Gnutella 0.6 protocol differentiate between nodes running on machines with low hardware capacity and nodes running on machines with a high hardware capacity. If you have a high bandwidth, a good processor and enough RAM, you will be assigned the responsibilities of an UltraPeer. A change of the querying protocol was also made. In version 0.6, leaf nodes will forward an index of the files they provide to the ultrapeer they are connected to. When a leaf node is searching for resources, a query is submitted to its ultrapeer which floods the query among the other ultrapeers it is connected to, in the same way it was done in the 0.4 protocol. The difference is that only ultrapeers are involved in the flooding. This allows for a larger proportion of the network to be "visible" to each peer, as each ultrapeer takes responsibility of acting as a proxy [21] for its leaf nodes.

Unfortunately, the measures taken in Gnutella 0.6 are not enough. They are improvements over the former protocol in terms of scalability, but scalability is still an issue, and the problem with partitioning of the offer space remains.

The introduction of ultrapeers in Gnutella may have been encouraged by the *supernodes* introduced in the eDonkey and KaZaa / Fasttrack networks. As the fasttrack network is proprietary and good documentation on its structure and operation does not exist publicly [4], we can not go into detail. But through use of the system, it is a common understanding that it has a *gnutella 0.6 - like* structure, with superpeers, and that messages are flooded among the superpeers in the resource discovery phase of operation. The eDonkey actually has a separate program for its superpeers [19]. Unlike the superpeers of Fasttrack and ultrapeers of Gnutella 0.6, these are more of dedicated servers and do not share any files themselves. Apart from that the operation is similar to that of Gnutella 0.6 and Fasttrack.

Optimizations

A number of optimization attempts to the flooding based protocols have been attempted. [22] describes a couple of techniques and evaluates them. *Expanding ring* searches is a way to control the TTL field better when searching for resources such as files. The idea is to start with a TTL value of e.g. one, and submit a query. In the case of no response, the TTL is increased to two, and the query is resubmitted. This continues until a response is received, or such a high TTL value is reached that the system gives up on the query. This is assumed to work because most of the things users of these systems search for are popular

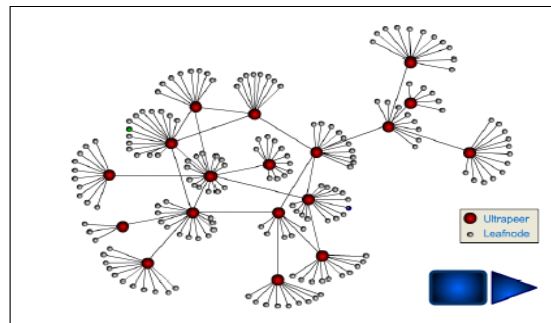

Figure 2.9: Gnutella 0.6 : Leafnodes are only connected to superpeers. Superpeers are interconnected. The figure is taken from the lecture notes of the course inf5090 at the university of Oslo.

content, and assumed to be found within a small ring of hops around the user. *Random walks* denotes a way of searching where the message is not flooded on all outgoing links at each node that receives the message, but sent along one or a few random links at each node. This reduces the number of messages sent at each level, and the idea is that it should increase the visibility of nodes in the network.

Again, this is a further improvement in the way these protocols operate, but it is not enough. So it seems that with these types of architectures, the problem of scalability and partitioning of the offer space can not be solved.

Summary

The systems described in this subsection are self-organizing and largely decentralized in their operation. However, the systems ability to scale involves a trade-off against visibility of resources kept within the system. This makes these types of systems largely unusable for resource discovery in middleware such as QuA. When the middleware asks for resources of a type, the underlying peer-to-peer network must be able to locate all resources of that type that the network holds. These systems can give no such guarantee. In addition, replication in these systems will be extremely hard to build and control as there is no way to have any idea of how many copies of a resource that are in the network at a given time.

2.5.3 Structured decentralized overlays

Structured decentralized overlays introduce a structured approach to organizing the nodes in the overlay, as well as to object or resource placement, and message routing. In this way they make a trade-off, trading some flexibility for better scalability and guarantees on node and resource visibility[4]. In theory, all nodes and thus all resources can be reached in the structured decentralized overlays presented here.

[4]As opposed to the unstructured overlays that sacrifice visibility and scalability for increased flexibility.

This section will present a few widely known structured overlays. A short overview of Pastry will be given here, but since this technology is particularly relevant to this thesis, the interrested reader may read section 2.6 for a walk-through of the most relevant concepts of Pastry.

Pastry: a quick overview

In a Pastry network every node is assigned a unique node-ID in a circular ID space. IDs are to be handed out randomly through use of a distribution algorithm that ensures a uniform distribution in the ID space. Examples on how this can be done include computing a cryptographic hash over a nodes IP-address or over its public key. When presented a message and a key, Pastry routes the message to the node with ID numerically closest to key among live nodes. Pastry takes network locality into account when building up routing tables. Routing distance relative to the chosen locality metric should therefore be relatively small. Pastry nodes keep track of their nearest neighbours in the nodeID space. Since these IDs are randomly distributed, nodes with close IDs should be diverse in geography, ownership, jurisdiction, and network locality. Applications may use this to increase resiliency to network failures, node breakdowns and similar unexpected events. Pastry routes messages to the node that has ID numerically closest to the message key. Because of the way routing is performed and the way routing tables are constructed, the message will with high probability first reach the node that is closest to where the message originated according to the proximity metric. In the routing scheme, IDs and keys are treated as sequences of digits with base $\mathcal{B}$ where $\mathcal{B} = 2^b$ and b is a configurable parameter with typical value of 4 [21]. For every hop a message takes, the routing algorithm brings the message to a node that has a prefix that shares one more digit in its nodeID with the prefix of the key than the previous node did, or to a node that shares the prefix length but is numerically closer to the key. This means that the algorithm will always converge.

Tapestry

Tapestry [36] is, like Pastry, a structured overlay based on the Plaxton mesh structure [21]. Nodes are assigned nodeIDs from a 160 bit circular identifier space through a function that gives random IDs distributed uniformly in the ID space. IDs are treated in routing as sequences of digits in base $\mathcal{B}$ where $\mathcal{B}$ has a typical value of 16. Each node has a routing table, called neighbour table, which consists of several levels of maps. Starting at level n = 0, the entries of the nth level map are nodeIDs of other nodes, with a prefix that matches the local nodes prefix in the n first digits. In each map level, there are B entries. One for each possible digit after the map-level (n) digits that match the local nodeID. Using the increasing prefix match routing with these routing tables, a message will reach its destination node in at most $log_{\mathcal{B}}N$ hops in the overlay if the neighbour maps are consistent. To improve resistance to node failures, there are backup links at each map level. Each node also stores reverse pointers to each node having links to them.

Tapestry offers an API upon which applications can be built. What is interesting is that the API is completely different from that of Pastry, even though the systems have very similar structures and properties. There is no room to

go into detail here, and details can be found in [36]. Basically, Tapestry offers a method to publish an object based on a Globally Unique Identifier (GUID). The object itself will reside on the publishing node, but pointers to it will be placed along the path to its surrogate root, which is the node with nodeID closest to the objects GUID in the identifier space. Messages can be routed to that object by calling the routeToObject method with the objects GUID and applications applicationID as parameters. It is also possible for multiple nodes replicating an object to publish the object. The surrogate root node of the object, and some nodes on the way, will then have pointers to multiple homes of the object. The pointer information is soft-state, so that publishing nodes have to republish at certain intervals.

Tapestry is also self organizing, and handles changes in neighbour tables at joining of new nodes and departure of nodes automaticly. Nodes use periodic beacons on outgoing links to detect node failures.

CAN

The Content Addressable Network (CAN) [24] is a structured peer-to-peer overlay that provides Distributed Hash Table functionality. CAN constructs a d-dimensional coordinate space. This coordinate space is at any given time dynamically divided into zones. Each node "owns" a zone. When a node joins the network, it will contact a node and split that nodes zone into two. The joining node will from that point own one half, and the node that previously owned the whole zone will own the other half. When using this system in a resource discovery scenario, each resource will map onto a single point in the coordinate space. This point will at any time lie within a distinct zone owned by a particular node. Thus, a node is always responsible for the resources that map to points in the coordinate space that lies within its own zone. As the coordinate space is divided dynamically between live nodes in the network, the load of sharing resources is roughly evenly distributed between live nodes.

When determining where a pair of $< key, value >$ should be stored, a hashing function is applied to the key, yielding a point in the coordinate space. The node that owns the zone that this point maps into is responsible for that key. When retrieving the entries stored with this key, the same hash function is used on the key, yielding the same coordinate. A message is routed to the node that at this point owns the zone that this coordinate lies within. The corresponding value can then be retrieved.

To achieve this routing, some routing state needs to be kept by each node. Each node has the IP address and coordinate zone of each of its neighbours. When routing messages, the message is at each hop routed to the neighbour that has a coordinate zone closest to the destination coordinate of the message. [24] states that in a system of n nodes, each node has a fixed size routing table of $2d$ as it has $2d$ neighbours. The average number of hops a message has to take to get to its destination is $(d/4)(n^{1/d})$.

Chord

Chord [31] provides functionality to map a key to a single node. Scalability is reached through a distributed routing table. All nodes in the system only has links to a limited number of other nodes. A node receives a unique identifier

by using a hash function on its IP address. Identifiers are arranged in a circle. Mapping of keys to responsible nodes are subject to positions on the identifier circle. A key gets its own identifier on the circle from the hashing function. The node responsible for this key is the node succeeding the key in the identifier space. This is called the *successor node* of the given key in Chord. In an overlay network of n Chord nodes, the size of a Chord node's routing table would be of $O(log(n))$ entries. The number of hops required for routing a message in a Chord network with consistent routing tables would also be $O(log(n))$. The cost of a node joining or leaving the network is $O(log^2(n))$

The routing table of a chord node consists of i entries. The ith entry of the routing table points to the first node to succeed n by *at least* 2^{i-1} on the identifier circle.

Chord is fully self organizing. The maintenance protocols are described in detail in [31], but will not be covered here. Chord is capable of notifying the application whenever the key set each node is responsible for changes, but does not directly support ways of replicating data. However, replication is possible by assigning multiple keys to each piece of data. When Chord notifies the application that the keyset the hosting node is responsible for has changed, the application can use this to bring the system back to a consistent state.

Summary

All systems presented in this subsection give guarantees on visibility of resources and scalability in terms of number of messages routed in resource discovery and the number of hops[5] taken by messages. This means that in theory, all of these systems are perfectly usable in the design of a resource discovery component for QuA.

What these systems sacrifices relative to the unstructured systems described above is flexibility. To be able to find a resource from this system, you need the key that was used to decide the placement of the resource. In general, this means that all participating entities must agree upon how the keys are generated based on some metadata on the resources. This will not be a problem in QuA resource discovery, as discovery of resources is based on service types anyway.

2.5.4 Summary

In this section we have seen that the unstructured peer-to-peer overlays give no guarantees on preserving the offer space of resources within the system. This is in conflict with the consistency requirement stated in section 2.3.7. We have also seen that the structured overlays can give us guarantees on this point. Further, they give some good guarantees on routing table size, which will help to keep the node state on a reasonable level as well as keeping the lookup time low when routing messages, and number of hops required to route a message to its destination. Some of the protocols in addition offer APIs which can be used to ease the task of building applications on top of implementations of these protocols. That feature is reason for [13] to classify those systems as peer-to-peer middleware.

[5]Number of hops in this context is in the logical plane, i.e. in the application level, not at IP level.

2.6 Pastry walkthrough

[26] presents the design and evaluation of Pastry. This section contains a summary of Pastry as presented in [26], and reviewed in [21] and [13]. Readers that either has a good knowledge of structured P2P overlays in general or of Pastry, and readers that do not want to read the full introduction to pastry may read the "Quick overview" of Pastry from the previous section and skip the rest of this section. However, a full understanding of how Pastry operate is recommended to fully understand the design and implementation discussions of chapters 4 and 5.

2.6.1 Structure

Pastry is a structured peer-to-peer overlay based on the Plaxton mesh data structure [21]. All Pastry nodes have a 128-bit nodeID that identifies the node in the network. The nodeIDs are chosen from a circular key space by an algorithm that ensures an even distribution of keys in the key space. The ID also positions the node in the circular keyspace. The circular structure of the keyspace is used in the efficient routing mechanisms of Pastry. Even distribution of keys is ensured by using a cryptographic hash over the nodes IP-address or public key. [26] uses SHA-1 as an example algorithm.

2.6.2 Routing state

To make routing of messages possible, each Pastry node needs to keep some routing state. Each Pastry node has a routing table, a neighbourhood set and a leaf set. The routing table consists of $log_{\mathcal{B}}N$ rows and $\mathcal{B}$-1 columns. At row n, each of the $\mathcal{B}$-1 entries represents a node that shares a prefix with the table-owner in the first n digits. All numbers in the routing table are nodeIDs and all nodeIDs are in base $\mathcal{B}$. The n+1th digit of each entry has the value of its column place so that all the $\mathcal{B}$-1 possibilities of n+1th digit in the prefix are covered for the next step in the routing process. This way, with correct and fully filled routing tables, one can make sure that for each step you get to a node that shares a prefix with the key that is one digit longer than in the last step. b is a configuration parameter defining what base is used when it comes to the prefix-based routing, as $B = 2^b$ The choice of b involves a trade-off between the size of the populated portion of the routing table ($log_{\mathcal{B}}N$ * $\mathcal{B}$-1) and the maximum number of hops required to route a message between two nodes under normal operation ($log_{\mathcal{B}}N$). The neighbourhood-set M contains the node-IDs and IP-addresses of the $|M|$ closest nodes according to the proximity metric. This is not used directly in routing, but is used to keep the locality properties of pastry. The leaf set L contains the $|L|/2$ nodes with the node-IDs that are closest of the node-IDs that are larger than the node-ID of the leaf set owner, and the $|L|/2$ nodes with the closest smaller IDs. The leaf set is used during routing of messages. Typical values of $|M|$ and $|L|$ are $2 * \mathcal{B}$ and $\mathcal{B}$ respectively. Figure 2.10 shows the routing state of a hypothetical Pastry node in a Pastry overlay that uses 16-bit node-IDs. The configurable parameter b is set to 2.

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 2.10: Routing state of a hypothetical Pastry node in a Pastry overlay that uses 16-bit node-IDs. Figure taken from [26].

2.6.3 Routing messages in the overlay

The algorithm is invoked whenever a message arrives at a node, and whenever a message is dispatched from a node. It has 3 main steps.

- First it checks whether the message-key is numerically between the smallest and largest node-ID in its leaf-set. If it is, the message is bound for a node within the leaf set, and the message can be delivered directly.

- If not, step 2 is invoked. Step 2 starts with a calculation of the length l of the prefix shared between the current node-ID and the message-key. Then the node makes a lookup in its routing table row l, at the column of the number that matches the key in the $l + 1$th digit. If that entry exists, the message is routed to the node at that entry.

- If not, the algorithm proceeds to step 3. Here, it attempts to route the message to a node whose node-ID shares at least as long a prefix with the message-key as the current nodes node-ID does, and is numerically closer to the message-key.

This algorithm is bound to converge because each step either takes the message to a node that shares a longer prefix with the message-key or to a node that shares as long a prefix and is numerically closer. Pseudo-code of the algorithm, and a detailed description of each step, is contained in [26].

2.6.4 The Pastry API

A simplified version of the Pastry API is shown in [26]. In this API Pastry exports 2 methods.

1. `nodeID = pastryInit(Credentials, Application)`

2. `route(message, key)`

Method 1, pastryInit, initiates state for the local node and causes it to join an existing network or to start a new one.

Method 2, route, causes Pastry to route the message to the node with the closest nodeID to the provided key among live nodes.

Applications have to export 3 methods.

1. `deliver(msg,key)`
2. `forwad(msg,key)`
3. `newLeafs(leafSet)`

Method 1, deliver, is called by Pastry when a message that arrives is bound for this node.

Method 2, forward, is called by Pastry before forwarding a message to the next node on the routing path so that the application may perform last second changes to the message. For example, the application can check if its node is within the replication domain of the key, and respond to it. Routing to the nearest according to a proximity metric, of the closest k nodes to the key, as described in [26] and implemented in PAST [16] is probably done by terminating the message before its final destination, via this method. Caching can also be done via this method. Caching in PAST is described in [27].

Method 3, newLeafs, is called by Pastry whenever there is a change in the local node's leafset. It is important for many applications to be able to adjust to leafset changes. PAST is an example of an application that stores objects on the node that has nodeID closest to a key generated over some object information among live nodes. PAST also replicates the data on the k nodes with IDs closest to the key. To be able to keep the invariant that the closest k nodes to the key should have the object, it is important to be aware of leaf set changes.

In the implementation of the P2PBroker, an implementation of Pastry called FreePastry was used. The FreePastry API will be discussed in chapter 5.

2.6.5 Joining the overlay

When a node X wants to join a Pastry system, it first needs to find an existing member Y of the system to connect to. How this is done is irrelevant to this thesis. When X has found a node Y to connect to, it uses Pastry's ability to route a message to the node whose node-ID is nearest to the given key to find its starting-point for its own tables. Node X sends a special join message to Y with the node-ID of X as key. The message eventually reaches the node Z that is numerically closest to X in node-ID. All nodes on the path from Y to Z sends X their routing state tables. X uses this information to build its routing state. As basis for its leaf set, X uses the leaf set of Z, which is its closest neighbour in node-ID space. As a basis for its neighbourhood space, X uses the set of Y,

since it is assumed that the first known node in the system is close to X in the proximity space. The number of messages exchanged for X to reach a level where it has enough routing state to perform routing is $O(log_B N)$.

2.6.6 Self repair and survivability

In the case of node failure, Pastry repairs itself. When a node X contacts a node that has dropped out of the network as part of routing, it contacts other nodes to get routing state to update its tables. In the case where the dropped out node was part of the leaf set, the node with incorrect state tables contacts the node in its leaf set with id as far away from its own as possible in the direction that the failed node was positioned in the node-ID space, and updates its own leaf set with information from that nodes leaf set. This means that unless $|L|/2$ nodes with adjacent node-IDs fail simultaneously, Pastry will lazily repair its leaf state. If the dropped-out node is in Xs routing table, it contacts other nodes of the same row to get their entries. If no appropriate node can be found this way, X asks the nodes in the entries in its routing table at a lower row-number to get an appropriate entry. As for the neighbourhood set, the nodes are contacted regularly to check that they are alive. If anyone has dropped out recently, it asks for the neighbourhood tables of its own nearby nodes and picks a replacement out of that set according to the proximity metric. Experiments in [26] shows that Pastry is able to recover completely from a sudden drop of 10% of the nodes in the network with this lazy repair.

2.6.7 Network locality

Pastry also promises some nice locality properties. In its routing table and in the neighbourhood table, each node tries all the time to keep nodes that are close in terms of the proximity metric. This metric is application specific, but an example is number of IP hops, where a lower number of hops would be more desirable. A good description on how locality is maintained is given in [26], but the short version can be made relatively simple. The first assumption is that when a node X contacts a node Y to join the system, Y is likely to be close to X in locality terms. And with the additional assumption that Y has good locality in its state tables, X can use Ys neighbourhood table and level 0 routing table as a basis. Further, as the join message propagates through the system from Y to the node Z that has node-ID closest to the node-ID of X, the intermediate nodes on the way can send their highest level routing table that still has the same prefixes as Xs node-ID to X and X can use this as a basis for its routing table at that level. The assumption for this is that as the join message propagates through the Pastry network, the prefix routing makes sure that the set of possible entries in the routing table at the appropriate levels gets exponentially smaller as the matched prefix gets larger. So, as you get larger prefix matches, you have to look in a greater network proximity area for appropriate routing table anyway. The tables encountered on-route to the destination, which as mentioned is assumed to have good locality for their parent nodes, gives relatively good locality to the node X. To avoid cascading errors that would eventually lead to bad locality in the routing table, X now contacts the nodes in its neighbourhood set and in the routing table to get their state and updates its own state with appropriate entries that have better locality. When messages are routed

with this locality in the routing tables, it is assumed that the message will get closer to its destination in node-ID space for each hop, while travelling as short as possible in the proximity space on that hop by hop basis. It is clear that this procedure cannot give optimal end-to-end routes according to the proximity metric [26], but experiments show that it does give relatively good routes.

Chapter 3

Requirement specification

Based on the goal of the thesis, and findings in the background search of chapter 2, this chapter will present the requirements for a `QuA implementation broker` component, from now on referred to as `P2P-broker`. The requirements presented here are a foundation for the design which is presented in chapter 4.

All of the requirements are presented with describing text, and summed up in a statement preceded by the word **REQUIREMENT**. All statements contains one of the words *must* or *should*. The word *must* indicates that this is a absolute requirement for the design and implementation. The word *should* indicate a looser requirement, that one should strive to fulfil.

3.1 Functional requirements

This section describes the functional requirements to the `P2P-broker`.

Specification conformance

The `P2P-broker` has to follow the specification for a `implementation broker` component in QuA. In practice this means that it has to offer an interface conforming to the `ImplementationBroker` of QuA.

REQUIREMENT The `P2P-broker` must conform to the specification of the `QuA implementation broker`.

Consistency

To provide a reliable trading service, the broker must guarantee that resources advertised to it are returned as answer when they match an incoming query. The broker must not loose data. Neither must it give false data, for example reply with data on a service that does not exist. If asked for resources of a type, it must return all resources that match the type, and only resources that match the type. It must return all matching resources that it knows to be available and only resources that it knows to be available.

REQUIREMENT The `P2P-broker` must be consistent when dealing with resources.

Filtering

In QuA, each QuA type is associated with a number of attributes. Every service mirror describing an implementation or configuration of this type must include values for the attributes that require values. When asked for a resource matching a type, it is expected that the broker filters out resources based on a supplied list of attributes.

REQUIREMENT The `P2P-broker` must support attribute filtering.

Subtyping

To support evolution of services, the broker must support subtyping of service types. When queried for a service of a service type, it must return all services that conform to that type, and only those that conform to that type.

REQUIREMENT The `P2P-broker` must support subtyping of service types.

3.2 Non-functional requirements

This section presents the non-functional requirements to the `P2P-broker`.

Robustness

It was stated in our goals that we wanted to see if peer-to-per technology could help us build a robust resource discovery service. As nodes go down, or networks fail or gets partitioned, chances are that resources could become impossible to discover as the only nodes that knew of them now either are dead or disconnected from the rest of the network. The broker must be able to counter this.

REQUIREMENT The `P2P-broker` must show robustness to data loss during node and network failure.

Availability

It is of importance that users of the broker find it available. The service needs a high "up-time".

REQUIREMENT The `P2P-broker` must have high availability.

Scalability

To support many users, large amounts of data and a high number of queries, the broker must prove to be scalable to a large number of participating QuA-capsules.

REQUIREMENT The `P2P-broker` service must scale to a large number of users.

Resource consumption

As the nodes running the peer-to-peer broker needs to save as much resources as possible for other tasks, it is important that the broker uses as little resources as possible.

REQUIREMENT The `P2P-broker` should have low resource consumption.

Non-contributing nodes

As some nodes may have very restrained resources, it may be profitable to let some nodes use the broker without contributing to the service.

REQUIREMENT The `P2P-broker` should allow nodes to use the service without contributing to the service.

Chapter 4

Design of a peer-to-peer broker

In this chapter we will discuss the design of a peer-to-peer based implementation broker, the `P2P-broker`. A discussion of implementation specific topics is given in chapter 5. This chapter relates to the goals of the thesis by answering the first question of the problem statement of section 1.2. In addition it aids the work of chapter 5 in reaching the second sub goal of section 1.2.1.

This chapter is structured as follows. Section 4.1 gives an overview of the basic concept of the system. In section 4.2 we investigate how the property model of QuA could be extended to support some P2P-broker functionality. Section 4.3 shows how responsibility for individual resources are mapped to individual nodes to share the load between nodes. Section 4.4 shows how the design intends to handle robustness of the system through the use of a replication scheme. Section 4.5 shows how QuA is to be glued to the peer-to-peer overlay through the P2P-broker, and gives an idea of how responsibilities in the system are divided between layers. Finally, in section 4.6, we examine how the requirements from chapter 3 have been addressed by the design presented in this chapter.

References to the requirements stated in chapter 3 are made along the way to indicate which parts of the design cover which requirements. Some requirements may not be fully addressed before chapter 5. At the end of this chapter, a short summary will be given as to how the requirements are going to be fulfilled.

4.1 Conceptual view

The basic concept of the system is to build an `overlay network` of all nodes configured to use the `P2P-broker` as `implementation broker` in QuA, by using a peer-to-peer system to allow the nodes to connect into an overlay network. In this way, instances of a QuA platform that are configured to run the `P2P-broker` are allowed to communicate and cooperate to share the load of hosting service mirrors, and increase the offer space of resources. Figure 4.1 shows the basic concept. Every node is running a `P2P-broker`, which consists of 3 parts. The logic needed to communicate with the other parts of the QuA

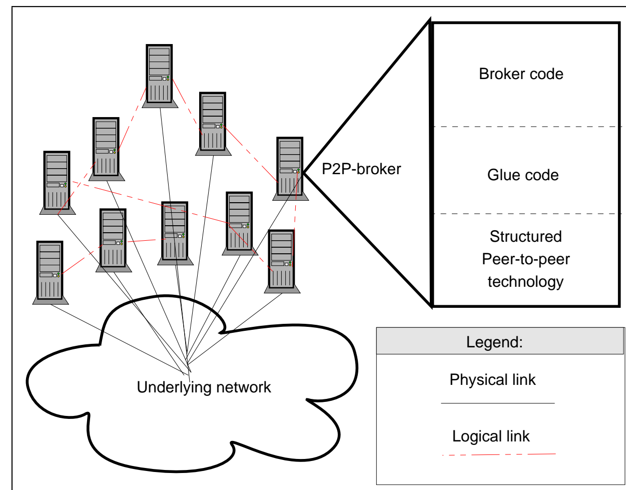

Figure 4.1: Basic concept of the P2P-broker.

middleware is referred to as the *broker code*. The *glue code* binds the broker code to the *peer-to-peer code*. The last part is the actual peer-to-peer node, which acts as the `P2P-broker's` endpoint in the overlay. To follow the requirement from 3.1 about consistency, it is vital to choose a technology that does not reduce visibility of any resources. In section 2.5.2 we saw how this property rules out all unstructured peer-to-peer technologies as they all to some degree are forced to trade network visibility for scalability. This leaves us with the structured peer-to-peer networks discussed in 2.5.3 as our only option. Thus the design of the `P2P-broker` demands that some sort of structured peer-to-peer network is used to construct the overlay. As we can see in the figure, each node only has knowledge of a limited number of other nodes, which they have established connections to via the physical network.

A collection of `P2P-brokers` constitute a distributed system that work as a unit to provide a service that is common and equal to all interconnected instances of the software.

4.2 Extending QuA properties

Before we can continue the discussion, we need to extend the property model of QuA. Resource discovery in QuA, and not property models, is the topic of this thesis, and as little space as possible will be used on the topic here.

From the discussion on QuA in chapter 2, we know that each service mirror has a map of $< property\ type >, < value >$, where the range of property types that are allowed in the map is determined by the QuA type specified by the service mirror. However, the value range of a property type is not specified by any means. In order to support any other functionality for matching properties than equality matching in the broker, this will have to change. For some of the funcionality that will be discussed in the next section, we also need the broker to be able to determine the value range and value type for properties

dynamically.

The proposed extensions to QuA to support the broker functionality are inspired by the ODP trader discussed in chapter 2.4 and are summarized as follows:

- The notion of a property type is concretized into a property type class, which specifies:
  1. The `name` of the property type.
  2. The `value type` of the property type.
  3. Whether the property type has an enumerated value range or not.
  4. If the property type has an enumerated value range, a set of possible values.
  5. Specification of matching operator for filtering.
- An object of this class for each property type of a QuA type to be stored with the QuA type in the type repository.
- A way for the broker to find this information at any time.

The `value type` of the property type determines if the value range is in integer values, float values, String values, boolean values or a general object value[1].

Other requirements may come for other purposes, but the ones proposed here are sufficient for the design and implementation of the P2P-broker as discussed in this thesis.

4.2.1 Filtering on properties

One of the requirements for the P2P-broker from chapter 3 was that it has to be able to filter service mirrors based on properties. In the current release of QuA, the limitations of the property model indicates that the implementation broker only is able to do an equality check on the properties when filtering. By using the value type and matching operator extensions to QuA described above, the P2P-broker will be able to perform filtering on a wider range of property types.

The `basic implementation broker` that already exists in QuA handles property filtering by an algorithm resembling the psedo-code of algorithm 1. At the end of the algorithm, all local service mirrors that match the properties of the incoming service mirror in type and all properties are in set M.

For the P2P-broker, the same approach can be used, but the comparison step must be switched for a slightly more complex step, where the matching operator of the property type decides how the matching is done.

In addition, we have to take into account the fact that the P2P-broker is a distributed system. In order to minimize network traffic, we need to filter resources on the node that has responsibility of the resources. Sending the whole set of mirrors matching the specified type to the local node, before the local node discards most of them because they do not match the property description is obviously a waste of resources in a distributed systems context.

[1]Other values may be added as needed. This thesis does not aim to come up with a general property model, but specifies only the additions needed to make the functionality of the P2P-broker as described in this thesis work.

Algorithm 1 Property matching

```
Set M := new Set();
Set S := retrieveLocalSetOfMirrors(incomingMirror.getType());
for all serviceMirror n_i in S do
  for all property p in serviceMirror n_i do
    if valueOf(p) != n_i.valueOf(p) then
      skipToNextMirror;
    end if
  end for
  // Mirrors n_i that get this far have matched all properties.
  M.add(n_i);
end for
```

4.3 Mapping service mirrors to nodes

In order to take advantage of the structured peer-to-peer systems and their strengths, we need a way of dividing responsibilities between the participating nodes. As we saw in section 2.5.3, nodes in structured peer-to-peer systems are given Ids from a global Idspace, and assumed to have responsibility of an area of the Idspace around the Id that was given to them. Any message sent to an Id that lies within a node's Idspace, even if it is not exactly matching any nodeID, is sent to the node that has responsibility for that domain. Responsibility of resources or objects in these systems are handled by associating each resource with a key[2] from the Idspace and see which node's domain that key lies within. The node that has responsibility of the given area gets all messages regarding that resource, and thus assumes responsibility for that particular resource. In Pastry, a node is assumed to be responsible for the keyspace[3] that is the collection of all keys that are numerically closer to itself than any other nodeId in the global circular keyspace. Tapestry nodes are responsible for all keys that have their nodeId as the longest matching Id to the key by prefix matching in a global circular keyspace. Chord nodes are responsible for all keys that have a value that lies between the numerical value of its own key, and the numerical value of the node that precedes it in the global circular keyspace. In CAN, things work out slightly different. Instead of a circular keyspace, nodes are arranged in a n-dimensional coordinate system, where each node has a responsibility of the immediate surrounding area of coordinates to itself. Resources have to be given a coordinate, and the node that has responsibility for the area that coordinate lies within assumes responsibility for the resource. Although the operation of these systems differ, they all have a way of dividing responsibilities based on some sort of keys.

From the discussion of QuA in chapter 2.3, we know that resource discovery in QuA is closely related to `QuA types`. Each `service mirror` has a `QuA type` associated with it that defines the type of service that `service mirror` represents. When a `service planner` is planning a service, it asks the `implementation broker` for resources that fit the description of the service.

[2]To avoid confusion, it is common in the literature to denote Ids referring to objects or resources as keys and Ids referring to nodes as nodeIds or Ids even though they are generated from the same identifier space.

[3]In this chapter, the terms "keyspace" and "Idspace" will be used interchangeably

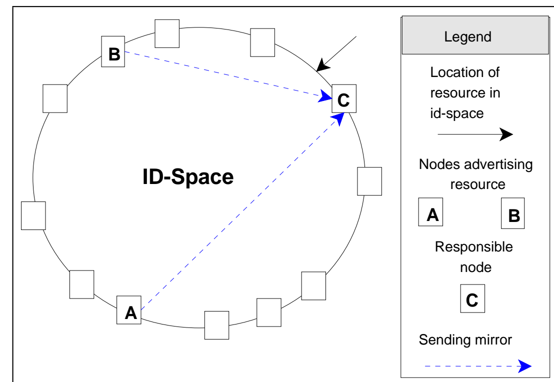

Figure 4.2: Mapping of service mirrors to nodes in normal configuration.

That description basicly consists of the `QuA type` of the service, and a `set of properties` the service need to match.

To be able to map `service mirrors` to `nodes`, we have to map `service mirrors` to `keys` in the Id space the nodes get their `nodeIds` from. In this thesis we will try two configurations of the `P2P-broker`, yielding two ways of mapping `service mirrors` to nodes. Both approaches are discussed in subsections to this section. The alternative configuration builds on the normal configuration, so the reader is advised to read this section sequentially.

4.3.1 Normal configuration

What is referred to as the normal configuration represents the most intuitive way of mapping `service mirrors` to nodes. As we know, when discovering `service mirrors` that conform to a service specification, all `service mirrors` that match the specification share the same `QuA type`. They may differ in any other aspect, but they all specify to the same type of service. Therefore, the most intuitive way to map `service mirrors` to `nodes` is to base the keys on the `QuA type` of the service. As we saw in section 2.5.3, all of the presented systems create keys and Ids with a hashing function to ensure even distribution of Ids in the identifier space. These hashing functions will always create the same id from the same input.

Consider figure 4.2. This figure illustrates the issue from a Pastry point of view, but as discussed above, it can easily be related to any of the other discussed structured peer-to-peer systems[4]. The P2P-broker program running on node A and node B both receive an advertiseMirror call. The resources they advertise are not the same, but they conform to the same `QuA type`. E.g. the `service mirrors` advertised are both describing a `QuA capsule`, but one has options for running Java code only, and the other has options for running smalltalk code only[5]. Both nodes find out which `QuA type` the mirror specifies and uses the string representation of the type as a basis for calculation an

[4]Pastry networks are chosen as example networks in the illustrating figures in this chapter because the Pastry topology and Pastry way of dividing the keyspace between nodes are thought to be easier to understand for the reader.

[5]This might not be the best example, as abilities such as that advantageously can be modelled

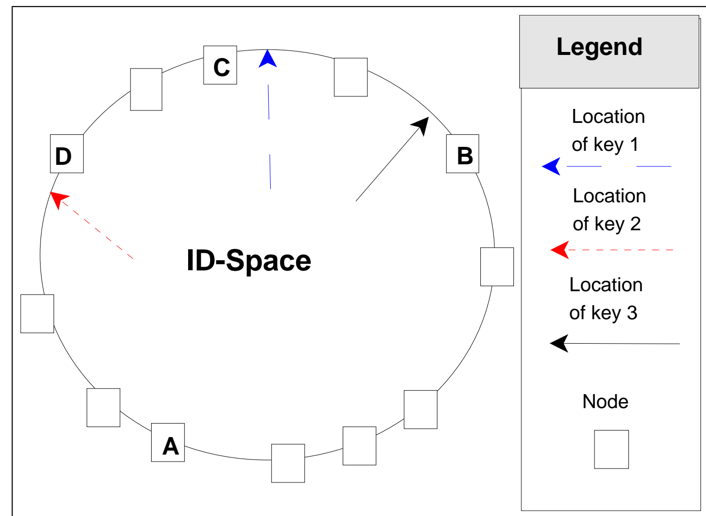

Figure 4.3: Mapping of service mirrors to nodes in alternative configuration.

overlay specific key for the resource. That basis will be known as keybase or idBase[6] throughout this thesis. As keys in the overlay are constructed through a hashing function, a specific idBase will always yield the same key in the same overlay network.

In this example, the key's place in the overlay keyspace is pointed out with an arrow in figure 4.2. When that is done, they send a message to that key with a label telling the recipient to consider this an advertisement of the contained mirror. As node C is the closest node to the key of all live nodes in the id space, node C has responsibility for the part of the keyspace that the key lies within, and receives both messages and stores both mirrors. If the `P2P-broker` program on any of the other nodes receive a getMirrorsFor call, it finds the `QuA type` of the service description given and sends a message asking for implementations of that type to the corresponding key. In our example, node C will receive all messages asking for implementations of the `QuA capsule type` as the key corresponding to that type is node C's responsibility.

4.3.2 Alternative configuration

Now imagine that a great many nodes have joined the network of `P2P-brokers`. If all of those nodes announce the capsule that host their own instance of QuA as a resource to their own P2P-broker, a number of service mirrors representing capsules equal to the number of nodes will be hosted at one single node. Worse yet, each time a service planner in any of those nodes want to plan a service, it may have to ask the broker for capsules that can host the components that constitute the service. All those queries will end up as incoming messages at the node responsible for that key (node C in figure 4.2). When the service

as subclassing of a QuA Capsule supertype. But for the purpose the examples in this chapter, we will disregard that possibility.

[6]The terms "keybase" and "idbase" will be used interchangeably.

planner is looking for capsule, it may even specify a number of attributes. For example it is possible that the capsule must be able to run smalltalk code. In our example, node C now has to search through all the service mirrors to find the capsules that match the description. This may take a while, especially if many service planners decide to plan a service at the same time.

Consider figure 4.3. The network of P2P-brokers may be seen as a *distributed hash table*, where each node has responsibility for a unique part of the key-range. In what we previously discussed, we saw that one node is responsible for each key as it lies within the node's key-range, and that node is the one node responsible for that resource. But it is possible to associate more than one key to each resource advertised. For example it is possible to take the properties of each service mirror into account when advertising them. We use the example once again with nodes advertising their own capsules as resources, but now in relation to figure 4.3. To make things simple in this discussion, we will assume that there only is one *enumerable property*[7] type for a capsule, specifying the type of code hosting capabilities it has. For simplicity, we also assume that a capsule may only host either java or smalltalk code, not both, and nothing else. In other words, the value range of the hosting capabilities property is *enumerated* to `Java` and `Smalltalk`, and it is not possible to have multiple values.

The idea of the alternative configuration is simple; By trading in more used storage space for each resource, but distributing it on participating nodes, the search space and thus time for queries which specifies needs through properties could be reduced. We will do this by creating more than one idBase in the cases where the advertised service mirror specifies values for enumerable properties. For this to happen, some of the property types specified by the QuA type specified by the mirror must have enumerated value ranges as explained in section 4.2.

The difference between this alternative configuration and the normal configuration described above, is the way idBases are constructed from service mirror metadata. As mentioned above, the metadata we are going to use to produce idBases for the alternative configuration are the QuA type, and the enumerable properties specified by the service mirror.

In general, a service mirror can be characterized as:

$$T, p_0, \ldots, p_i, \ldots p_n$$

where T is the type specified by the service mirror, and each p_i draws its value v_i from an enumeration domain D_i.

From that information, we can define idBases in the alternative configuration to in general be the form:

$$idBase = T + x_0 + \cdots + x_i + \cdots + x_n$$

$$x_i \epsilon \{v_i, s\}$$

$$v_i \epsilon D_i$$

[7]In this thesis we define the term "enumerable property" as "property that has an enumerable value range".

where T is the type of the service, v_i represents the value for the ith property type for the service mirror, s is the empty string representing a `wildcard`, and the $+$ marker indicates string concatenation. To be sure that idBases based on properties are generated in the same way in all participants of the network, an ordering of the property values have to be made. The ordering is based on the names of the property types for the QuA type specified for the mirror, and are in increasing order.

The P2P-broker instance that receives the initial advertisement of a new mirror goes through the properties list of the advertised service mirror and finds all *enumerable* properties that are specified. Then, by following the increasing order of the property types for the QuA type specified by the service mirror, it creates keybases for all possible combinations of the QuA type and values of the *enumerable property types* where the values are either the value specified by the service mirror for the property type, or an empty string representing a `wildcard`. One keybase for each possible combination of value or wildcard for all property types has to be created so that there will be one keybase that will match the keybase generated from any query that specifies the right value for any number of the enumerable properties specified by the advertised service mirror.

Similarly, a P2P-broker that receives a query for resources for a type and some properties, creates *one* idBase based on the QuA type wanted and the enumerable properties specified in the service mirror. The idBase will be in the form:

$$idBase = T + v_0 + \cdots + v_i + \cdots + v_n$$

$$v_i \epsilon D_i$$

where T is the type of the service, v_i represents the value for the ith property type for the service mirror in increasing order, and the $+$ marker indicates string concatenation.

In our example there are three possible keybases that can be created from the type and the enumerable property, of which any announced mirror at most can produce two. The three keybases are `QuA-capsule`, `QuA-capsule:Java`[8] and `QuA-capsule:Smalltalk`. Now, as we have more than one idBase for each `service mirror`, we associate one key with each keybase (again consult figure 4.3). Imagine that `QuA-capsule` yields key one, `QuA-capsule:Java` yields key two, and `QuA-capsule:smalltalk` yields key three. If node A advertises its own capsule as a resource, and that capsule has capabilities of hosting smalltalk code, both node C and node B would assume responsibility of hosting that resource, but under different conditions. Node C would now respond to all queries for a capsule without any preferences to hosting capabilities. Node B would respond to all queries for a capsule that has capabilities of hosting smalltalk code. Furher, node C would host one `service mirror` for each `capsule` advertised. Node B would only host `service mirrors` for capsules with smalltalk capability, and node D would host `service mirrors` for capsules with Java capabilities. The observant reader may notice that it is possible for more than one key belonging to a service mirror to be associated with

[8] The ":" delimiter is placed between type and property for readability, and is not supposed to be there following the definition of the form of keybases given above.

one node. However, the probability of this happening decreases linearly with the number of nodes joining the system. Further, if it happens, it will not cause a funtional problem, as the broker will have to be able to distinguish resources based on keybases anyway. The only negative effect would be that one node has multiple service mirrors of the same type without really needing to, and so uses up unnecessary storage space.

This scheme is possible to implement in any hash-table based application. However, it introduces a lot of redundant data only to give a chance of limiting the number of entries one would have to search through. To implement a scheme like this in a single machine program would therefore in many cases not make sense at all. In the case of a peer-to-peer based distributed system however, there are reasons to believe that some performance may be gained as the load of holding the redundant data can be spread across nodes, and the scheme also gives opportunities to share the filtering duties among multiple nodes in the case of high traffic. We also believe that this could be a way of using the strengths of structured peer-to-peer technologies and make a better `P2P-broker`.

On the other hand, a scalability problem in terms of storage space will be experienced using this scheme if the number of enumerable properties for an advertised service mirror becomes large. Actually, it will happen rather quickly, as the number of duplicate service mirrors advertised in the alternative configuration for a resource r equals 2^n, where n is the number of enumerable properties that the service mirror for r specifies.

If multiple values for each enumerable property type was allowed to be specified for each mirror, like it is in Jini (see section 2.4), fine granularity of the enumeration of the value space would also become a problem. By disallowing multiple values for each enumerable property type, a fine grained value space is not a problem, rather a bonus, as it allows for even better distribution of the service mirrors on nodes i a lot of service mirrors are advertised. Likewise, it would distribute queries better if a lot of service mirrors for a type were advertised, and a lot of queries for that type with different property values were submitted. However, with fine granularity of the value space, it would be more categories for service mirrors to fall into, and thus harder to create the right query to find resources. A good discussion on this problem in the context of Jini entry objects can be found in [23]

4.3.3 Responsibility and joining and leaving nodes

As nodes join and leave the network, responsibilities of areas of the keyspace change. As a result responsibilities for resources may also change, as a joining node may take control of an area of the keyspace that some resources map into, or a node that has responsibilities for some resources decide to leave. The underlying peer-to-peer technology promise that their networks are self-organizing, so that new nodes adjust into the network and gets routing state set up correctly, but it cannot take control of application specific state, like the sets of `service mirrors` that each `P2P-broker` has. In other words, the peer-to-peer part of the `P2P-broker` is able to find out which parts of the id-space

the node is responsible for at any time, even with joining and leaving of nodes, but the actual consistency of data has to be addressed in this design.

First we consider the case where a new node, called node A, joins the overlay and assumes responsibility of an area of the keyspace containing keys that belong to existing resources in the overlay. Now, two things have to happen. First, the node that just lost the responsibility for the resources, node B, has to discover that another node now has responsibility. This can be done by monitoring the local routing state and re-calculate the responsibility-area of the node at given intervals. As soon as the joining of node A is discovered by node B, node B has to send the relevant `service mirrors` to node A. Second, node B has to, upon completing the task of sending the `service mirrors` to node A, remove the service mirrors corresponding to the keys it no longer has responsibility for.

Next we consider the case where a node, called node C, leaves the network, and a nearby node, called node D, assumes responsibility of some resource from node C. If this happens, we want node D to receive the relevant `service mirrors` from node C, but this is not as easily done as in the case above. Basically, there can be two ways that node C leaves the network. It can either leave voluntarily or involuntarily.

Leaving involuntarily would imply a sudden drop from the network, unexpected by all. It might be e.g. because the node died or the network link that connected it to the overlay went down. In any case, it will be impossible for D to get hold of the `service mirrors` because the data is already lost.

If the node leaves voluntarily, one might think that the solution will be easy. That all node C would have to do was to send the data to node D. Unfortunately it is not. As long as node C is connected to the network, node D will not want the mirrors because, as node D sees it, node C is still responsible for the key. If node C leaves the network, it no longer will be capable of routing a message to node D. There are of course ways around this problem. For instance, node C may send a direct private message to node D after it has left the network by keeping node D's IP-address. However, the solutions will often involve a special trick to evade the basic rules of the peer-to-peer network, and it will probably be easier to handle the issue equally no matter how node C left the network. Besides, we want the overlay layer to handle all network traffic.

The best solution is probably to replicate the data somewhere, let node C die, and let node D assume responsibility via the replicating node. How we see the replication scheme being designed is covered in section 4.4.

4.3.4 Other P2P-based resource discovery systems

Although peer-to-peer based resource discovery systems that fits the requirements stated in the introduction to section 2.4 was hard to find, there exists some peer-to-peer based systems for resource discovery. Twine [5] is probably the most notable of these systems. Twine builds on the Intentional Naming System, which focuses on resource discovery in the mobile domain, and Twine is little more than a peer-to-peer extension to INS. The system as a whole is not comparable to the P2P-broker, mostly because of the way resources are

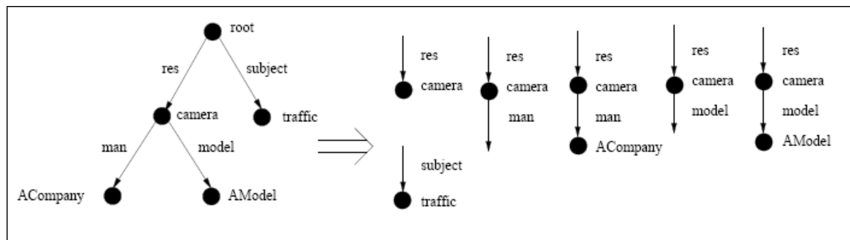

Figure 4.4: Creation of strands from an attribute / value tree in Twine. Figure taken from [5].

modelled, but the link between the peer-to-peer substrate and the resource discovery system is similar to what is presented here.

Twine uses Chord to distribute responsibility for resources among participating nodes. Twine creates something they call *strands* that are ultimately the basis for keys in the system, and as such are similar to our `idBases`. Resources in Twine are represented by a resource description consisting of $< attribute, value >$ pairs. Twine is not intended for use in component based middleware, and has no need for strong association of types to resources. Instead, the type of a resource can be included as one of the attributes describing a resource. This also means that there is no strong connection between the type of a resource and other properties of a resource. For each attribute of a resource, a pair of $< attribute, value >$ is added to the resource description.

Twine creates trees of these $< attribute, value >$ pairs as hierarchical structures of attribute types is possible. Top level attributes in the hierarchy connect to the root of the tree. Other attributes are connected to their parents in the hierarchy. On resource advertisement, strands are constructed from the trees in the way shown in figure 4.4. The details of the procedure are found in [5], and only a compact short-version summary is presented here. Strand generation does not produce one strand for each combination of attributes, but it produces one strand for each possible prefixed subsequence of attributes and values in each attribute hierarchy, where the top-level attribute in the hierarchy is the prefix.

When resources are queried for, one of the longest strands from it tree is extracted at random, and the node that has responsibility for the given key is asked for resources that fit the resource description. The reader should note that when this query arrives, the recipient has to perform attribute matching for all attributes, as only a subset of the resource description is used for determining responsibility for the resource, and for determining which node that resolves the query.

In comparison with the P2P-broker, the normal configuration only uses the resource type to determine responsibility. This means that every resource of that type has to be matched for properties against the resource description of the query.

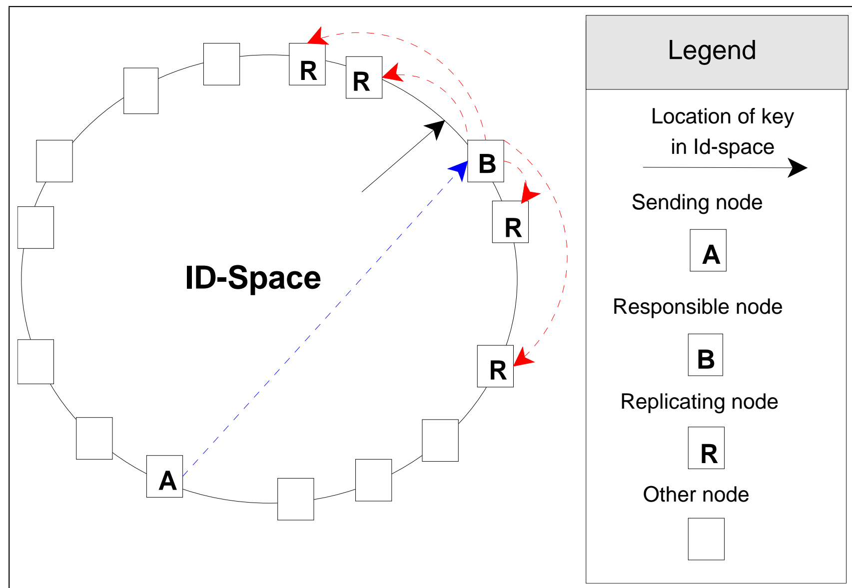

Figure 4.5: Replication in the P2P-broker.

The Twine way uses one of the longest subsets of the properties, and as such filters out a subset of non-matching resources for a given sort of resource implicitly via key mapping of strands.

In contrast, the alternative configuration would filter out all resources that do not match the enumerable properties of the resource description of the query implicitly, via key mapping of idBases.

Also note that the way attributes are described in [5], all attributes could be enumerable properties as described in 4.2.

Effectively, the strand-generation of Twine does not partition the offer-space as efficiently as the alternative configuration described here, but it scales better in terms of used storage space for resource descriptions, as the number of strands produced does not increase exponentially with increasing attribute count, as the number of idBases in the alternative configuration does. Therefore, an algorithm for producing idBases in the P2P-broker similar to the strand-producing algorithm in Twine is a possible candidate for a third configuration of the P2P-broker.

4.4 Modelling replication

The requirement specification stated that the broker must show resilience to node and network failure with respect to loss of data in section 3.2. This section shows how we intend to solve the problem by building in replication of data in the network. As a side note, the functional requirement from section 3.1 about consistency would also be under threat in an environment where nodes leave the network unexpectedly if not dealt with. By using replication, we expect to

solve this problem as well.

A key feature of structured peer-to-peer network is that the placement of nodes in the global identifier space relative to each other is very organized. It is always possible to figure out who is the closest neighbour in a given direction in the keyspace. These systems also have strict algorithms on routing messages, always routing to the node that at a given time has responsibility for the given area that the destination id of the message lies within. The combination of these two properties gives us the opportunity to use the immediate neighbours of a given node, node A, as the replicator nodes for node A. If node A unexpectedly leaves the network, one of its immediate neighbours, node B, will be in control of the area of the keyspace node A was responsible for. In effect, node B will receive all messages regarding the objects that node A just recently had. In some cases where node A had multiple resources associated with different keys, the responsibility of resources may be spread amongst a few of the immediate neighbours, depending on the system used. In any case, replicating data on the nearest neighbours to node A will solve the problem. In addition, each node must recalculate its immediate neighbours and area of key-responsibilities frequently so that replicas always are up-to-date.

In a special case, node A may leave and a node C may join the network with an Id that lies between node A and node B in the keyspace. In this case, constant monitoring of the routing state of node B should discover the arrival of node C. By recalculating the responsibility-area of Ids, B and C should be able to figure out which keys node C is responsible for, and arrange for B to send the relevant data to it.

Now consider figure 4.5 showing how replication would work in a Pastry network. The figure illustrates the basic idea. A resource gets advertised at node A. Node A calculates the Id [9] that corresponds to the `service mirror` and sends an advertising message to the network. Node B is the node responsible for that Id, so node B receives the message. The first thing node B does is to *replicate* the `service mirror` by sending a `replicate message` to the `replicating nodes` R.

In the original Pastry paper [26], the authors describe a system called PAST [27] [16], which is a system for persistent storage of data. In the cited papers, the authors describe a replication scheme that works in they way described above. A replication factor k is introduced, which represents the number of replicating nodes. In total, the $k + 1$ nodes closest to the key in the identifier space, are responsible for storing the resources for that key. This indicates that the above described way of building replication is possible with Pastry.

Tapestry has a different API than Pastry, and is intended to be used in a slightly different way. However, [36] shows that it is capable of routing messages to nodes closest to an Id in the Id space, like Pastry can, and that there is access to the routing state, which is very similar to that of Pastry, so it is likely that the same things can be achieved with Tapestry.

If using Chord, [31] suggests that replicas are placed on the k nodes succeeding the responsible node, as the Chord routing mechanism will route a

[9] Or possibly multiple Ids from multiple keybases if running the alternative configuration, but in this example it is only one key.

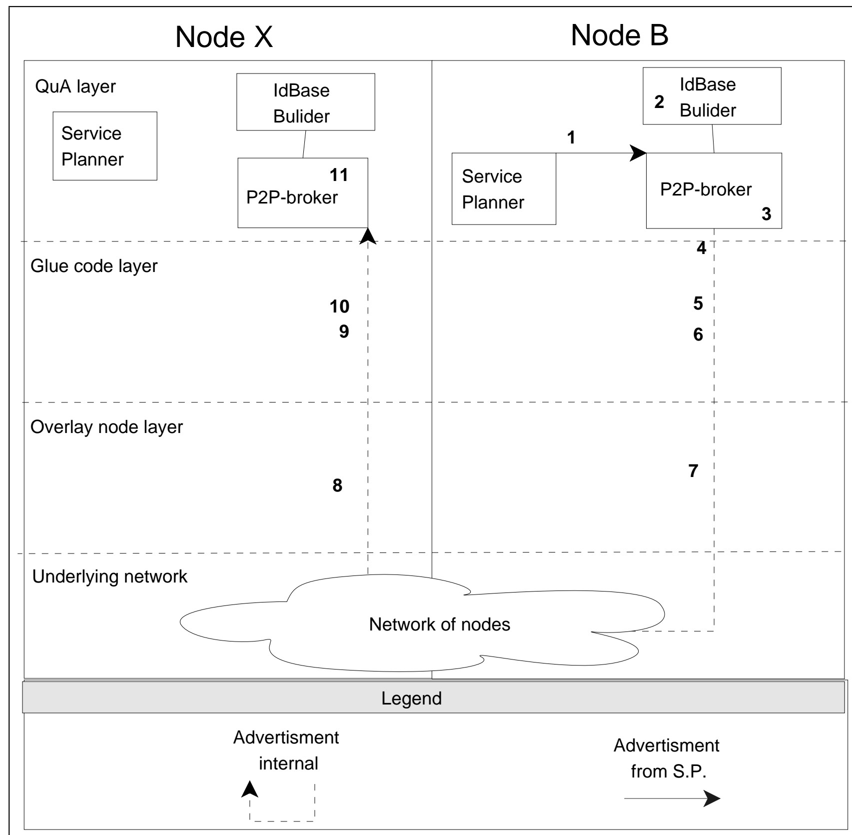

Figure 4.6: Layer responsibility in advertisment example.

message intended for a node A to the succeeding node in the case where node A fails.

Replication in CAN could prove to be a bit trickier, as the number of directions one would want to send replicas depend on the number of dimensions the system is configured with. However, [24] proposes replication of data on neighbouring nodes as a technique for increasing data availability in the case of popular content, and so the technique should be usable to increase data availability in case of node failure as well.

4.5 The layers of the P2P-broker

In this section we will move inside the `P2P-broker` and see how responsibilities are divided between the layers of the design. To help guide us through this section, we will present a scenario. In section 4.5.1 , we will go through the scenario. In sections 4.5.2, 4.5.3 and 4.5.4 we will sum up the responsibilities of the QuA layer, the Glue layer and the peer-to-peer layer, respectively.

4.5.1 Scenario

We will now walk through a scenario to identify the responsibilities of each layer of the P2P-broker design. The scenario consists of two parts. In part one, we will follow the route of an advertisement, and in part two, we will follow the route of a query for resources, and its reply.

Part one: advertisment

In part one of the scenario, the service planner at `node B` will advertise its own QuA capsule as a resource. This subsubsection will present the necessary steps of an advertisement in the P2P-broker, and identify the responsibility of each step within layers. Figure 4.6 will help us through the first part of this scenario. In addition to showing the layering of the P2P-broker, the figure has numbers matching each step of the procedure described below, so that the procedure should be easy to follow.

1. `Node B` advertises its own `capsule` as a resource by creating a `service mirror` describing the `capsule` and use the `advertise` method of the `implementation broker interface`. The resource is of the `QuA Capsule` type, has capabilities of hosting Java components, and has 512 MB of RAM. All of that is described in the `service mirror`.
2. In this scenario, the P2P-brokers are running in normal configuration as described in section 4.3.1, and node B creates the keybase that is used to decide which node in the overlay network that gets responsibility of the resource as described in previous sections of this chapter. The creation of the keybase is done in the idBase builder component which is a pluggable component within the P2P-broker. The difference between the normal and alternative configurations of the broker will only be which idBase builder they will use, as that component decides how idBases are generated, and implicitly how keys are generated.
3. The QuA layer creates a message, which it fills with the service mirror and labels the message as an advertisement. The message is also labelled with the keybase.
4. After creation, the message is passed on to the glue layer.
5. At the glue layer, the keybase is used to create an overlay-specific key that will be used to route the message to the right node.
6. The message is put into a wrapper-message specific to the overlay type and passed on to the overlay layer with the created key slapped on as destination.
7. The overlay layer is responsible of routing the message, node by node via the underlying network, until it reaches its destination, in our case the `node X` in the figure.
8. At the overlay layer of `node X`, the message is checked to be at the right destination before it is passed on to the glue layer.

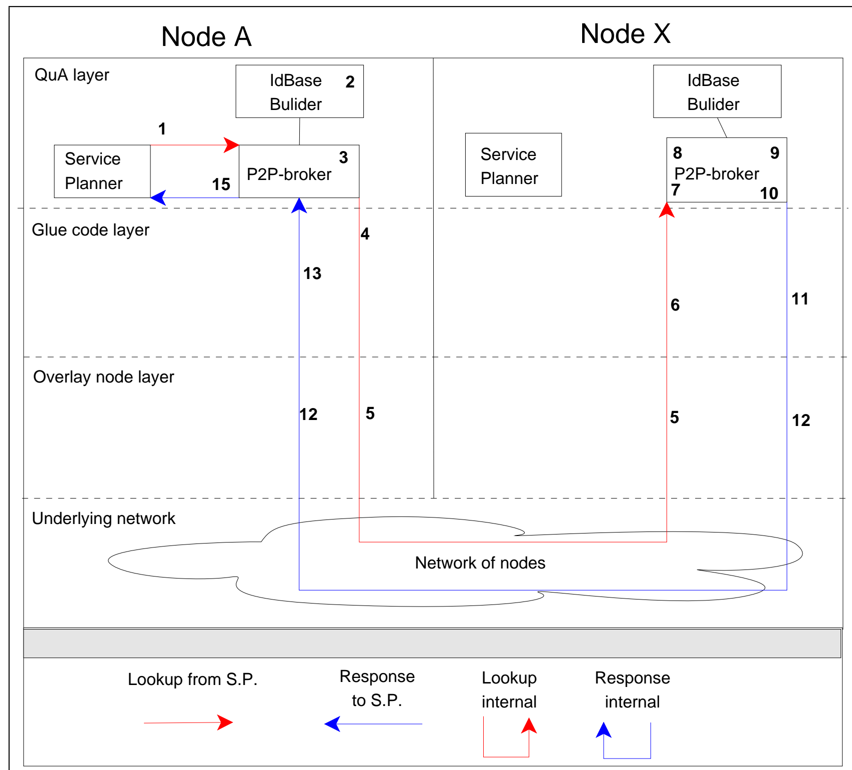

Figure 4.7: Layer responsibility in query example.

9. The glue layer at `node X` now rips off the wrapper message, and passes the inner message on to the QuA layer.

10. The glue layer is also responsible for informing the relevant replicating nodes that they have a resource to fetch, and to make sure it happens.

11. At the QuA layer, the message is unpacked and the `service mirror` is added to the local set of mirrors corresponding to the keybase.

Part two: query

Figure 4.7 illustrates the situation of part two of our scenario. The service planner at `node A` is planning a service, and needs a capsule resource with specific capabilities. We will see step-by-step how the P2P-broker resolves the query.

1. At `node A`, the `service planner` asks the `P2P-broker` for a capsule resource that has Java capabilities and at least 256 MB of ram.

2. The broker calculates the keybase that will be used to calculate the Id for the resource using the idBase generator component.

3. Next, it creates a message stating that it is a lookup for matching services. The message contains the service mirror describing the wanted resource, including any map of properties provided, and is labeled with the keybase produced.

4. The message is passed on to the glue layer which wraps it in message that suits the underlying network, labels the new message with a key generated from the keybase and passes it on to the overlay layer.

5. The overlay layer routes the message through the network, node by node, until it reaches the destination node, which is still `node X`.

6. At `node X`, the wrapper message is unwrapped in the glue layer and the message is passed on to the QuA layer.

7. At the QuA layer, the P2P-broker unpacks the message and extracts the keybase and the property map.

8. Next it will retreive all local service mirrors that that are stored with the kebase. The reason to pick mirrors conforming to the keybase and not the QuA type in this case is that in odd cases while using the alternative configuration, more than one keybase belonging to a given QuA type may have Id assigned to the same node. Further, we want the design to be able to have a pluggable keybase generator, and not re-write the code for the broker-logic every time the keybase generator is replaced.

9. After it has found all conforming local service mirrors, the broker will filter out all mirrors that do not match all properties in the property map, in the way sketched in section 4.2.1. In our example, we know that at least one mirror will match the properties, namely the one advertised by node B, but there may be multiple matching mirrors. The reason for filtering service mirrors remotely to the node that asked for them is to save as much network bandwidth as possible by creating the smallest set of mirrors possible for the reply message.

10. After filtering, `node X` will create a reply message containing the service mirrors and pass it on to the glue layer as a reply message.

11. The glue layer will use the sender Id of the original query message as recipient for the result message. Wrap the message into a overlay specific message, and pass it on to the overlay.

12. The overlay layer layer will route the message back to `node A`

13. The glue layer at `node A` will unwrap the message and pass it on to the QuA layer.

14. At the QuA layer, the P2P-broker will return the service mirrors in the message as result to the `getMirrorsFor` call from the `service planner`.

Next we will sum up the responsibilities of each layer.

4.5.2 QuA layer

At the QuA layer, communication with QuA is handled. When invocations from QuA is received, this layer is responsible for creating a basic message indicating the type of invocation and labelling it with a keybase indicating responsibility for the resource type. A pluggable keybase generator will be used to create the keybases. In the case of advertisements, a number of messages equal to the number of generated keybases must be created.

This layer also has reponsibility of storing service mirrors that the node is responsible for, and for finding relevant service mirrors as response to a lookup message from another node. Replicated service mirrors are also stored in this layer, as this makes for immediate recovery when a node goes down. When a node responsible for some key goes down, the neighbour that assumes responsibility will already have the replicated service mirrors ready in the QuA layer and is ready to respond to any lookup regarding the given key.

Filtering of resources happen in this layer, and it is important to do filtering on the remote node in order to reduce message sizes during lookup operations.

4.5.3 Glue layer

The glue layer glues the QuA layer onto the overlay layer. The responsibility of this layer is to hide the differences in operation in possible underlying peer-to-peer technologies from the QuA-layer. This layer receives messages from the above layer, creates an Id that is suitable for the underlying layer based on the keybase given from the above layer, and creates a message wrapper for the message that can be handled by the underlying layer. When messages arrive from the overlay layer, this layer unwraps the message and sends it to the QuA layer. This will be achieved by creating a general interface that the QuA level broker uses to contact the glue layer. Calculation of when the need to update replicated resources is also handled at this layer.

A new glue code must be provided for each overlay technology one wishes to use. The code at the broker layer will be separated and independent from any technology at the overlay layer, and any implementation at the glue layer.

In [15], an attempt is made to make a common API for structured peer-to-peer overlays. The interesting part in relation to this thesis, is that they show how methods to route a message to a node responsible for a key, and to monitor which nodes that should be replicating which keys, can be implemented in CAN, Pastry, Chord and Tapestry. In the design given in this chapter, that is all that is required of the underlying overlay layer. Thus [15] shows that the design presented here does not exclude any of the mentioned peer-to-peer technologies.

4.5.4 Peer-to-peer layer

This layer is responsible for keeping the network organized and for routing messages to the right nodes. The component used for this layer will be supplied by a third party and as far as possible used on an "as is" basis.

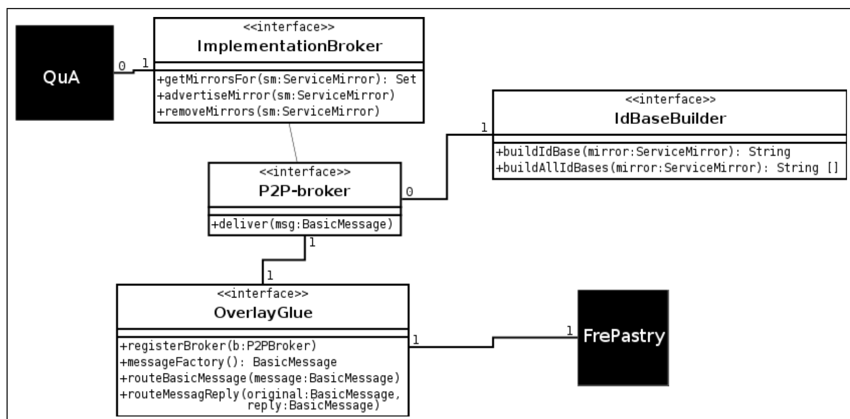

Figure 4.8: Interfaces of the P2P-broker

4.5.5 Layer interfaces

To be able to keep a clean separation between components, and to have a starting point for the implementation of the P2P-broker, the interfaces shown in figure 4.8 have been created and identified. The interfaces shown here should be sufficient to implement the P2P-broker by the design described in this section, but some of the interfaces may be extended as part of the iterative process of prototyping in order to build the best possible implementation. Interfaces to the overlay layer are not shown here, as those are specific to the technology chosen as overlay, and will probably have to be changed if the overlay technology is to be switched at some point.

The `ImplementationBroker` interface is the one QuA uses to contact the P2P-broker. The `IdBaseBuilder` interface is created to allow for a pluggable idBase builder component. Through the `OverlayGlue` interface, the QuA layer of the P2P-broker component talks to the glue layer. Likewise, the glue layer uses the P2P-broker interface to talk to the QuA layer of the component. How the glue layer communicates with the overlay layer is as mentioned overlay technology specific, and will be discussed in the implementation chapter (Chapter 5).

4.6 Relating the design to requirements

Finally, we will take a look at each of the requirements stated in chapter 3, and see how they have been addressed in the design, and address those that have not yet been addressed.

4.6.1 Specification conformance

To conform to the specification, the component in the QuA layer communicating with QuA must implement the `implementation broker interface` specifying the behaviour of the `implementation broker` component of QuA. How

to deal with the implications of implementing this interface is covered int section 4.5. During the iterative testing related to the implementation phase, the component will be tested in QuA to see if it is possible to run the QuA test-applications with the P2P-broker set as implementation broker. That will give an indication on how the component operates relative to the QuA implementation.

4.6.2 Consistency

To deal with the consistency requirement, the broker must not loose any of the mirrors advertised to it as a distributed system, and it must route advertisment and lookup messages concerning the same keybase to the responsible node. This is covered by replication, discussed in section 4.4, and by the combination of correct idBase builder and working overlay implementation, discussed in section 4.3 and section 4.5 respectively.

Another aspect of consistency is how to deal with nodes that advertise resources that are dependent of that node's participation in the system. Although it is defined to be out of the scope in section 1.4, it deserves a few lines here. If the service planner asks for resources of a type and gets that resource in return, it may try to plan a service that involves bindings to a resource that does not exist anymore.

Solving the problem is not easily solved in the P2P-broker alone. Rather, a cooperation between the service planner and the P2P-broker is needed. Solutions to this problem is often based on soft-state, leases (as used in Jini, see section 2.4), timeouts, or whatever term people like to tag the solution with. The P2P-broker could associate each advertised service mirror with a `discovery lease`, and whenever a service mirror is considered for as a match for a query for resources, the lease will be checked for expiration and any service mirror with an expired lease could be evicted from the system.

However, even if each advertised service mirror is associated with a timeout value, and evicted on expiration, the service planner may find itself in situations where it receives service mirrors from the P2P-broker that depend on a "dead" node. Therefore, to solve the problem, the service planner must check whether any node that a service mirror is dependent on is alive or not before it plans a service based on that service mirror.

Although this only is a coarse grained sketch of a solution, the topic is outside the scope of this thesis, as stated in section 1.4, and a solution to the problem will not be pursued any further.

4.6.3 Subtyping

Subtyping is not discussed in this chapter. Subtyping can be supported in three ways. First, the broker may be responsible of figuring out the supertype dependencies of any advertised resource and advertise it again for each supertype. Second, the party advertising the resource may be held responsible for advertising the resource once for each type it conforms to. Third, the service planner

may be responsible for finding any dependencies of types it asks the broker for, and may ask multiple times.

As there are ways of resolving this issue without implementing it directly in the broker, this issue will not be adressed further in this thesis.

4.6.4 Other issues

Filtering

Filtering is discussed in section 4.2.1. It is supported by simply implementing a property filter. The filtering of service mirrors will be done on the node responsible for the service mirrors of the given type, and not by the querying node, to save network.

Robustness

Robustness to node and network failure is addressed by replication, discussed in section 4.4. The robustness of the system will be tested as described in chapter 6.

Availability

The availability issue is implicitly resolved. As long as a node running QuA with the peer-to-peer broker is alive, the broker will be available as the underlying peer-to-peer network is self-organizing. The self-organization perspective of the P2P-broker will be tested as described in chapter 6.

Scalability

The scalability issue is addressed by implementing the broker on top of a structured peer-to-peer network, at least in theory. The way this works in practice will be tested as described in chapter 6.

Non-contributing nodes

To support nodes that want to be part of the peer-to-peer network without contributing to it because the node is weak when it comes to physical resources, we would have to alter the way the peer-to-peer networks operate. That is not within the scope of this thesis. Therefore this requirement has been disregarded in this design.

Resource consumption

The resource consumption of the `P2P-broker` is an implementation specific issue. This will be further addressed in chapter 5. However, as the non-contributing nodes issue will not be solved in this thesis, the resource consumption will quickly become a matter of how many resources a node has to store, more than the actual size of the program.

Chapter 5

Implementation

This chapter will present the implementation of the `P2P-broker`. This chapter relates to the problem statement of section 1.2 by making a tool to use when answering the second question of the problem statement. In addition, successfully implementing a `P2P-broker` and make it work with QuA would be a direct solution to the second sub-problem.

This chapter has the following outline. We will start by choosing technology for the implementation in section 5.1. Next we will take a look at an overview of the final implementation in section 5.2. In section 5.3, we will take a closer look at the chosen peer-to-peer technology from a programmer's point of view. How the functionality at the broker level has been implemented is the topic of section 5.4. The glue layer is inspected in section 5.5. Section 5.6 shows how replication has been implemented. Finally, in section 5.7, we take a look at possible improvements for the P2P-broker implementation .

5.1 Choosing technology

Before starting the implementation of the `P2P-broker`, we have to decide what technology to base our implementation on. In this section we will briefly discuss the technology chosen, and the basis for the choices.

The most important choices were what peer-to-peer technology to use, which implementation of that technology, and which programming language to program the `P2P-broker` in.

5.1.1 Peer-to-peer technology

The first choice was the one of peer-to-peer technology. As we saw in chapter 4, at least all of the structured peer-to-peer technologies reviewed in section 2.5.3 are viable choices, and with high probability, all structured peer-to-peer systems of this generation are viable choices. In any case, when looking for a suitable implementation we would have to have something to look for.

Criteria

The following became the main criteria for the choice of peer-to-peer technology;

1. Stability
2. Availability of implementation
3. Openness of implementation
4. Documentation
5. Programming Language

The first criterion is that the implementation should be stable. By that we mean that it should be tested that the functionality of the protocols the implementation is built upon are working in that implementation. The implementation should also be easily available. It should be freely downloadable and have a licence that allows use of the software in our application without limitations and free of any charge. Openness of the implementation code would allow us inspecting the code in case odd situations happen. It would be preferable that the implementation was *open source*, so that it would be possible to make small adjustments if that would become necessary. To be able to get started with the software as soon as possible, good documentation would be of great help. Finally, the language in which the implementation was programmed would be of interest. Java would be preferable as it would be easiest to make it interoperable with the current QuA prototype which is programmed in Java.

FreePastry

Without using much space on discussing the alternatives, the choice of implementation fell on FreePastry [2] version 1.4.4. The FreePastry project is a cooperation between Rice University of Houston, U.S.A. and Max Plank Institute for Software Systems, Saarbrücken, Germany. Compared to other systems, FreePastry seemed to have the best combination of the above described properties. The 1.4.4 release has been a stable release since April 28th 2006 and has already been used in a number of systems [2].

Distribution of FreePastry is under a BSD-like licence, and the implementation is freely downloadable. As for openness, FreePastry is distributed both binary and as source code, and modification of the source code is allowed by the licence under certain conditions. The documentation consists of a thorough Javadoc [1] as well as a very helpfull tutorial [20] written by Jeff Hoye to help programmers get started. Additionally, the community using and building FreePastry appeared to be both large and very active. In retrospect, this has proven to be very true indeed. Finally the implementation of FreePastry is in Java, which was the preferred choice.

We will quickly go through the possible alternatives to FreePastry:

- There exists an alternative Pastry implementation from Microsoft Research called SimPastry / VisPastry (actually two implementations that

are closely related). However, the homepage of the project stated that no documentation existed. In addition the last release was a beta-release from 2002, and it did not seem all that promising.

- A Java implementation of Tapestry exists, but the development team has been disbanded due to relocation of project members, and the implementation has not been in development since 2004. Further, the implementation had some weird dependency on a special build of the IBM Java development kit, which was no longer available at the IBM website.
- at the time of search for available implementations, the Chord project stated at their website that no release of Chord was currently publicly available.
- Information on implementations of CAN was hard to find at all

5.1.2 Programming language

In choosing the programming languages to use, a few things were important. The QuA prototype currently in active development at Simula is in Java. FreePastry is implemented in Java. If any other language than Java was to be used, complex bridging would have to be applied in one form or another. With limited time to use in the prototyping, it would be most undesirable to get involved in a complex multi-language patchwork just to get components to interoperate. In the light of these arguments, Java is clearly the best option, and was as consequence chosen as programming language for the implementation of the `P2P-broker`.

5.2 An overview of the implementation

This section presents an overview of the implementation of the `P2P-broker`. Figure 5.1 shows a UML-like class diagram of the core of the implementation. The diagram is not complete, only the most relevant classes, methods, and fields are filled in. Neither does it follow the UML standard to its full[1], but is intended to aid the reader in getting an overview of the important parts of the implementation, and more importantly, a sufficient overview in order to follow the discussion in this section, and the rest of this chapter. Some information will be filled in via tables and further diagrams as we enter specific topics.

Counting source lines of code (SLOC[2]) is a well known way of estimating the size of a software project. Although estimating workload and sizes of software projects is way out of scope of this thesis, we thought it might be interesting as a curiosity to mention. SLOCcount is a tool for calculation of the SLOC[3] for software projects created by David A. Wheeler, and we used it to see how the implementation of the `P2P-broker` compares to the current release

[1]The diagram is only a digram intended to show the structure of code in classes, without any great attempt made at following a certain software process or modelling language, but drawn in the UML environment of the Linux DIA tool.

[2]Abbreviation for "source lines of code"

[3]When counting source lines of code, we only count the lines that actually contribute to any computation. All commenting lines are for example disregarded.

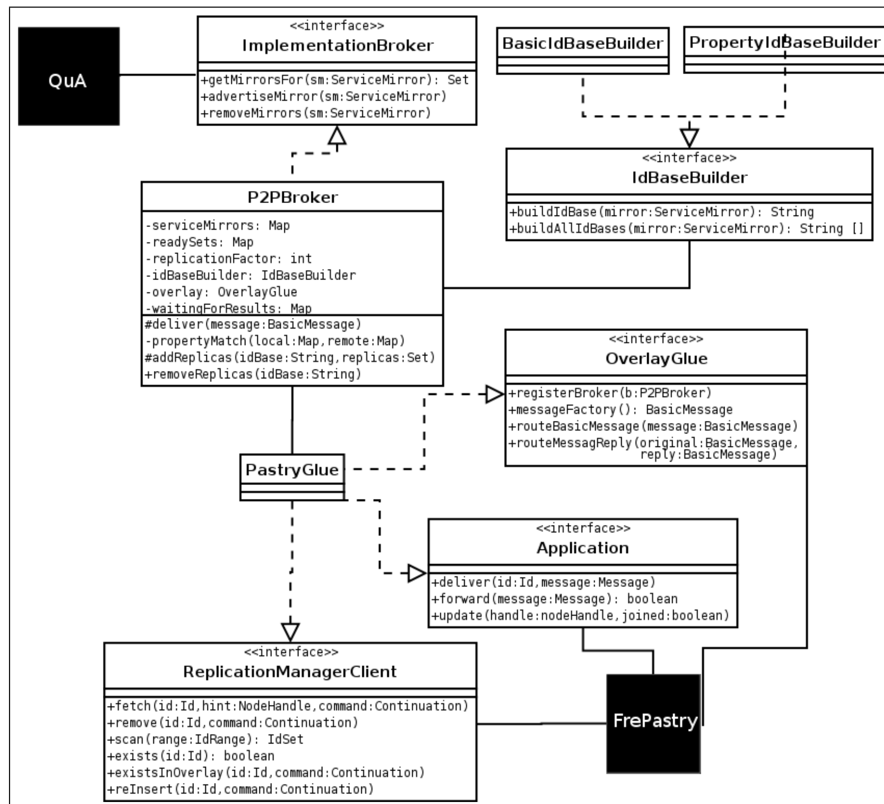

Figure 5.1: Overview of the implementaiton.

of QuA. The tool could tell us that the total QuA core implementation including the implementation of the P2P-broker counts 20400 source lines of code, of which 2375 lines, split into 28 classes and interfaces, belong to the P2P-broker implementation.

Resource consumption

How much resources the P2P-broker uses is of interest to see if it is deployable on small computers with limited resources. However, as discussed in section 4.6.4, small nodes will have problems using the P2P-broker anyway, as it probably is the number of service mirrors stored in the system that will be the problem.

Using the system monitor of Ubuntu, we observed that starting a single Java VM with one P2P-broker node increased the memory consumption with 9.2 megabytes. However, starting a Java VM with two P2P-broker nodes instead of one only consumed an additional 1.8 megabytes. This indicates that Java VM and the loaded Java libraries takes the most space, and that the actual program of the P2P-broker including the underlying FreePastry node only uses about 1.8 megabytes. The overhead of using the Java VM can possibly be reduced by using a different Java VM that is more resource-efficient. In addition, it is possible that the size of the loaded libraries can be reduced to further reduce memory footprint.

5.2.1 The basic system

In the figure, the rest of the QuA middleware and FreePastry is drawn as a black box indicating that we will not go into that at the moment. `QuA` communicates with the `P2PBroker` through the `ImplementationBroker` interface. The `P2Broker` uses the `BasicIdBuilder` or `PropertyIdBaseBuilder` to build Id bases. As the `P2P-broker` is pluggable in the QuA architecture through the `ImplementationBroker` role, these two components are alternative pluggable components through the `IdBaseBuilder` interface that they implement.

The `BasicIdBuilder` is the realisation of the normal configuration of the `P2P-broker` as discussed in section 4.3.1, with the name of the class reflecting that this is the most basic way of constructing id bases. Likewise, the `PropertyIdBaseBuilder` is the realisation of the alternative configuration described in section 4.3.2, reflecting that the alternative configuration takes properties into account when constructing id bases.

The `P2Broker` is connected to the `PastryGlue` through the `OverlayGlue` interface. Further, `FreePastry` demands that some interfaces are implemented by applications laying on top of it, for any `FreePastry` built system is to operate. One is the `Application` interface, required for FreePastry to be able to deliver messages to it. This interface is the same as the one presented in section 2.6, except that the `newLeafs` method has changed name to `update`. The other is the `Replication` interface, which really is a part of FreePastry's PAST implementation as will be explained in the next section, which is required to get replication to work. Finally, the PastryGlue communicates with FreePastry through a few interfaces provided by FreePastry. These interfaces are an

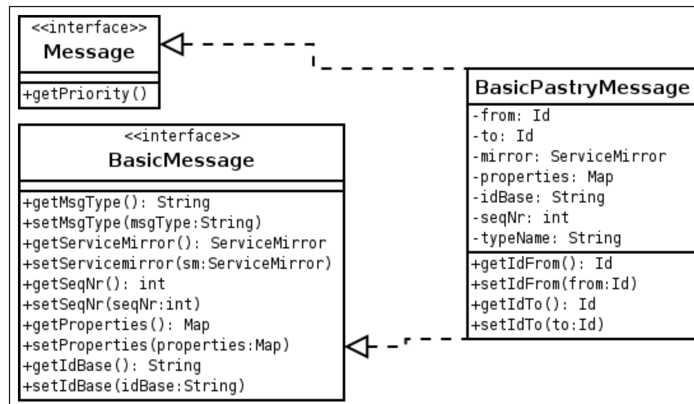

Figure 5.2: Overview of the message structure.

improvement on the basic interface presented in [26] based on the work[4] presented in [15]. We will describe FreePastry and the FreePastry API in section 5.3.

5.2.2 The message structure

As we saw in Chapter 4, instances of `P2P-broker` need to send messages to communicate with each other so that they can cooperate on hosting Service Mirrors and resolving queries. First of all, messages routed through FreePastry need to implement the Message interface provided by FreePastry. Second, the BasicMessage interface was created to be able to make a clean separation between the `broker logic` layer and the `glue` layer. Any message sent between the `P2PBroker` and the `PastryGlue` must implement the `BasicMessage` interface. As shown by figure 5.2, `BasicPastryMessage` implements both these interfaces, and can be used to send a message all the way from the `P2PBroker` class of one node to the `P2PBroker` class of another node without the `P2PBroker` class needing to have any knowledge of Pastry or the FreePastry implementation.

5.3 The FreePastry implementation

In this section we will take a closer look at the FreePastry implementation, and how FreePastry looks from an application programmers view. The FreePastry implementation is more than just a Pastry implementation. Shipped with it is also implementations of PAST [27], a distributed hash table, and Scribe [28], a system for application level multicast, both built upon FreePastry. As mentioned in 4.4, PAST includes a replication scheme built over Pastry, and FreePastry offers the replication manager used in PAST to easier build replication for applications using FreePastry. To gain access to it, the application should

[4] In the FreePastry tutorial [20] we can read that the interfaces we speak of *"... provides an interface similar to the one presented in [15]"*

implement the `ReplicationManagerClient` interface shown in figure 5.1 and register itself with the replication manager.

Next, we will present the application programmer interfaces, and see how to communicate with FreePastry. We will also take a look at how to bootstrap a FreePastry node and network. Discovering nodes at the network level is not within the scope of this thesis, but we will quickly go through the way it is done in the `P2P-broker` and refer to alternative ways of doing it.

5.3.1 Communicating with FreePastry

In the implementation of the `P2P-broker` we have implemented two interfaces to allow `FreePastry` to talk to our application. Both these interfaces are shown in figure 5.1 and will be discussed next. We will also discuss the relevant parts of the interfaces offered by `FreePastry` used in the P2P-broker implementation in this subsection. As these are a part of `FreePastry`, they are not shown in figure 5.1.

The Application interface

First, consider the `Application` interface. Whenever a `message` arrives at a `FreePastry node`, `FreePastry` use this interface to communicate with the application above, in our case the `PastryGlue`. In all cases where the node receiving the message is not the final destination for the message, `forward` is called. In other words, `forward` will be called on all nodes on the path from the origin to the destination of a message. At the end of this message, the application may choose to stop the message, by returning `false` as return value. If it chooses to return `true`, the message will be forwarded as soon as the function is completed. In other words, the application decides whether the message is forwarded or not. This can be used for example to build caching, as nodes on the route to the message destination can check if they are able to answer to the message before passing it on, and stop the message if they are able to answer.

In contrast, the `Deliver` method is called by `FreePastry` any time a message is bound for the current node, which would mean that the current node is seen by `FreePastry` as the final destination for the message.

The `Application` interface is also used by `FreePastry` to inform the overlying application of changes in the leafset[5]. As explained in section 2.6, the leafset is the set of the nearest nodes to the local node in the identifier space. Changes in this set are of interest to the application, and thus to us, because it is helpful in keeping application level invariants depending on the leafset intact. An example of this is that the k nodes closest to a node A should replicate the resources that A is responsible for.

[5]Confusion may arise at this point, as the set is called leafset in papers describing Pastry, like [26], where the neighbourhood set m describes the closest nodes according to the proximity metric. In the FreePastry implementation however, the set of closest nodes in Id space is known as the neighbour set. This is possibly to get closer to the definitions in other papers describing structured P2P systems. In this thesis, we will stick to the definitions given in [26] when discussing any form of Pastry technology.

The ReplicationManagerClient interface

Second, consider the `ReplicationManagerClient` interface. As noted above, FreePastry includes an implementation of PAST, and it is possible to use the PAST `ReplicationManager` to help in implementing replication. To use the `ReplicationManager`, an application has to implement the interface called `ReplicationManagerClient` so that the `ReplicationManager` can communicate with the application when needed.

The `fetch` method is called by the `ReplicationManager` whenever it discovers that the application running the local node should have had the resources associated with a given key. The application is supposed to contact get hold of the resources associated with the key and regard itself as a replicator of that key. To find out which keys a node is missing resources for, it asks the application to `scan` the set of keys it has responsibility for and return the set that lies within the IdRange specified by the `ReplicationManager`. In the case where the `ReplicationManager` discovers that the application holds keys that it should not hold, it asks the application to `remove` them.

To find out if a key is registered locally, the `ReplicationManager` asks the application if the key `exists` locally by calling the `exists` method. When this method is invoked, the application is supposed to look through the set of keys associated with the resources it hosts, and see if the key is among those keys. In some cases, the `ReplicationManager` wants to know it a key exists on other nodes than the local node. In these cases the `ReplicationManager` calls the `existsInOverlay` method, and the application is supposed to send a message to find out if the key exists in the overlay. If a negative answer to that message is received, but the local node has the key, the application may be asked to `reInsert` it into the overlay. In this case the application routes a message to the node currently responsible, often called *root node*, for the key given as parameter to the method call. The message must contain the resources held at the local node that are associated with the key.

Node

Through the `Node` interface, the application can get hold of information regarding the local pastry node. The interface contains 6 methods, and these are those that are of interest to us.

1. Environment = getEnvironment()
2. Id = getId()
3. IdFactory = getIdFactory()
4. NodeHandle = getLocalNodeHandle()

Through the `getEnvironment` method, applications get access to the environment that the pastry node lives in. In short terms, the environment is an object holding information on the environment the Pastry node lives in. For example, the application can kill the pastry node through destroying the environment it lives in. This is interesting for testing purposes as discussed in chapter 6.

The `getId` method returns the node's Id in the global identifier space. By calling `getIdFactory`, the application get access to the `IdFactory` associated with the node's environment. With this, it is possible to build Ids in the global identifier space as will be discussed in section 5.5.1. The `getLocalNodeHandle` method returns a `NodeHandle` reference which among other things can be used to check whether the underlying pastry node is "alive" or not. In addition to all this, the application registers itself with the Pastry node through the `Node` interface.

Endpoint

The `Endpoint` interface represents the endpoint of the local Pastry node. Through this interface, the application can route messages and get information on the routing state of the Pastry node. The interface contains 24 methods, and can not be presented in full. The most relevant methods are presented below.

1. NodeHandleSet = neighbourSet(int num)
2. IdRange = range(NodeHandle handle, int rank, Id lkey)
3. NodeHandleSet = replicaSet(Id id, int maxRank)
4. void = route(Id id, Message message, NodeHandle hint)
5. CancellableTask = scheduleMessage(Message message, long delay, long period)

The `neighbourSet` returns the leaf set[6] of the local Pastry node. The leaf set is among other things used in replication. The parameter `num` specifies the size of the set returned. From a call to `range` the application can retrieve an `IdRange` representing the range of keys the `NodeHandle` specified is responsible for. It is not allowed to specify a `NodeHandle` representing a node that is not within the leaf set. The `replicaSet` method is called to get a set of nodes that are responsible for replicating the given `Id`. The `maxRank` parameter indicates the number of replicating nodes wanted. A number of nodehandles equal to `maxRank` are returned. When calling the `route` method, FreePastry is told to route the `Message` specified as argument to the node responsible for the id sent specified as the `id` argument. If a `hint` is specified as argument, the method routes the message directly to that node first. The message may then be routed on from that node, if it does not turn out to be the true destination for the message. The `scheduleMessage` method offers a way for applications of sending messages to itself. The `Endpoint` has a timer thread that makes sure that the message is sent back at the intervals specified by the `period` argument, after first waiting a `delay` argument worth of milliseconds. This feature turns out to be very useful in that it eliminates the need for the application programmer to start and handle his own timer threads.

[6] Mark, the leafset and not the neighbourhood set is returned, see footnote 5.

IdFactory

An `IdFactory` creates Ids suitable for the `FreePastry` network. The interface contains 14 methods, which are mostly different ways of creating Ids based on different types of data. There are only two methods that are of interest in this thesis.

1. Id = buildId(String)
2. IdSet = buildIdSet()

The `buildId` method builds an Id based on a `String`. This is the one used to create `FreePastry Ids` from the `idBases` created by the `IdBaseGenerators`. The `buildIdSet` method returns an empty IdSet that can be filled with FreePastry Ids.

5.3.2 Bootstrapping the Pastry node

Although discovery of Pastry networks or Pastry nodes at the network level is not within the scope of this thesis, the reader may be wondering how a P2P-broker finds a network of other P2P-brokers to connect to, and this subsection will give the reader a few hints. As Pastry networks are self-organizing, a joining Pastry node will only need to discover one other already joined Pastry node to successfully bootstrap into the network. There are a few ways of making a bootstrapping mechanism that will work successfully;

- The joining node could use a webservice to discover live nodes.
- The system could be configured to try to connect to a number of nodes that are assumed to be established, connecting to each one in sequence until one that is alive is contacted.
- Create a multicast discovery solution like the one used to discover potential lookup services in Jini [33]. (see section 2.4.1)

The current implementation of the P2P-broker supports starting it through QuA, where it will attempt to connect to a specific node at a configurable address and port while taking a port on the local node that is currently free for itself, but also supports starting a node by specifying which port that should be used through a separate start-up procedure outside QuA.

The current implementation of the `P2P-Broker` supports two ways of starting it, each with different ways of finding a node to connect to.

- If started as a qua core service, i.e. in the `implementation broker role`, through use of QuA, the `P2P-broker` will look for a node to connect to at a pre-configured IP-address and port. The connecting node itself will take a currently unused port at the localhost as its own address.
- The `P2P-Broker` can also be started by a separate program. In this case, the address and port the node it wants to connect to can be specified, as well as the local port for the starting instance of `P2P-broker`.

By doing the latter first, it is possible to create a network by allowing all QuA started nodes to connect to the other node. A more advanced solution is preferable, but since it is not within the scope of the thesis to investigate the best possible ways of discovering nodes at the network level, this was not pursued.

5.4 Implementing the broker functionality

In this section we will review the implementation of the `P2P-broker` at the `broker level`. Central parts are the `P2PBroker` class, and the classes implementing the `idBaseBuilder` interface.

5.4.1 Configuration parameters

There are a few parameters that should allow configuration based on the environment the `P2P-broker` is deployed in. In addition to parameters regarding the bootstrapping of the `FreePastry node`, which will not be covered here as it is outside the scope of the thesis, are the replication factor `k`, deciding the number of nodes that will hold replicas of each data, and the setting of which `idBaseBuilder` to use. The default setting for the replication factor during the implementation and testing of the `P2P-broker` has been 5, which means that the responsible or root node and 4 replicating nodes for a total of 5 nodes will hold each resource. At startup, the `P2PBroker` class passes this straight on to the `OverlayGlue` which uses it in calculation of the replication. Recall the discussion on broker configurations in section 4.3. Using the `BasicIdBaseBuilder` corresponds to normal configuration. Using the `PropertyIdBaseBuilder` corresponds to configuring the broker in the alternative configuration.

BasicIdBaseBuilder

In the `buildIdBase` function, the `BasicIdBaseBuilder` extracts the string representation of the QuA Type that the service mirror specifies to. The string is returned as idBase. What idBases are used for is discussed in sections 4.3.1 and 4.3.2. The `buildAllIdBases` function does exactly the same, apart from creating a string array, with the only entry being the string representation of the QuA type. The array is returned. The reason why there is two methods performing what seems to be the same job, is that the class has to conform to the `idBaseBuilder` interface.

PropertyIdBaseBuilder

The extensions of the QuA property model proposed in section 4.2, were not implemented due to limited time. The places in the implementation where the extensions are needed are in the PropertyIdBaseBuilder, and when filtering service mirrors based on properties.

The effects on the propertyIdBaseBuilder is that information of every property type used in testing the alternative configuration of the P2P-broker has to be hard-coded into the the propertyIdBaseBuilder class. This was seen as

the most time-efficient way to prototype the alternative configuration of the P2P-broker as described in section 4.3.2

The algorithm for creating idBases is the `buildAllIdBases` function resembles the pseudo-code in algorithm 2.

Algorithm 2 Property idBase builder, buildAllIdBases

serviceMirror *sm* := incomingServiceMirror;
String[] *R* := new String[];
String[] *base* := new String[];
String *type* := *sm*.getType();
for all *property* in *sm*.sortAlphabeticallyOnPropertyTypeName() **do**
  base[*i*] := *property*.getValue();
end for
for all Possible ways to construct *idBase* on the form

$$idBase = type + x_0 + \cdots + x_i + \cdots + x_n$$

$$x_i \epsilon \{v_i, s\}$$

$$v_i \epsilon base$$

Where *type* is the type of the service, v_i represents the value for the *i*th property type for the service mirror, i.e. the ith place in the *base* array, *s* is the empty string representing a `wildcard`, and the + marker indicates string concatenation **do**
  R[i] := *idBase*
end for

At the end of the algorithm, all idBases are in *R*, ready to be returned to the caller of the method. To provide an ordering of properties when generating the idBases, as described as necessary in section 4.3.2, the alphabetical order of the property types based on the property type name is used in this implementation.

The `buildIdBase` method is used in queries for service mirrors. When the getMirrorsFor method is invoked at a P2P-broker, we want to find the service mirrors that match the QuA type and *all* the properties specified in the service specification. Therefore, the `PropertyIdBaseBuilder` implementation implements the `buildIdBase` function by extracting the QuA type from the service mirror and concatenating it with the values from each of the enumerable property types in alphabetical order.

5.4.2 Communicating with the OverlayGlue

The `P2PBroker` class communicates with the `OverlayGlue` incorporated by the `PastryGlue` as shown in figure 5.1. The interface shown in this figure is somewhat stripped, but contains the methods relevant to this discussion.

The `registerBroker` method allows, as its name implies, the P2PBroker to register itself with the OverlayGlue implementing class. This allows for a two-way communication between the P2PBroker and the OverlayGlue. To be

able to separate the P2PBroker entirely from what peer-to-peer technology lies under the OverlayGlue, the `messageFactory` method returns a message that complies both to the `BasicMessage` interface and to the underlying technology. In our case, the messages returned are instances of the `BasicPastryMessage` class as shown in figure 5.2. `routeBasicMessage` routes a message to a destination node, as will be explained in section 5.4.3. As will the operation of `routeBasicMessageReply` which is used to reply to a message sent as a result of a `getMirrorsFor` call to the `P2PBroker`. The `original` message is a stripped version of the message that caused the reply. It is included so that the node that initially sent the message can find out what the reply is a reply to. The `reply` contains the data that makes up the result to the initial call to `getMirrorsFor`.

The `OverlayGlue` communicates with the `P2PBroker` by sending incoming messages to it via the `deliver` method. In the `P2PBroker` the `deliver` method can be seen as a dispatch method, that delegates responsibility to sub methods based on the type of the incoming message. There is also a method asking the `P2PBroker` broker to add a set of replicated service mirrors to the local set based on an idBase. This method is called `addReplicas`. Similarly there is a method to remove replicated mirrors in the circumstance where the node longer is supposed to replicate them called `removeReplicas`.

5.4.3 Sending and receiving messages

Whenever a `P2PBroker` receives an invocation of `advertiseMirror`, `getMirrorsFor` or `removeMirrors`, it has to create a message containing the relevant data and pass it on to the `OverlayGlue` which is responsible for converting the message to be suitable for the underlying overlay network technology. The `P2PBroker` starts by asking the `OverlayGlue` to create a message conforming to the `BasicMessage` interface. Next it computes the idBase for the message by sending the service mirror as argument to one of the methods of the `idBaseBuilder` attached to it. When that is done, it uses the methods supplied by the `BasicMessage` interface (see figure 5.2) to attach the relevant data, set the message type for the message and add the idBase before it sends the message to the `OverlayGlue`. As this layer has no knowledge of the peer-to-peer technology used, it cannot specify the recipient of the message directly. Rather, the destination node for the message is implicitly decided by the `P2PBroker`, as it is the idBase that ultimately is the base for the computation of the key that decides the location of the node responsible for the message. The computation of key from idBase is done at the `OverlayGlue`.

The message type field is restricted to only four types of messages. Table 5.1. shows these message types and when they are used to label a message.

Message type	**Used when Receiving ...**
`query`	A getMirrorsFor method call
`advertisement`	An advertise method call
`result`	A query message
`removeMirror`	A removeMirrors method call

Table 5.1: Message types in the P2PBroker

5.4.4 Storing service mirrors

When service mirrors arrive at the `P2PBroker` of the node that is responsible for them, it has to store the service mirrors for later retrieval. We know that the idBase is being used to decide which mirrors are candidates for later queries, and so the most intuitive way would be to use a hash map to store service mirrors and use idBases as keys. This will allow for quick retrieval of the right service mirrors when a request arrives. However, it is possible to have multiple service mirrors conforming to one idBase. This is easiest to see if we are thinking in terms of the `normal configuration`, where the idBase for a resource is the string representation of the `QuA type` of the resource. The whole basis for the service planning driven approach of QuA is to have alternative implementations to choose from when planning. To represent this, each idBase which operate as keys in the local hashmap are associated with a Java Set, containing all service mirrors conforming to that idBase.

Had this not been a distributed system, this would have worked "out of the box". However, keeping the set as a *set* and not a *bag* in normal Java implementations of the Set interface is based on the `equals` methods that all objects in Java inherits from the Object superclass. The implementation of that method in Java is apparently based on the assumption that an object will always reside in the same Java-VM forever, which is not the case in the world of the `P2P-broker`. Therefore, after experiencing trouble with service mirrors duplicating themselves in the sets, the equals method was overridden in the `ServiceMirrorImpl` class implementing the `ServiceMirror` interface in QuA for the purpose of this thesis. Now, the `P2PBroker` assigns a `QuA name` created by QuA to each service mirror that arrives as the result of an `advertiseMirror` call. A QuA name is always unique through all instances of QuA. It is therefore used as identifier in the overriding `equals` method of the `service mirror`.

5.4.5 Implementing advertiseMirror

When a P2PBroker receives an `advertiseMirror` invocation it has to advertise that service mirror to the `P2P-broker` distributed system. To do that it has to create a message and send it to the node, or nodes, that are found to be responsible for the given service mirror. Since the `advertiseMirror` call has no return-value, the call is asynchronous, and there is no need take any special measures because of the P2P-broker being a distributed system. The only thing that we have to make sure is that the message arrives at the right node, so that it will be possible to locate the resource from any node where the `P2PBroker` receives a `getMirrorsFor` call. Reliability of message delivery is not handled at this level.

To decide which node, or nodes, the service mirror will reside on, the `P2PBroker` uses the attached `idBaseBuilder`. The `buildAllIdBases` method is used in advertising service mirrors as it generates one idBase for each node that will have responsibility for it. Returned from that method is an array of the strings that that make up the idBases that ultimately decides on which nodes the service mirror will reside. For each idBase in the array, the `P2PBroker` creates one message packed with the service mirror and the idBase, and labels it as an advertisement message. In sequence, each message is passed on to the

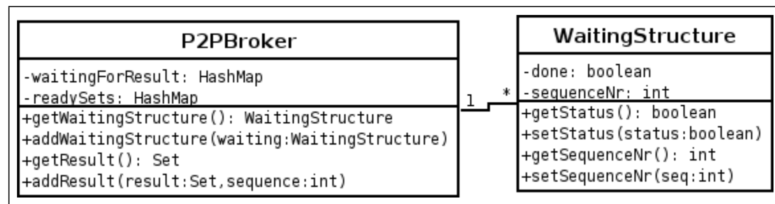

Figure 5.3: Added complexity of distributed search for service mirrors.

underlying `OverlayGlue` layer which handles the process of sending messages from there.

Nodes responsible for the idBases created when advertising the service mirror realize that they are responsible for the idBase when they receive an "advertisement" marked message through the `delivery` method. When this happens, the service mirror is stored by the receiving `P2PBroker` as specified in section 5.4.4.

5.4.6 Implementing getMirrorsFor

Upon receiving a `getMirrorsFor` call, the `P2P-broker` is supposed to find and retrieve all service mirrors that correspond to the specification passed as argument to the method. The `P2PBroker` program gets started by using the `buildIdBase` function of the `idBaseBuilder` to construct the correct idBase for the query. The next step is to find service mirrors corresponding to the idBase. But as the `P2P-broker` is a distributed system and the `getMirrorsFor` function call is synchronous, we get a problem to work around.

To find the node responsible for the idBase, a `BasicMessage` has to be created that can be sent to the node that has the wanted resources. The `P2PBroker` fills the message with the idBase, the service mirror describing the wanted service and labels it with the "query" message type. Next it passes the message on to the `OverlayGlue`. But then what happens? After the message has been passed onto the underlying FreePastry node for routing, the thread that initially invoked the `getMirrorsFor` function at the local node continues to run. But an answer is expected in form of a String returned from the function call, and the right answer is still to arrive.

Solving this problem is trivial, as we can simply establish a condition variable that the thread will sleep and wait upon. When the result message arrives at the local node, the waiting thread can be awaken and continue towards the return statement, retrieving the result on the way. But what happens if multiple threads ask the `P2P-broker` at the same node sequentially? If a second thread invokes the method before the first thread is given an answer, the `P2PBroker` will have a problem of deciding which thread should receive the results arriving through the `deliver` method.

To solve this problem, a more advanced waiting condition solution had to be implemented. The solution involves a sequence number generator that has synchronized access, providing unique sequence numbers. Each invocation of

the method generates one sequence number. The sequence number is attached to the query message sent to find the right service mirrors. In addition a class called `WaitingStructure` was created, and each thread gets an object from this class. This is an extension of the system compared to what is shown in figure 5.1. Figure 5.3 shows only the added complexity to allow for this solution.

In addition to giving a reference to the created `WaitingStructure` objects to the thread that is waiting for the result, all objects of this type are inserted into a hash map kept by the `P2PBroker`. The thread that invoked the `getMirrorsFor` function now falls asleep waiting for the `done` field of its `WaitingStructure` object to turn to true. When results arrive, the broker extracts the sequence number from the result message, retrieves the correct `WaitingStructure` object from its hash map, turns the `done` field to true, and adds the set of received matching service mirrors to a hashmap of sets that are ready for pickup. Finally it awakes all threads[7] sleeping on waiting structures. Each awaken thread may now check if the result they are waiting for has arrived. If that is the case, they can pick it up from the `readySets` hash map and continue. If not, they go back to sleep.

On the remote end, the `P2PBroker` receives a message labelled "query". When receiving a query message, the `P2PBroker` has to find the mirrors that match the given idBase[8] and properties specified by the service mirror attached to the message. The first thing it does is to retreive the set of service mirrors matching the idBase from the local storage. Next the `P2PBroker` will go through each mirror in the set, match the properties against those asked for in the way described in section 4.2 and see if the mirror complies to the requirements specified by the properties. All mirrors that conform to the specification are put into a set and shipped back to the node that sent the query message as a result message.

On a side note, it is possible that a network-reliant broker should allow for asynchronous calls to a getMirrorsFor call, involving a call-back function of some sort. This was not done in this thesis because of the limited amount of time for implementation, and the fact that it was no support for it in QuA at the time of implementation and no test-applications or service planners that could use this functionality.

5.4.7 Serializing service mirrors

The reader may wonder how the serialization of `service mirrors` was done. Fortunately, the service mirror architecture is so well constructed that this did not become a problem at all. The only references kept in a service mirrors are to other service mirrors, other parts of the service mirror architecture, or are represented by QuA names, and thus not reliant on any specific run-time environment and easily serializable. Built in Java serialization was sufficient for our purpose.

[7]An obvious improvement would be to awake only the thred that was waiting on the recently arrived resources. This could be implemented by keeping thread identifier in the WaitingStructure class. Again, time was limited, and this was not a priority.

[8]Mark that this will imply that it matches the QuA type

5.5 Gluing the broker to FreePastry

This section describes how the P2PBroker was glued to FreePastry by the PastryGlue component implementing the OverlayGlue interface shown in figure 4.8. Topics covered in this section include how messages are relayed from the P2PBroker to the FreePastry node, dispatching of incoming messages and how reliability of message delivery is implemented. Replication is also handled mostly in this layer, but that topic is so big that it is covered in a section of its own; section 5.6.

5.5.1 Relaying messages

One of the main responsibilities of the glue layer is to convert the messages the P2PBroker wants to send into a form that is suitable for the underlying peer-to-peer technology. As the P2P-broker is implemented in java, the PastryGlue does not have to explicitly wrap messages from the broker layer in new messages that are suitable for FreePastry. Instead it is sufficient that the messages it creates in the message factory conform to the Message interface specified by FreePastry, in addition to the BasicMessage interface required by the P2PBroker.

By a call to routeBasicMessage, the P2PBroker informs the PastryGlue of its need to send a messsage. The PastryGlue takes the idBase from the message, sends it to the buildId function of the attatched IdFactory of FreePastry to create a key that belongs to Pastry's identifier space. Next, the message's To field is filled with that key, and the From field is populated by the Id found from a call to the underlying Node. It then passes the message on to FreePastry, where it is routed to the node responsible for the key. The implementation of the routeBasicMessageReply method is quite similar, but the To field can obviously not be based on the idBase attached to the message, but instead the From field of the originalMessage.

5.5.2 Dispatching incoming messages

The deliver method that PastryGlue inherits from the Application interface, is implemented much like a dispatcher. It sorts messages based on message type and passes it on to helper methods. This task is in principle relatively simple, as most messages are supposed to be handed directly on to the P2PBroker. However, some complexity is added by the need for replication and the way reliability of messaging is handled. The latter is discussed next. All messages with message types found in table 5.1 are sent to the broker. Messages with types found in table 5.3 are related to replication, and messages with types found in 5.2 are related to reliability of messaging.

Message type	Means that ...
reminder	the message reply timeout is reached for a sent message
ack	the message is an acknowledgement

Table 5.2: Message types related to reliability

5.5.3 Reliability of messaging

Even as a Pastry network is supposed to always deliver a message to the node closest to the key that the message is sent towards, some simple testing during implementation discovered that some messages are indeed lost. To counter this, it was decided to implement a simple *send-ack* scheme at the glue layer. Doing this adds some complexity to the implementation of `PastryGlue` and specifically to the `routeBasicMessage` and `deliver` methods. In addition, a structure for holding the non-acked messages and sort these had to be added, as well as mechanisms for sending message acknowledgements.

To implement a simple send-ack or send-reply scheme, three basic additions to the code are needed. First, we need a way of recognizing what a reply is a reply to. Second, we need a way to send an acknowledgement saying that the original message arrived. Third, if no ack is received in a while, we need a way to find out that the originally sent message is with high probability lost, and then resend the message.

The first addition is realized by adding a helper class to the `PastryGlue` called `MessageKeeper`. The added complexity of the `MessageKeeper class` to the overview given in 5.1 is given in figure 5.4. An object of this class is created every time a message is sent with the `routeBasicMessage` method of `PastryGlue`. In addition, a field called `PastrySeq` is added to the BasicPastryMessage class to keep track of messages. The same sequence number is added to both the message and the `MessageKeeper` object before it is passed on to `FreePastry`. All messages that are not accounted for have an entry in the `outstandingMessages` hash map in `PastryGlue`.

The second addition is realized by a helper method that is called each time a message that needs acknowledgement arrives in the `deliver` method. That method creates a message with the "ack" message type, marks it with the `pastrySeq` of the incoming message, the right `To` and `From` fields, and sends it away.

To realize the third addition, we have used the excellent `scheduleTask` function of the `Endpoint` interface in FreePastry. A message, labeled with the "reminder" type, is created with the same `pastrySeq` as the sent message whenever a message is sent reliably. This message is passed on to FreePastry by calling the `scheduleTask` function. The `CancellableTask` object is kept within the corresponding `MessageKeeper` object, so that the reminder message can be cancelled when an ack arrives. However, if the ack does not arrive within the timeout value sent as both long arguments to the `scheduleTask` function, the reminder message will arrive in the `deliver` method of the PastryGlue. Whenever this happens, PastryGlue can use the `pastrySeq` number to retrieve the right `MessageKeeper` object and resend the message that it has not received ack for. Both the maximum number of re-sends, and the timeout have been tuned multiple times during the iterative testing (consult section 1.3), and are at the time of writing set to 3 times and 30 seconds respectively. To find appropriate values for these parameters, a deployment of the component with testing in a real network is needed.

When a message that is seen as an acknowledgement to another message arrives at the deliver method, the `MessageKeeper` object that corresponds to that class is retrieved from the outstandingMessages map. The `CancellableTask` is cancelled, so that no more reminder messages regarding that message are sent

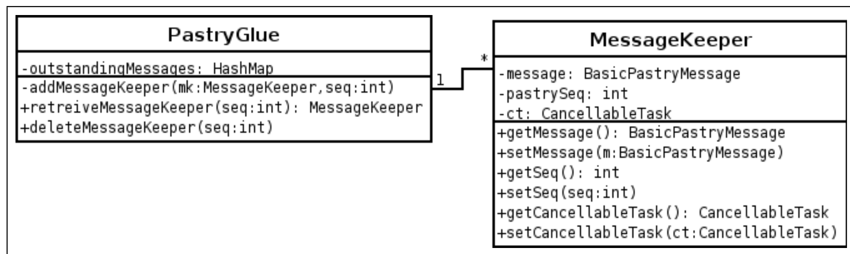

Figure 5.4: Added complexity of controlling reliability of FreePastry messaging.

from FreePastry. Next the object is removed from the hash map.

Messages that are a reply to another message, like the result message is a reply to a query message (see table 5.1), do not generate MessageKeeper objects, and are not acked, as they are in a way ready acknowledgements to the message they are replies to. Likewise, messages that will generate reply messages, like the query message that will generate a result message in reply, are not acked by the ack-mechanism of PastryGlue. The result is sufficient as acknowledgement in this case. This is based on the assumption that queries will not take very long time to resolve compared to the assumed time-overhead in sending messages across the network. That assumption may prove not to hold in all environments, and is a candidate for change in the system.

5.6 Implementing replication

This section presents the work of implementing a replication scheme for the P2P-broker. As explained in section 4.4, the PAST and Pastry papers [27] [16] [26] describe a replication scheme for PAST that is very similar to what is needed for a `P2P-broker`. After an e-mail discussion with the FreePastry team where it was claimed that the replication manager used in PAST would be sufficiently generic to support replication for our use, it was decided to build replication based on that replication manager. In the next sections, we will take a look at the basic PAST replication architecture in section 5.6.1, and investigate how to implement the support the replication manager needs from the application to calculate which datasets to replicate on each node in section 5.6.2. In section 5.6.3 we will learn how the replication manager provided with FreePastry turned out not to be sufficiently generic after all. In section 5.6.4 the solution to that problem is discussed.

5.6.1 PAST replication manager overview

The replication manager provided with FreePastry is the one developed for their PAST implementation. Figure 5.5 shows a basic overview over the components that make up the architecture of the FreePastry replication manager. The `ReplicationImpl` is an implementation of the role `Replication`. The job of the `Replication` role is to communicate with the same role at other nodes in the leafset, and agree upon which nodes are responsible for which sets of keys.

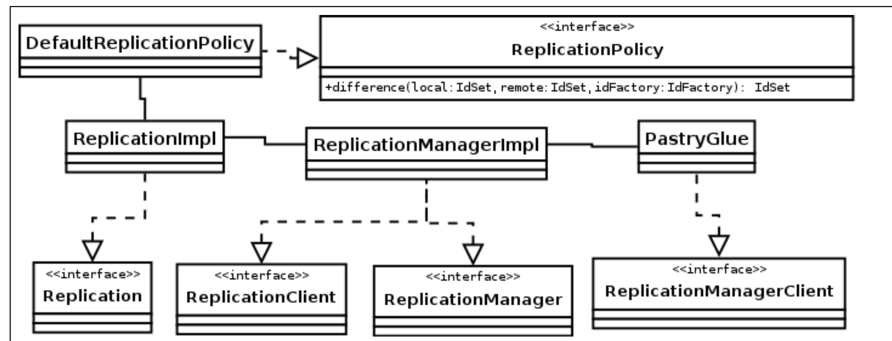

Figure 5.5: Overview of the FreePastry replication manager architecture

From the `ReplicationManagerImpl` which implements the `ReplicationClient` interface to become a client of the `Replication` role, it gets information on which keysets are present on the local node, and which keys are not present on the local node, but in the network as a whole. The latter is discovered by asking the implementing class of the `ReplicationManagerClient` interface which is discussed in section 5.3.1. In addition to helping `Replication` in this way, the `ReplicationManagerImpl` implements the `ReplicationManager` interface and offers an increased level of communication with the application, which in our case is the `P2P-broker` represented by the `PastryGlue`. The `ReplicationManager` role is also responsible for making sure that the application retrieves the resources bound to keys that it is asked to fetch through the `fetch` call of the `ReplicationManagerClient` interface.

From communication with the `Replication` role on other nodes, a local `ReplicationImpl` object finds the sets of keys that other nodes currently have objects attached to. These sets are filtered through the `IdRanges` that the local `ReplicationImpl` object has found the node to be responsible for or within replication reach for, also through communication with other nodes. The filtered sets represents keys that the local application should store. From the `ReplicationManagerImpl` it receives the set of keys that are actually stored at the local node at the present time. Through the pluggable `ReplicationPolicy`, it finds which keys need fetching and sends the list of keys that need fetching to the `ReplicationManagerImpl` which makes sure that the application fetches them. The pluggable `ReplicationPolicy` is the role that should ensure a generic and flexible replication architecture. The `DefaultReplicationPolicy` implements the difference method by simply checking which keys in the remote set that does not exist in the local set, and return those. This is equivalent of finding out for which keys the remote node has one or more resources attached to it that the local node does not have any resources attached.

5.6.2 Replication at the glue layer

For our `P2P-broker` to support replication through the replication manager supplied by FreePastry, the `ReplicationManagerClient` interface shown in figure 5.1 and discussed in section 5.3.1 was implemented by `PastryGlue`.

First, some new message types had to be added. The new message types are

shown in table 5.3. Some new infrastructure was also added to the `PastryGlue` to help in implementing the functions. First of all, the `PastryGlue` needs to always know which keys the node holds resources for at any given time. Therefore, each advertisement and removeMirror message are intercepted, and a data structure called `pastryIdtoIdBase` implemented as a hash map is kept along with a normal `Set` of keys held. The set will allow for very quick response when asked for the local keyset, while the idToIdBase map will help when the node is asked to retrieve the resources corresponding to a given key.

Message type	Means that the message ...
`replicateNow`	asks this node to start replication procedure
`re-insert`	contains mirrors that are re-inserted by replication
`fetch`	asks for mirrors to replicate
`fetch-reply`	is a response to a fetch message
`exists`	asks if this node has resources for a key
`exist-reply`	a reply to an exists call

Table 5.3: Message types related to replication

To be able to retrieve the keys that should be stored on a node, but that are not, the `fetch` method is implemented. On the local node, the method consists of creating a message with the "fetch" message type, loaded with the key needed to be fetched. The message is sent with the key to be fetched in the Id field, and with the hint provided from the `ReplicationManagerImpl` specified as hint to the `route` method of the endpoint. When a hint is provided, `route` is supposed to send the message directly to the hint first, and route the message on from there if it turns out not to be the right node. In some odd cases, however, FreePastry will not send the message to the hint directly if it believes that the local node is closest to the key of all nodes. Therefore, `PastryGlue` is monitoring fetch messages to identify these situations, and respond to them with a backup solution, where the node-Id of the `hint` is used as id in the `To` field of the message instead of the `key`. When receiving a message at the remote node, the `pastryIdToIdBase` map is used to find the corresponding idBase, and the `P2PBroker` is asked to return all service mirrors that correspond to that idBase. The set of service mirrors are put in a "fetch-reply" message and sent back to the local node. Back at the local node, `PastryGlue` sees if the set is empty or not. If the set is empty, fetch failure is reported back to the `ReplicationManagerImpl`. If the set is non-empty, the set is sent to the `P2PBroker` and success is reported to the `ReplicationmanagerImpl`.

When the `remove` method is called, `PastryGlue` simply finds the idBase that corresponds to the id and asks `P2PBroker` to remove all resources for that id.

In response to a call to `scan`, a copy of the set `PastryGlue` stores of ids stored locally is returned to the `ReplicationManagerImpl`.

An `exists` call makes `PastryGlue` check if the id sent as parameter exists in the local set of ids. The boolean result is returned.

The `existsInOverlay` call asks for the same thing as the `exists` call, only on a remote node. The method asks if it is possible to route a message to a node that has the key based on the key at the moment. The implementation of that method sends a message with the "exists" type towards the key asked for.

At the remote node, it is checked whether or not the key exists, and the result is sent back with an "exist-reply" message. Back at the local node, the answer from that message is relayed to the `ReplicationManagerImpl`.

If asked to `reInsert` a given key into the network, the local node retreives the service mirrors conforming to the idBase corresponding to the key and sends them towards the node currently root for the specified key. The remote node extracts the mirrors from the message and sends them to the `P2PBroker` which treats them as just being advertised.

5.6.3 Why using the basic PAST manager is insufficient

In QuA, it is possible to advertise multiple resources matching each QuA type. As explained before, the way service planning is performed in QuA requires QuA type conformance to be the basis of resource discovery. That is why the QuA types of the advertised service mirrors are used when building idBases for the mirrors, and this leads to a situation where we may have multiple resources belonging to each Pastry key.

As explained above, the `DefaultReplicationPolicy` decides which keys that are needed to be fetched on the local node by figuring out which of the keys in the remote set that are not present in the local set. If a key present in the remote set also is present in the local set, the replication policy will decide that no fetch is required for that key.

The reader may easier understand the situation from an example. For the sake of simplicity, we assume in the following example that the normal configuration of the P2P-broker is used, but the same problem would exist in any configuration of mapping resources to nodes that will work with QuA.

QuA capsule resources that are advertised map to the idBase "QuACapsule". Suppose the node responsible for the key that maps to that idBase is node A. Node R is within the reach for replication of that key. Node B advertises itself as a resource by sending an advertisement message for a QuA capsule resource. Node A receives the resource and adds it to its local storage. This is the first resource corresponding to that idBase that node A has received. The `ReplicationImpl` component at node R communicates with the `ReplicationImpl` component at node A, and gets hold of node A's idSet. It compares that to the local set, and finds that the key is needed to be fetched as it does not exist in the local set. Node R consequently fetches the resource for that key. Next, node C also decides to advertise itself as a service, and sends an advertisement message with a service mirror describing itself as a QuA capsule. Node A receives the message and adds the mirror to the set of mirrors held locally corresponding to the "QuACapsule" idBase. That set now contains two mirrors. The set at node R still only contains one. Now the `ReplicationImpl` component node R decides that it is time to update its replicas again, and asks for the idSet of node A. It sends the idSet of node A and node R to the `difference` method of the `DefaultReplicationPolicy` object. The `DefaultReplicationPolicy` checks that all keys contained in node A's keyset also are contained in node R's keyset, and decides that no fetching is needed. But we know that in this situation, the right thing to do for node R would be to fetch resources for the key based on the QuA capsule type again, as the set of mirrors contained at node R is different from the one kept at node A.

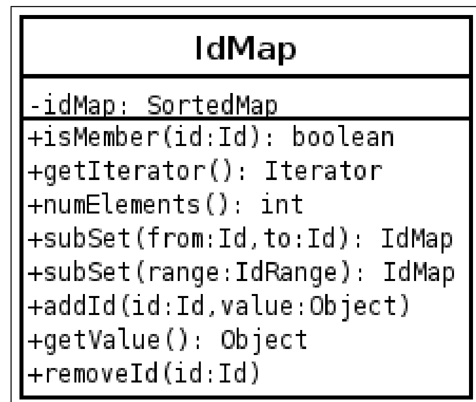

Figure 5.6: The IdMap class

Luckily, one would think, the `ReplicationPolicy` is pluggable in the FreePastry replication manager architecture. Unfortunately, things turned out not that simple. Although the `ReplicationPolicy` component is pluggable, that does not help, as the information being sent to the `difference` method is insufficient to do the comparison in any other way than the way it is done in `DefaultReplicationPolicy`. The only information kept in the `IdSet` objects sent to it is a set of Pastry ids. The only operation that is sensible to do on two sets of Pastry ids is an equality comparison.

It quickly became apparent that there was a need of providing a replication policy like the one used in FreePastry with more information.

5.6.4 The solution

To provide the `differece` method of the replication policy with more information, we will have to change the parameters sent to it. In fact, the need arises for a new data structure to replace the IdSet. The reader should consider that it is the IdSet class that is incapable of containing sufficient information. The IdMap class shown in figure[9] 5.6 contains a map instead of set, but otherwise the same methods as the IdSet class of FreePastry. As the class contains a Map, it is possible to add additional information on each key in the keyset contained at each node that can be used by the replication policy when deciding which keys to fetch! In fact, as what is put into the map are plain Java Objects, an architecture built around an IdMap class would support any replication policy without the rest of the replication architecture needing to be aware of it. Any information needed to make a application specific judgement on what keys to replicate in the `difference` method of the replication policy component can be stored in the hash map. By creating only a new replication policy that can figure out how to use that information for each application, the replication architecture can be made completely generic and application independent.

Now one might think that the path is straight and clear from here. Simply

[9]Again, only the most relevant information is shown in the figure.

replace the IdSet with the IdMap, build a new replication policy, and support it in PastryGlue, and all will work. Unfortunately, that lack of foresight permeated the entire implementation of the replication architecture of FreePastry. It seems as they have simply not thought of the possibility that anyone would want to *replace* their pluggable replication policy component. As a result, a total revision of each method in all classes shown in figure 5.5 has been performed for the implementation of `P2P-broker`. In addition, all the three message classes used in FreePastry's implementation had to be reconstructed to be able to support IdMaps instead of IdSets.

After re-implementing the FreePastry replication architecture, a choice of how to compute which keys need fetching needed to be made. Because of the limited time available to do the implementation and the surprise of the amount of work that had to be done to re-implement the replication architecture, a simple solution was chosen. The `P2PBrokerReplicationPolicy` uses a count of how many mirrors that are associated with a key at the local and remote node as basis for comparison, and makes a simple equality check. If the counts are equal, the key will not be fetched. If the counts are not equal, the local node will fetch resources associated with that key from the remote node.

To support that replication policy, a few changes was made to `PastryGlue`. First, the data structure holding the local set of keys is now changed to hold a map over keys and how many service mirrors associated with each key, called `replicaIdMap`. The number of mirrors associated with each key is updated by the `P2PBroker` by a special function call called `updateValue` each time it receives new mirrors. Second, the `ReplicationManagerClient` interface was changed to support IdMaps instead of IdSets, as `PastryGlue` now communicates with `P2PBrokerReplicationManager` instead of FreePastry's ReplicationManager. The changed replication code due to the substitution of IdSet for IdMap and the new replication policy is visualized by figure[10] 5.7. PastryGlue now supports a new `scan` function called `scanWithCount` and a new `exists` function called `existsAndEquals`. The new scan function takes the `replicaIdMap` and uses the internal `IdMap` function `subSet(range)` to create a new IdMap to return, that contains all keys within the range and all asso-caited counts. The new exists function checks whether the key exists, and if it does, if the count is the same as the `value` argument.

5.6.5 Re-querying for resources when response yields 0 service mirrors

When membership of the P2P-broker network changes due to joining or leaving nodes, the replication scheme can not start working towards getting the system back into a consistent state before the routing tables of FreePastry are updated. The reason is that the replication manager uses the routing tables to configure which nodes that should keep which keys, and it will not detect the arrival or departure of nodes within replication reach for a key before the underlying routing layer does. Therefore, it will arise situations where, for a limited amount of time, a node `A` that receives a message regarding a key `k`

[10]This figure shows only the actual changes from figure 5.5

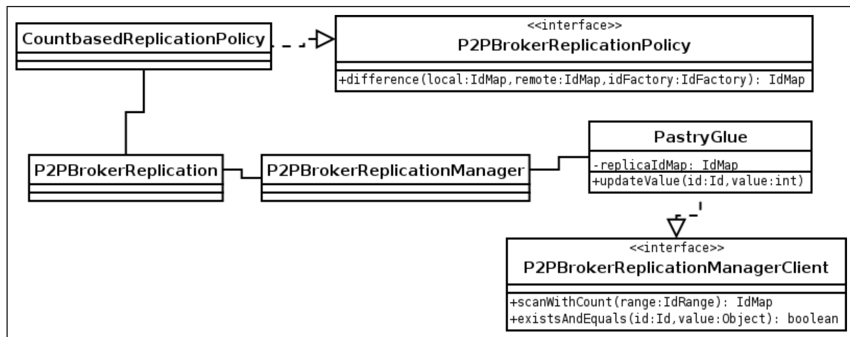

Figure 5.7: The fundamental changes to the replication architecture

may not yet have received the service mirrors that already are in the network that corresponds to the key. This will only last for as long as it takes for the replication scheme to calculate that node A should fetch service mirrors for key k. Still, it is possible that during this short period of time, a query message arrives at node A asking for resources for key k. Node A will not have any way of figuring out that although it does not currently have any resources for this key, it should have had them in storage. Consequently, node A will respond with a result message containing no service mirrors.

A temporary fix for that problem has been implemented in the P2P-broker. When the querying node B receives an empty set of service mirrors from node A after asking for resources of the type corresponding to key k, it will *re-send* the original query message after a delay. The node will do this up to a total of 5 times, if it does not receive any service mirrors, to compensate for the possibility of multiple nodes in turn assuming responsibility of key k within a short period of time.

5.7 Possible improvements

As the implementation of P2P-broker as presented in this chapter was done within a limited period of time, some shortcuts were made. Although it does work functionally, there are a few improvements that can be made that has the potential of increasing the effectiveness of the implementation.

5.7.1 Forward method / Caching

The Application interface implemented by PastryGlue contains a method called forward which is called just before a message which is not intended for that node arrives at the node. With some effort, this method can be used to exploit Pastry's locality properties, described in section 2.6, by letting replicating nodes answer query messages instead of routing them on. In addition to exploiting the locality properties, this effort would also create a caching mechanism, as well as further distribute querying between the nodes of Pastry by allowing replicating nodes for a key to share the burden of resolving queries

for that key. However, to see any great effect of this, the system would have to be deployed in a large scale environment where nodes are diverse in location. Therefore, improving this was not a priority in this implementation.

5.7.2 Difference method

The `difference` function of the currently deployed replication policy bases its decitions on which keys to fetch on a count of service mirrors associated with the keys on nodes. This approach was chosen only because of the limited time available, and the fact that it was easy to implement. There are situations where this method would fail. Situations where the count can be the same on two nodes, although the set of mirrors are different. Still, for these situations to appear in practice some bad luck must occur, and even if it happens, the system will repair itself as soon as a new service mirror for that type is advertised. For the situation where the count is equal, but not the set of mirrors, to arise, a sequence of events like the following must happen. First, a service mirror must be advertised to node A, which at the time is responsible for key k. Then, a new node, node B, must arrive that has one mirror less than node A for key k, and assume responsibility for the key k. Next, a new service mirror must be advertised for key k to node B *before* the replication architecture at node B realises that node A has more mirrors than it has itself. This was not seen as likely enough to happen to spend any time implementing an alternative method.

Further, if the `implementationBroker` interface allowed for removal of specific service mirrors, and not only all that comply to a specific QuA type, as it probably will in the future, removing one mirror from the set at the node responsible for the idBase, will cause problems in the replication algorithm as the replication will tell the responsible node that it has to few mirrors and should fetch the mirror it just recently deleted. As removal of single mirrors is a feature not used in any of the QuA test applications, it was not taken into account when considering implementing decisionmaking in the `difference` function of the replication policy.

Still, there are a couple of good alternatives. Instead of calculating a count of mirrors and shipping that around in the IdMap, one could calculate the byte size of the mirrors held at each node, or even compute a hash over some information extracted from each mirror. For example, it is possible to use QuA names for this, as every service mirror are given its own QuA name when they are advertised to the P2P-broker. The size-check in bytes seems to be the most reasonable way, as the calculation of a comparison component should be as light-weight as possible in terms of computing time. Still there is a possiblity of that failing too, due to differences in memory consumption by the storage of service mirrors on different nodes that run on different platforms. However, as the current implementation is in Java, it should not be a problem.

5.7.3 Fetching during replication

Each time a fetch operation is performed in the current implementation, the complete set of service mirrors are fetched from the remote node to the local node, even though only the service mirrors contained in the remote set that are

not present in the local set are needed. To improve the efficiency of fetching, the "fetch" messages sent from the local node to the remote node must specify which service mirrors the local set contains so that the remote node can be able to send only the service mirrors that are present in the remote set but not in the local set. Doing this will significantly decrease the size of fetch-reply messages when the number of service mirrors advertised for given keys is high. To be able to globally identify service mirrors, the QuA names attached to service mirrors in the advertisement phase as described in section 5.4.4 can be used. When creating a "fetch" message, all the QuA names for the service mirrors held locally must be compiled into the message. At the remote node, the set of service mirrors kept for that key must be filtered against the QuA names supplied by the "fetch" message. Only the service mirrors kept at the remote node that does not have a QuA name equal to any of the names supplied by the "fetch" message are sent back.

Chapter 6

Testing the broker

In this chapter we will examine how to test the P2P-broker. In the problem statement of section 1.2, we stated as a goal to try to find out whether or not the positive traits of peer-to-peer technology can be retained in the context of QuA resource discovery. The combined efforts of this chapter, chapter 7 and 8 will cover the discussion on that topic.

In this chapter, we will first examine the environment in which the tests are run in section 6.1. In section 6.2 the choice of tests to be run is discussed. Sections 6.3, 6.4 and 6.5 covers the design and setup for the individual tests in detail.

6.1 Test environment

Deploying and compiling all components needed for the P2P-broker is a cumbersome task, as both Java, Ant, a copy of the FreePastry library files and QuA needs to be installed in order to run the P2P-broker. Because of this, and the limited amount of available time, it was decided to run all tests on a single machine. As the performance of a peer-to-peer based system compared to distributed systems built on other architectural models like the client-server model, is based on distributing the load on actual machines, this decision rules out using time as a performance indicator.

All tests described in this chapter are run on a computer with the Intel Core 2 Duo 2,38 Ghz processor, and 3 gigabytes of RAM, using the Ubuntu linux distribution, running SUN's Java Standard Edition compiler and virtual machine version 1.6.0[1] build 105. The implementation of P2P-broker used is the one described in chapter 5. As for configuration of the P2P-broker, the normal configuration with the basic id-base generator will be used. Replication factor is set to 4, which means that the four closest nodes to the responsible node are told to replicate the resources.

[1] Also known as Java 6

6.2 What to test

The main positive characteristics of peer-to-peer technology are the ability to scale, the ability to self-organize and be independent of any central entity, and the possibilities for building in robustness to node failure, as the system is not dependent on any single entity in order to operate.

In section 1.2.1, a few sub goals were defined, and it was stated that by reaching the sub goals, the main problem statement of the thesis could be solved. The third sub goal, which is the focus of this chapter, chapter 7 and 8, is to find out to which degree the peer-to-peer system properties of scalability, self-organization and robustness to node failure have been retainded in the design and implementation of our P2P-broker.

In this thesis we will use three tests to address the three properties of peer-to-peer technology. Through the distribution test we will investigate the P2P-broker's ability to scale. The self-organization property will partly be investigated by the joining nodes, and partly by the departing nodes test, and the robustness property will be covered by the departing nodes test.

A peer-to-peer system's ability to self-organize independently of a central entity at the overlay network layer means the network's ability to divide responsibility for keys between them, and maintain enough routing state to consistently route messages to the correct nodes when nodes are constantly joining and leaving the network. To show that the P2P-broker has the ability to self organize, we must show that as nodes are joining and leaving, the P2P-broker is still able to have the responsible node at any time reply to a query for resources with the right data. By the design and implementation of the broker, this property, both in the case of joining and leaving nodes, are handled by the replication mechanism, described in sections 4.4 and 5.6. That mechanism is also the one making sure that the P2P-broker is robust to node failure with regard to delivering a consistent service. Therefore, the joining nodes test will cover the self-organization property in the case of joining nodes. The departing nodes test will cover the self-organization property in the case of departing nodes, as well as the system's ability to be robust with regard to node failure.

In chapter 7 we take a closer look at each experiment run with each of the three tests described in this chapter, as well as the results from each experiment.

6.3 Distribution test

Results from the distribution test will indicate the P2P-broker's ability to scale to a large number of resources and nodes as a distributed system. As all tests are decided to be run at a single computer, it would be impossible to observe any performance gains of distributing load on several instances of the P2P-broker by the actual time advertisements and queries take.

The design of the test described here is based on the assumption that if the system manages to distribute resources relatively evenly between the nodes participating in the system, it will have a fairly good ability to scale. This is believed to be a reasonable assumption because it is already shown by [26] that the Pastry routing algorithm has a good ability to scale with regard to message

delivery to nodes responsible for keys, so the actual scalability of the underlying routing layer is obvious. Further, the P2P-broker only hosts resources on the node that is responsible for the corresponding key, in addition to the keys it is responsible for replicating resources for, and the node responsible of hosting a service mirrors for a resource type is also the one that answers queries for resources of that type. The fact that filtering of resources is done at the root node for a resource, and not at the querying node, will also improve system scalability if we can observe that root node responsibility for resources are well distributed.

6.3.1 Test setup

For this test, an initial network of N nodes will be set up. These nodes will be given some time to establish themselves before continuing. The number of nodes will vary in experiments to give us an impression on how good the distribution is in networks of different sizes. Next, a simple test program that continuously uses QuA to advertise resources of random QuA types will be started at one or a few nodes. Each node in the network will keep a log over resources advertised that they are responsible for, and that they are responsible for replicating. After a preset number of resources are advertised, the nodes will be halted so that the logs can be investigated. The number of resources advertised will also be varied to observe the distribution, but will in the largest experiments be in the order of tens of thousands. From graphs created from the data of the logs, we will in chapter 7 see how well the resources distributes on the participating nodes.

So what is *good distribution*? The deviations of observations around a mean is a good indication on the closeness of observations in value. In statistics, the *standard deviation*, often denoted σ, is used to indicate the spread of observations. The *standard deviation* is the square root of the *variance*, which is the average of the square of all deviations from the sample mean.

A good distribution would be characterized by a small standard deviation in the observed count of resources that each node is responsible for. Exactly how small the standard deviation needs to be is hard to tell, but must be seen in relation to the value of the mean sample.

6.3.2 Expected results

The optimal result would be that the resources are distributed completely evenly between the nodes. However, this can not be expected. Distribution of keys in Pastry follow the secure hashing algorithm used by FreePastry. As the number of both nodes and advertised keys in a network approaches infinity, completely even distribution could be expected, but as with all statistics, the reliability drops with the size of the sample. What we can expect is to follow the laws of distribution laid down by Pastry and the cryptographic hashing algorithm it uses to generate keys and nodeIds, which means a relatively even distribution with statistical variances.

6.3.3 Limitations of the test

The test will not cover the cases where an uneven amount of resources are associated with each QuA type. For instance, in a real scenario, the number of advertised resources of the QuA capsule type may be particularly high. This is not a problem with the design of this test, as it is a known problem to which a solution is not proposed in the design or the implementation of the broker, and must be the target of further research.

6.4 Joining nodes

The purpose of the joining nodes test is to see how well a network of P2P-brokers reorganize themselves to cope with a stream of P2P-brokers joining the network. As nodes join the network, Pastry makes sure that the routing tables are updated so that the invariant that the closest node to a message's key in the id-space receives the message at any time is kept. To show that the P2P-broker is self-organizing in the case of joining nodes, we need to show that it builds on the property of Pastry by ensuring that when asked for resources corresponding to a key, the answering node answers with the resources held within the system that corresponds to the key.

6.4.1 Test setup

For the test to be easy to monitor, the test setup should have a low number of initially inserted nodes, and contain a low number of resources. Experiments with this test will have a starting network of 10 nodes. Then a new node is launched with a program that announces 10 resources of different fictional QuA types to the now 11 node strong network. After all resources have successfully been inserted into the network, a stream of N joining nodes will be started. Each joining node will ask for each of the 10 resources once, and keep a log-file of which node that answered the request for resources on each key, as well as checking if it was a successful request. After all of the N nodes have joined and asked for the resources, the log files will be joined to find a trace of each resource telling which node that answered requests for that resource at any time during the test.

6.4.2 Expected results

Because so few nodes are started initially, we expect to see a change of responsibility for each of the resources as nodes with IDs closer to the resource's key joins the network. As time goes by, we also expect to see that ID of the node answering the request for a resource gets closer and closer to the resource key.

6.5 Departing nodes

By letting nodes depart gracefully, we will be able to observe the P2P-broker's ability to self-organize in the case of node departure. But by letting the nodes die without warning, we will in addition be able to observe the P2P'broker's

robustness to node failure. As both of these properties are handled by the replication scheme of the P2P-broker, we find it reasonable to cover both properties with one test.

6.5.1 Test setup

For this test, we will start N nodes in an initial network. When all N nodes have completed the joining procedure, a program that advertises 3 resources of different fictional QuA types will be started. After advertisement of the three resources, the program will constantly ask for resources of those three types. The P2P-broker at the hosting node of the program will constantly log the answer to each query, to see which node answered the query for which key at what time, as well as logging if the answer was a success or not, and the number of times query messages had to be re-sent to get a result. As soon as all N nodes have joined the network, they will agree upon a time when they will start to "kill" themselves. That is, they will leave the network without prior notice.

Coordination on time in a distributed system is a difficult task and a big subject, and we will not go into that matter here. In the context of this test, it is not necessary that the start their timers at the *exact* same time as it is a random interval that they are supposed to die within anyway, which will be explained soon. But it is important that they agree upon a time with variations that are within only a few seconds.

For this test, a simple negotiation protocol based on a simple multicast scheme was implemented. When starting, all nodes send a message to the node responsible for a predefined key. The responsible node keeps track of all incoming messages, and which node that sent them. When all nodes have registered themselves, the node responsible for the key sequentially sends messages to all registered nodes. Upon receiving that message, the nodes start their timers.

Each node will die at a random time within a configurable random interval from the time when they received the message. At the end of the test, the only live node will be the logging node, and if the P2P-broker operates correctly, all resources should be maintained by that node at that time.

The test will be run multiple times with different random intervals set to try to stress the system. There are some system parameters that give indication as to a starting point when trying out different random intervals:

- The replication factor `k`.
- The FreePastry leaf set poll interval `p`
- The time it takes for the replication scheme to recalculate which nodes are responsible for replicating which keys, and make sure they do so.

The replication factor `k` is a known factor set to 4 in the experiments, and the leaf set poll interval `p` of FreePastry is, according to the FreePastry team, set to 30 seconds, and is not a configurable parameter. The replication scheme should be able to operate correctly if not the `k+1` nodes holding a resource dies within the time it takes Pastry to realize that immediate node neighbours have

left the network, and the replication scheme to re-establish the invariant that 4 nodes in addition to the root node holds the resources for any given key. By examining changes in the leaf set, the replication manager makes its decisions on which resources to replicate. FreePastry detects and reports changes in the leaf set by sending polling messages at the poll interval p.

In other words, if on node dies every 6 seconds on average, the system would not be able to keep the replication up to date even if:

1. The network latency and message transfer time is 0.

2. The time to recalculate the replication responsibility is 0.

3. The random distribution on time of death has a flat distribution.

Obviously, none of these statements are true. The network latency and the time the replication manager needs to recalculate responsibility are both unknown. The distribution time of death of nodes is indeed random. All of this must be taken into account, and the nodes must have a much lower death-frequency than every 6 seconds if the system should be expected to survive. In section 7.3, the performed experiments with this test will be described, and we will see what interval the system did and did not handle.

6.5.2 Expected results

As long as the replication scheme is working, we can expect to see that the nodes answering requests for the individual resources change as nodes leave the network. If the replication scheme at some point is unable to keep up the pace with the dying nodes, we will expect to see that from that point, the resource is no longer available in the network.

6.5.3 Limitations of the test

If nodes were deployed on multiple computers in a "real" network, network response times when dealing with replication would play a role that is impossible to show in our testing environment. Long network response times could cause problems to the replication scheme in situations where many nodes with adjacent nodeIds are constantly leaving the network unexpectedly. It could happen that the resources are not transferred to new replicating nodes quick enough to keep the invariant that k nodes are replicating each resource, even though the invariant would have been kept if the network response and data transfer time was zero. Over time, the resources may disappear from the network as a result of this. However, the network response time will be quite low in small networks, and only grow out of proportion to the replication timeouts in large scale, internet based, deployment where Pastry's locality properties would ensure that nodes that are close in nodeId space are diverse in network location, which should lower the probability of many nodes with adjacend nodeIds failing at the same time happening.

Chapter 7

Experiments

In this chapter, the performed experiments are presented, as are the results produced from the experiments.

We start by examining the results from the distribution test experiments in section 7.1, experiments using the joining nodes test are shown in section 7.2 and experiments for the departing nodes test are shown in section 7.3. Evaluation of the P2P-broker based on the results from the tests is done in chapter 8.

7.1 Distribution

Node nr	Root node	Replicated	Both
0	524	476	1000
1	476	524	1000

Table 7.1: Distribution of 1000 resources on 2 nodes.

Results for a total of 4 experiments performed using the distribution test are presented in this section. Figures 7.1, 7.2, 7.3 and table 7.1 show the results of the test. All 3 figures have the same structure. The nodes are shown along the x-axis, and the count of the number of resources they are responsible for is along the y-axis.

The x-axis label says “node Id”, which is a bit imprecise, as the program used to create the graphs, “gnuplot" had problems with the hexadecimal node Id values. Instead, all nodeIds were substituted with a number from 0 and up, indicating the placement of each node, relative to the other nodes, in the node Id space.

As is shown in the key of the figures, the blue line shows the number of resources each node is directly responsible, often called root-node, for. The green line shows the number of resources each node is responsible for replicating. The red line shows the number of resources each node is responsible for in total.

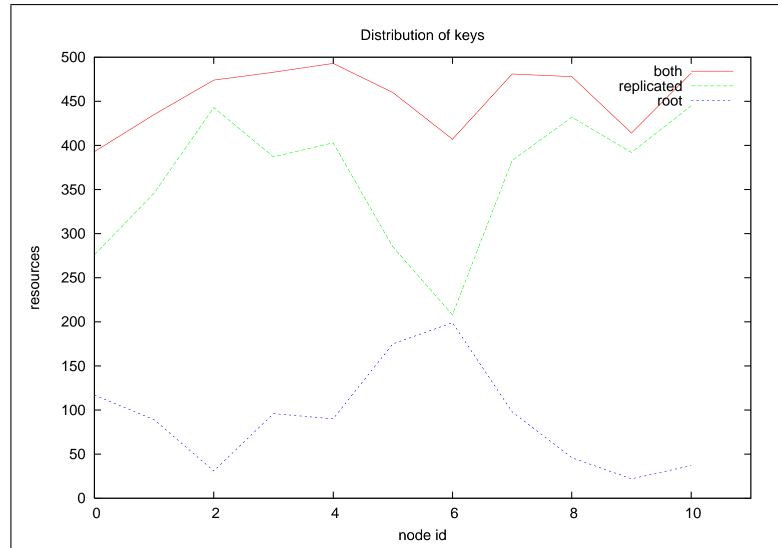

Figure 7.1: *Distribution test. 10 nodes sharing 1000 service mirrors.*

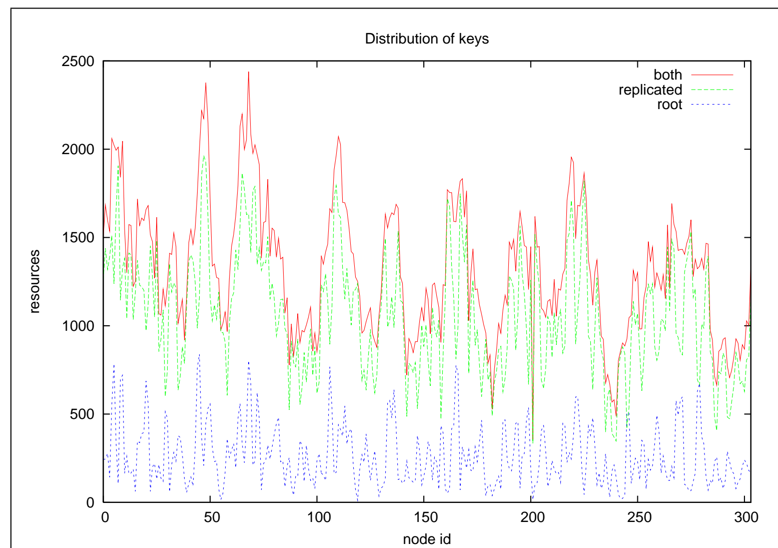

Figure 7.2: *Distribution test. 300 nodes sharing 81k service mirrors.*

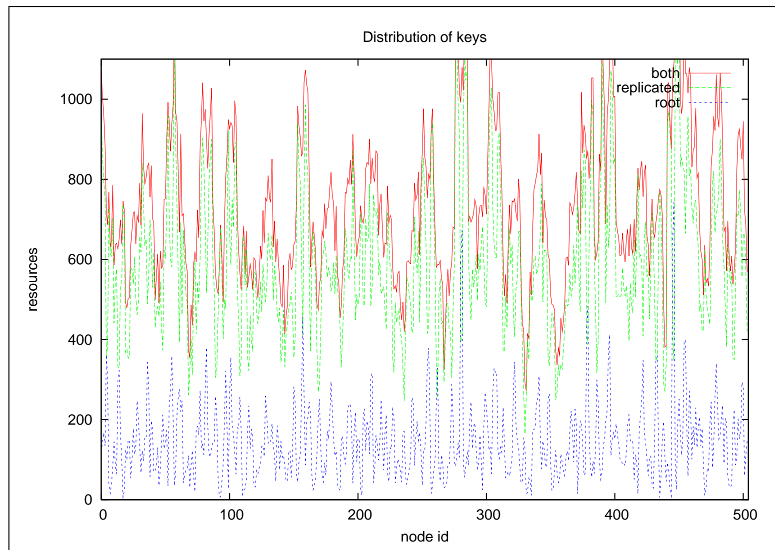

Figure 7.3: *Distribution test. 500 nodes sharing 73k service mirrors.*

First, only two nodes were interconnected to create a network, and a number of 1000 resources was advertised to the system. Results from that test are found in table 7.1. When the network was increased to 10 nodes, the distribution of resources was as shown in figure 7.1.

For the experiments where more than 100 resources were advertised, the advertising program had to take pauses to allow FreePastry to catch up. The advertising program is capable of announcing hundreds of service mirrors per second on the test-computer. FreePastry nodes only have a message buffer of 128 messages, and need some time to send messages even when they are bound for local machines.

The reason for this is that creating new objects in Java is a task that takes much less time than it does for a FreePastry node to get messages in its outgoing message buffer underway. Especially when a lot of nodes are running on the same computer, probably contesting for time on the same network interface.

Therefore, for the last two tests, when creating networks of 300 and 500 nodes, and wanting to advertise a total of 100000 service mirrors, 4 advertising programs were used. Each one advertising 25000 service mirrors in batches of 25 with a 50 second break just to be sure that FreePastry was allowed enough time to recover between announcements. With the 300 nodes test, about 81000 service mirrors were advertised after 14 hours had passed. Unfortunately, one of the advertising programs locked up after advertising about 6000 service mirrors, so the test stopped at advertising 81000 resources. The 500 nodes test experienced the same problems with two of the four programs, and only about 73000 service mirrors were advertised. Figure7.2 and 7.3 show the results of the 300 and 500 nodes tests respectively.

Multiple programs were used to speed up advertising. It would be possible to speed it up even more by letting all participating nodes advertise messages, but that would create problems with logging of the distribution of resource responsibility since the responsibility for resources would be constantly changing as some nodes would be joining while others were advertising. The only sensible solution was to allow for a core of nodes in the network stabilize themselves before letting a few nodes connect and advertise resources.

Tables 7.2, 7.3, 7.4 and 7.5 show a summary of a statistical analysis on the distribution from each experiment. The tables show the mean, standard deviation, max and min values for the number of mirrors nodes are responsible for, replicators for, and both.

Distribution of	Mean	Standard deviation	Max	Min
`Responsible`	500	33.94	524	476
`Replicating`	500	33.94	524	476
`Both`	1000	0	1000	1000

Table 7.2: Statistics of the distribution test with 2 nodes experiment.

Distribution of	Mean	Standard deviation	Max	Min
`Responsible`	90.90	57.25	199	22
`Replicating`	363.63	76.98	445	208
`Both`	454.54	35.74	493	393

Table 7.3: Statistics of the distribution test with 10 nodes experiment.

Distribution of	Mean	Standard deviation	Max	Min
`Responsible`	264.14	175.77	839	12
`Replicating`	1057.31	337.79	1964	335
`Both`	1321.46	375.02	2439	349

Table 7.4: Statistics of the distribution test with 300 nodes experiment.

7.2 Joining nodes

Only one experiment was run with the joining nodes test setup. A starting network of 10 nodes, plus one that advertised the 10 resources was set up, and a stream of 100 nodes joined the network as described in 6.4. The join rate is about 1 node per second. In figure 7.4 and 7.5 we can follow a total of 6 resources (3 in each graph), as the responsibilities for them are shifted between nodes as time goes by and more nodes join the network. For example, the "qua/Printer" resource traced in figure 7.5 changes root node 5 times. Every time the root node changes, the key of the new root node is closer to the resource key than the last root node. The arrows show the resource key location in the id space for each resource.

Distribution of	Mean	Standard deviation	Max	Min
`Responsible`	146.33	98.99	742	6
`Replicating`	587.65	183.42	1207	164
`Both`	733.98	187.74	1458	272

Table 7.5: Statistics of the distribution test with 500 nodes experiment.

At the x-axis, we have timestamps from when answers to the requests for the resources are received. The values are left out to make the graphs more lucid. Values along the y-axis are base 10 representations of the nodeIds of the responding nodes at each lookup. These graphs only show which node answered each request, not the success of the request.

For all graphs presented in this and the next section, each cross, star or vertical mark on the graphs represents one logged response to a query at that time. Sometimes, when responses are logged frequently, these marks will get very close, and it will appear as though the line turns from a thin line to a thick line. In other words, a thin line with no marks on it, means that no response was logged in this period of time. A thick line means that during this period of time, responses were logged all the time.

Figures 7.6, 7.7 and 7.8 show the success of each request for the three resources followed in fiugre 7.5. Again, timestamps are along the x-axis, but this time the y-axis shows only a count value. The red graph in each figure shows the number of retries that had to be attempted at the broker level to get a resource. The green graph shows how many resources that the response message to each request contained. Since it only was one resource advertised of each resource type, this value will never be larger than one. A zero, combined with the red graph rising to the value of 5, indicates that the system gave up on finding resources of that type. This did not happen in any of the experiments with the joining nodes test, but i did happen in some of the experiments with the departing nodes test, one of which is described in the next section.

Each of the figures 7.6, 7.7 and 7.8 should be seen in relation to the graph for the same resource in figure 7.5.

7.3 Departing nodes

A number of experiments with the departing nodes test was run with different numbers of nodes and different random intervals for the nodes to die within. In this section, results from two of these experiments are presented. In each of the presented experiments, a starting network of 100 nodes was running, before a single node, node A, that advertised three resources of different types and started to repeatedly ask for resources of the three resource types. In figure 7.9 and 7.13 we can follow the three resources from the start by seeing which node responded to the requests from node A as the 100 nodes starts to leave the network. At the end of the graphs, it is node A that is answering, as it is the only node left alive. In the experiment of figure 7.9 the random interval was set to 6000 seconds so that one node died every 60 seconds on average. Figure 7.13 show the experiment where the random interval was set to 3000

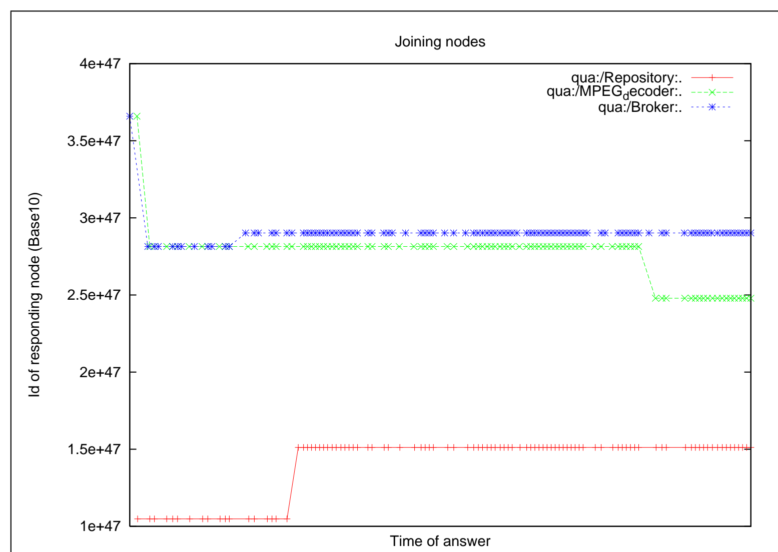

Figure 7.4: *Tracing of the resources "qua/Repository", "qua/MPEGdecoder" and "qua/Broker" through time where 100 nodes join an initial network of 11 nodes. Each graph trace responsibility for a resource through time. The arrows mark where the key of the resource its colour matches belongs in the id space. The time of each response is represented by crosses, stars, and vertical lines on the graphs. In the case where responses are very close in time, the crosses, stars, or vertical lines are plotted so close that they can appear to be a thicker line.*

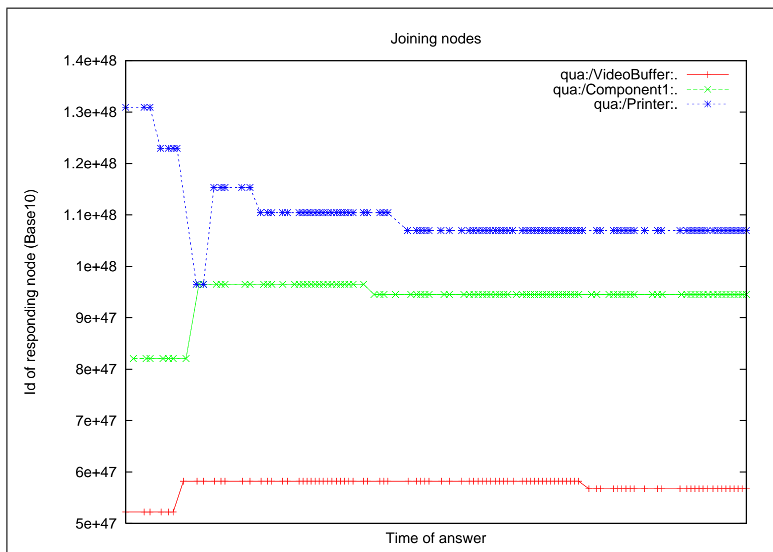

Figure 7.5: *Tracing of the resources "qua/videoBuffer", "qua/Component1" and "qua/printer" through time where 100 nodes join an initial network of 11 nodes. Each graph trace responsibility for a resource through time. Along the x-axis time passes, and the y-axis shows ids in the id space in base 10. The arrows mark where the key of the resource its colour matches belongs in the id space. The time of each response is represented by crosses, stars, and vertical lines on the graphs. In the case where responses are very close in time, the crosses, stars, or vertical lines are plotted so close that they can appear to be a thicker line.*

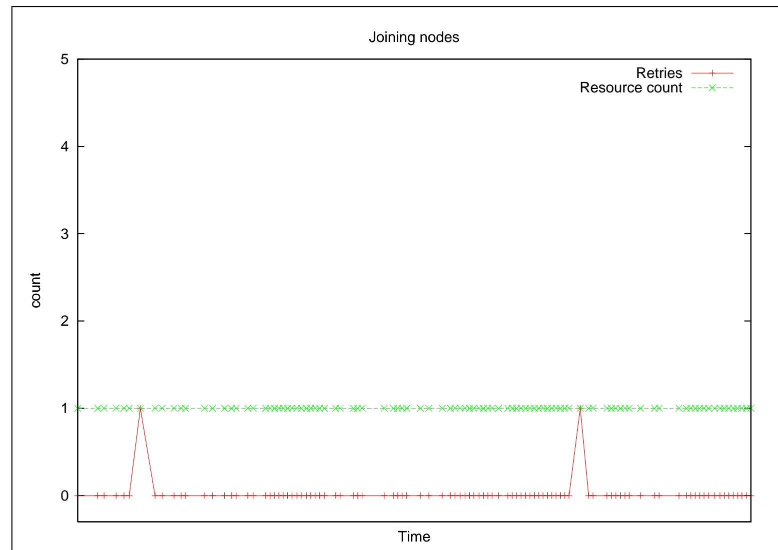

Figure 7.6: *Retries when asking for resource with key [0x6331...], representing "qua/videoBuffer" while 100 nodes join network. This graph should be seen in relation to the red graph of figure 7.5, representing the "qua/videoBuffer". In this figure, we have time at the x-axis, and a count at the y-axis. Crosses and vertical lines mark observations. We see that the re-try count rose to 1 both times the resource got a new responsible node.*

seconds, and nodes died every 30 seconds on average. The graps are intended to be interpreted in the same way as the graphs in figures 7.4 and 7.5 from the joining nodes test. However, in the joining nodes test, both figures of this type are from the same experiment, tracing a total of 6 resource types. In the two figures from this experiment we follow the same 3 resource types in each figure, but in two different experiments, with different sets of nodes involved.

What is interesting to see in figure 7.9 and 7.13 compared to the tracing figures from the joining nodes test is the long intervals we sometimes see where no response from a query is logged. As explained in the previous section, in the graphs from experiments on joining and departing nodes tests, a thin line indicates that no response was logged in this period of time. This will represent the time it takes for FreePastry to re-organize.

Figures 7.11, 7.12 and 7.10 show the success of queries for the resources traced in figure 7.9 in the same way explained for figures 7.6, 7.7 and 7.8 for the joining nodes test. The corresponding graphs for the experiment traced in figure 7.13 are shown in figure Figures 7.15, 7.16 and 7.14.

Also in the graphs of these figures, we see the long thin lines representing periods of no logged responses. These periods match the periods for the tracing graphs that they represent. We also see that in the figures from the experiment where nodes dies every 60 seconds on average, the retry graph never rises above 0, even when there are long periods of no logged response. This

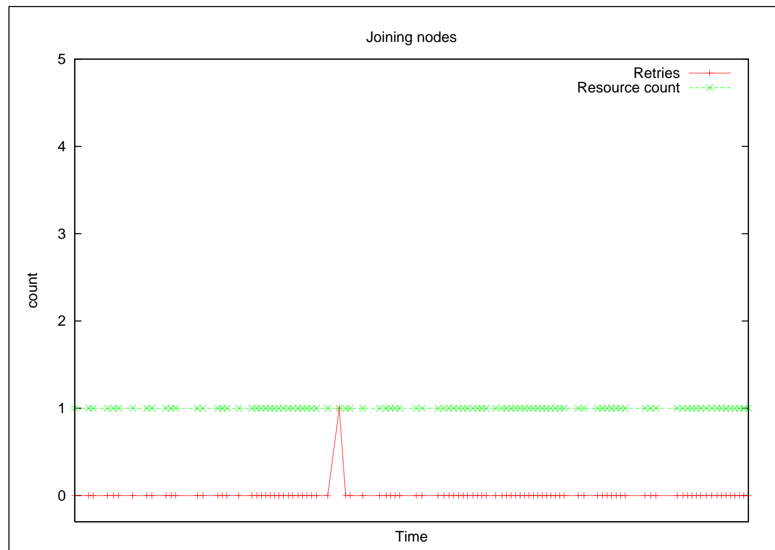

Figure 7.7: *Retries when asking for resource with key [0xA51C...], representing "qua/Component1" while 100 nodes join network. This graph should be seen in relation to the green graph of figure 7.5, representing the "qua/Component1". In this figure, we have time at the x-axis, and a count at the y-axis. Crosses and vertical lines mark observations. We see that the re-try count rose to 1 one of the two times the resource got a new responsible node.*

is because the retry counter indicates retries at the broker level, which is only used when a response arrives with no service mirrors in it. The long thin lines indicates that FreePastry does not deliver messages to the right broker.

In figure 7.14 and 7.16, we see the retries counter rise to 5 at the same time as the resource counter (green line) falls to 0. This means that the resource became unavailable after the last observation of a green 1 and a red 0. From the point where the retries counter rose to 5 and resource sunk to 0, we see that responses are logged regularly. This indicates that there is a node that answers the requests for the resource, but the answers does not contain any service mirrors.

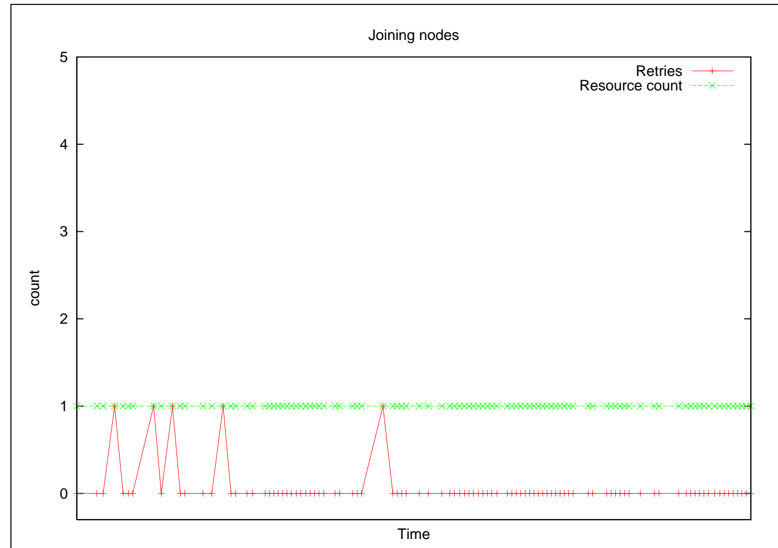

Figure 7.8: *Retries when asking for resource with key [0xBC77. . .], representing "qua/printer" while 100 nodes join network. This graph should be seen in relation to the blue graph of figure 7.5, representing the "qua/Printer". In this figure, we have time at the x-axis, and a count at the y-axis. Crosses and vertical lines mark observations. We see that the re-try count rose to 1 all five times the resource got a new responsible node.*

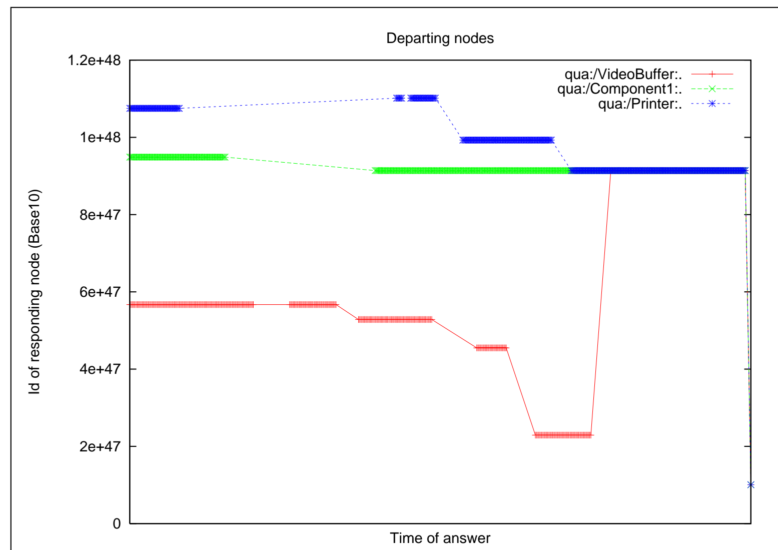

Figure 7.9: *Tracing 3 resources in a network of 100 departing nodes, where one node dies every 60 seconds on average. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently. The arrows show the approximate position of the resource keys in the id space.*

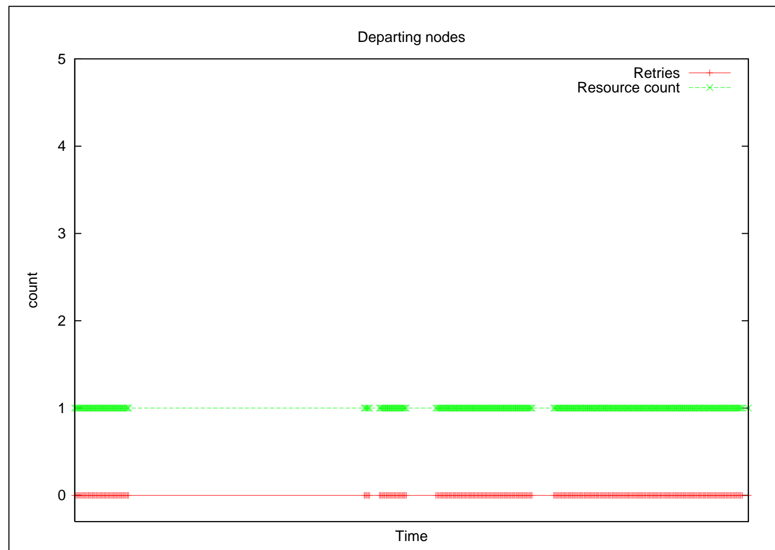

Figure 7.10: *Retries when asking for resource with key [0xBC77...], representing "qua/printer" with nodes dying every 60 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently.*

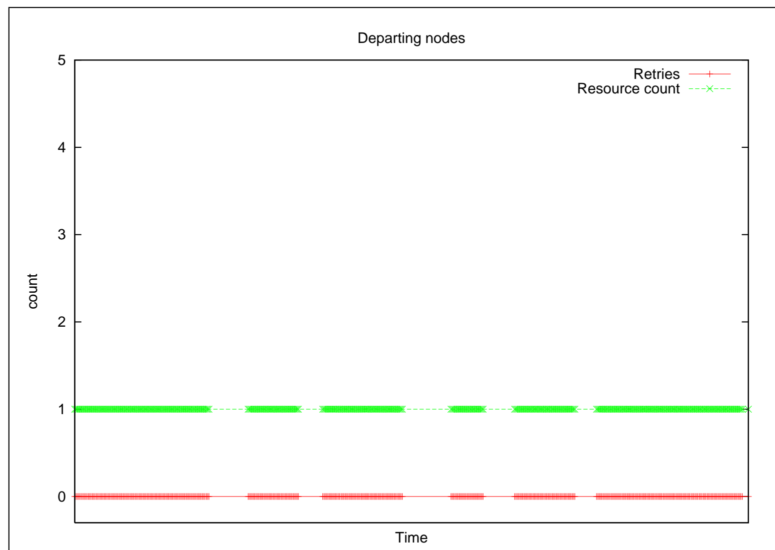

Figure 7.11: *Retries when asking for resource with key [0x6331...], representing "qua/videoBuffer" with nodes dying every 60 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently.*

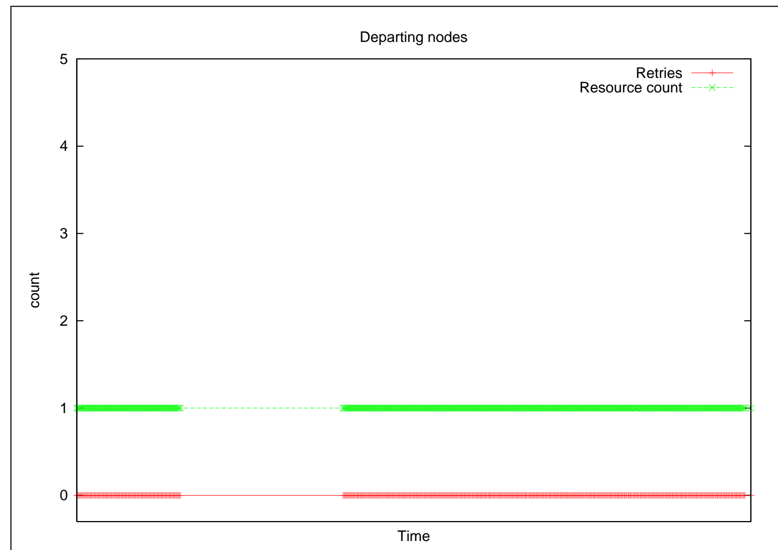

Figure 7.12: *Retries when asking for resource with key [0xA51C...], representing "qua/Component1" with nodes dying every 60 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently.*

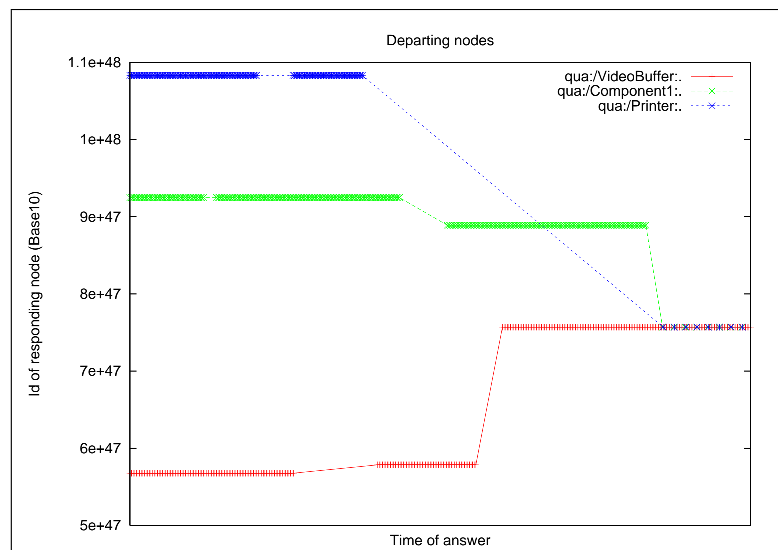

Figure 7.13: *Tracing 3 resources in a network of 100 departing nodes, where one node dies every 30 seconds on average. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently. The arrows show the approximate position of the resource keys in the id space.*

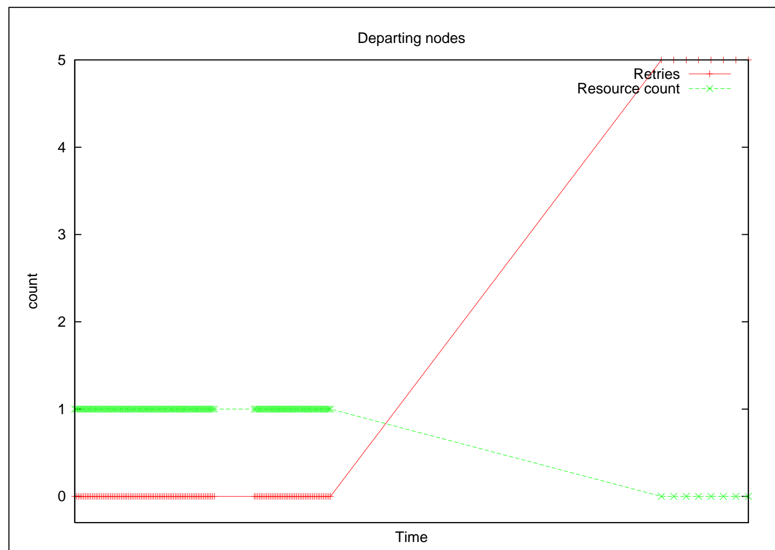

Figure 7.14: *Retries when asking for resource with key [0xBC77...], representing "qua/printer" with nodes dying every 30 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently. The retries counter rising to 5 combined with the resource count falling to 0 means the response logged was a failed response containing no resource.*

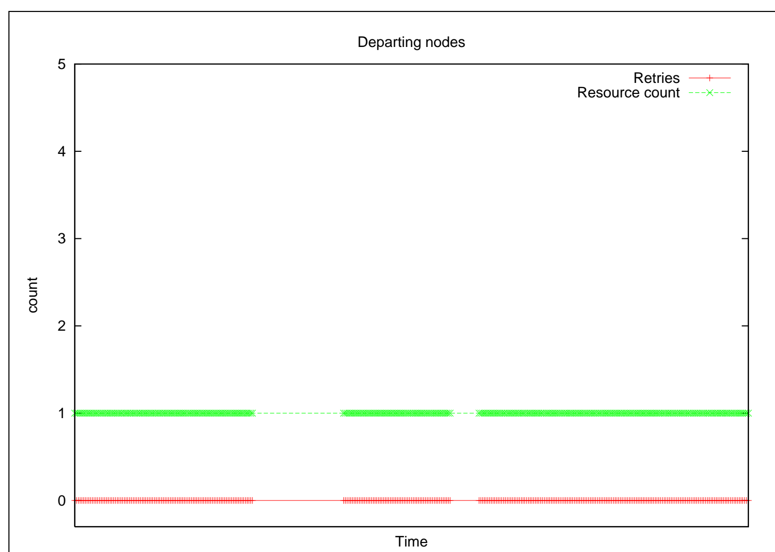

Figure 7.15: *Retries when asking for resource with key [0x6331...], representing "qua/videoBuffer" with nodes dying every 30 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently.*

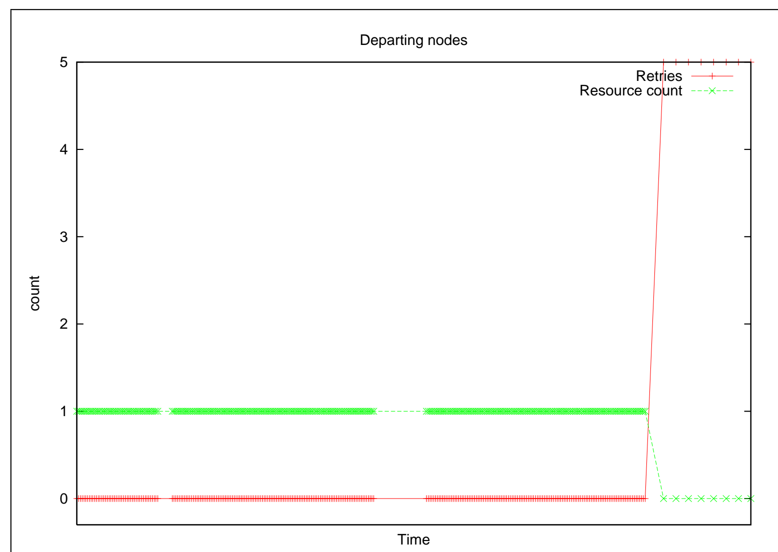

Figure 7.16: *Retries when asking for resource with key [0xA51C...], representing "qua/Component1" with nodes dying every 30 seconds on avg. Long thin lines means that no response was logged in this time period. Thick lines means that responses have been logged frequently. The retries counter rising to 5 combined with the resource count falling to 0 means the response logged was a failed response containing no resource.*

Chapter 8

Evaluating the broker

In this chapter we will evaluate the design and implementation of the P2P-broker in light of the experiments and results from chapter 7. We will evaluate the P2P-broker with respect to the three properties identified in the problem statement of section 1.2; scalability in section 8.1, self-organization in section 8.2 and robustness in section 8.3.

8.1 Distribution and scalability

Recall from section 6.3.2 that we expected the distribution of resources on the participating nodes in the experiments to be a relatively even distribution with statistical variances. To be able to get an impression of how the system operates under different conditions, we varied the size of the networks in our experiments.

As we see from figure 7.1, the resources are not nearly spread out evenly, but they are distributed among all nodes. Every node is assigned some resources, and most nodes are assigned between 50 and 150 resources. However, node 6 has an unusual high peak at almost 200 resources, and 3 nodes are below 50 resources. This means that node 6 would receive more than 4 times the requests that the three nodes with lowest resource count, had this been a real deployment. Still, compared to the alternative, where a single server would receive all requests for all 1000 resources, we must admit that the P2P-broker does distribute the load amongst all nodes.

Further, by looking at the count of resources each node is replicating, we see that because the nodes close to a root node for a resource in the Id space replicate resources for that node, the nodes that did not get responsibility for many resources, got responsibility for replicating a lot of resources. So the total load of storing the resources, represented by the "both" graph in the figures, is well distributed in a network of this size.

The impression that the "both" graph has better distribution than the "root node" graph is strengthened by both figure 7.2 and 7.3. However, even the "both" graph in both these figures have some distinctive peaks. From tables 7.4 and 7.5, we can also see that the max and min values are far out from the mean value in the 300 nodes and 500 nodes tests.

One might wonder why we do not see a more even distribution of resources to nodes when as much as 76000 and 81000 resources of random QuA types are advertised to a network of nodes using a hashing algorithm to decide the location. A probable explanation is that the resources are assigned keys that are fairly well distributed in the id space, but that the Ids of the nodes are clustered so that some nodes get responsibility for a larger range of keys than other nodes. Figure 8.1 illustrates the situation. As we know from section 2.6, keys in Pastry are assigned to the Pastry node that is closest in the id space. In practice this means that adjacent pastry nodes split the id space that lies between them in half. In figure 8.1, there are three classifications of nodes based on their placement in the global id space:

- Nodes marked C are clustered.
- Nodes marked B are at the border of a cluster of nodes
- nodes marked A are all by themselves with a huge space in each direction of the circular id space.

In practice, the nodes marked C will have only a short path in each direction of the id space to control. Nodes marked B will have a short path in one direction that they share with a C node, and a longer path in another direction which they share equally with an A node, which in total gives a larger id space to be responsible for than nodes marked C. The A nodes have a large space in both direction, that they share with B nodes, giving the A nodes the largest responsibility area for keys of all nodes in the figure.

Now imagine that 100000 resources are assigned keys that are evenly distributed in the id space. Even with a totally even distribution in the id space, the distribution of the resources to nodes would be uneven as nodes marked A in the figure gets responsibility of more keys than nodes marked B, which in turn gets responsibility of more keys and thus resources than nodes marked C.

From the statistical summaries of tables 7.3, 7.4 and 7.5 we can observe that the standard deviation to mean ratio is quite high for counts of resources that individual nodes are root node for. However, the standard deviation to mean ratio decreases and becomes quite acceptable in distribution of replicated resources on nodes. For the total count, the results are even better. These data support the theory described above.

The min and max values for the "replicated" and "both" rows of the 10 nodes experiment shown in table 7.3 are actually quite good. The reason for this is probably that the replication factor is not that much smaller than the total number of nodes in this experiment, so it is hard not to get anything to replicate, even if nodes are clustered, as the clusters are probably not larger than 5 nodes.

The experiment that the results of table 7.1 represents is from a network of 2 nodes, with advertisement of 1000 resources. The experiment is included because it shows that even in the smallest of networks, the P2P-broker is able to distribute resources evenly amongst nodes. It may seem strange that this

Figure 8.1: Clustering of nodes

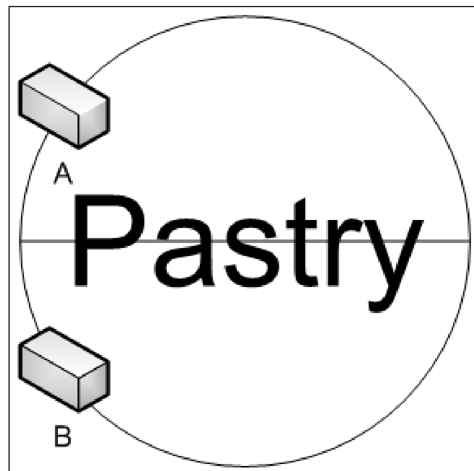

Figure 8.2: Two Pastry nodes share the Id-space of any network evenly

network actually gets a more even distribution of resources between participating nodes than what we see in the other experiments as one would think that the distribution of nodes in the id space gets more even as the network is increased in size. However, a two node Pastry network represents a special case. Consider figure 8.2 showing a random network of two Pastry nodes. As mentioned above, adjacent Pastry nodes effectively divide the id space between them evenly. In a two node Pastry network, the nodes are adjacent in both directions. That means that they share the id space in both directions with the same node, effectively taking control of one half each of the id space. In figure 8.2, if we think of the line going through the center of the circle as dividing the id space in two, node A will be responsible for any resource given a key that belongs in the northern half of the circle. Node B will be responsible for the rest. In practice, any way two nodes organize themselves into a network, they will end up sharing the total id space equally.

The conclusion must be that the P2P-broker seems to be able to take advantage of Pastry's properties with respect to distribution of resources, and that the variances we see between the number of resources each node is responsible for are as expected considering the way Pastry distributes responsibilities for keys to nodes through the secure hashing algorithm. This will make the P2P-broker distributed system able to scale as the numbers of participants, and through that the number of resources, increases.

8.2 Self-organization

The self organization property will in the case of the P2P-broker mean its ability to let the participating node in a network agree upon which node is responsible for storing service mirrors for given types, and respond to discovery messages for resources of those types, without the coordination from a central entity, even with participants of the network constantly joining and leaving. The P2P-

broker's ability to self organize in that manner in the light of the results from sections 7.2 and 7.3 is the topic of this section.

How a network of P2P-brokers self organize when nodes are joining is shown by the experiment described in section 7.2 and the related figures. Figures 7.4 and 7.5 show how responsibility of resources of 6 different QuA types change as more nodes join the network. From the graphs, we can for instance see that resources of the type "qua/repository" change host only one time, while resources of the type "qua/printer" changes root node 5 times. Unfortunately, "gnuplot", which was the program used to plot the test had problems showing the graphs correctly with hexadecimal values, and the ids at the y-axis had to be converted to base 10. This means that it is hard to see where the key of each resource belongs on the y-axis. However, it is possible to see that for each time one of the graphs change value at the y-axis, it has a shorter jump in y-axis values than for the last jump, unless it is a further jump in the same direction as the last. This represents the root node homing in on the key of the resource as more nodes join the network. This is in alignment with the Pastry property that the node with nodeId closest to a key in the id space at any time is responsible for that key, and indicates that the P2P-broker follows Pastry's self organization properties.

However, although figures 7.4 and 7.5 show which node that answers messages regarding individual resources at given times, they do not show the degree of success of the queries. As all queries for the three resources in figure 7.4 turned out to be successful, graphs to trace the success of the individual resources from that figure is left out of this thesis to save space. Figure 7.6, 7.7 and 7.8 shows how many times the P2P-broker had to resend the query message to get a non-empty set of service mirrors for each response to requests for individual resources. The figure maps directly to the results of figure 7.5. From these figures we see that each time the resources "qua/printer" and "qua/videoBuffer" change root node, one query message is re-sent once from the querying P2P-broker. On the first re-send the returning message contains the service mirror that should be returned, and the lookup for the resource is a success, although it takes a little extra time due to the resend. Queries for the resources of the "qua/Component1" type only provoke one re-send even though it changes root node two times.

The reason why each root node shift provokes a re-send in this experiment lies in the description of the joining nodes test that was used. The network started as a stable network of 11 nodes, containing 1 service mirror for each of 10 resource types after the advertising node had joined. Then a stream of joining nodes are asking for each of the 10 resources in succession. In other words, each time a resource with key `k` gets a new root node `R`, the first thing `R` does is to ask for each of the ten resources, including the one it just became root node for. In practice, the replication manager at `R` will never have time to get hold of the service mirrors for `k` before `R` gets time to send the query message to itself asking for resources for type `k`. In the experiment, the replication manager managed to get up to date before the first re-send of the message every time this occurred.

From figure 7.9 and 7.13 we can also observe the nodes that answer to

queries for specific resources change as nodes leave the network. This indicates that the P2P-broker is able to follow Pastry's way of re-organizing also with regard to node failure. Whether the data associated with a key are kept in the system or not is a robustness issue, and will be discussed in the next section.

The mechanism ensuring message reliability discussed in section 5.5.3 did not re-send any messages during the experiments for joining and departing nodes, although it has been observed kicking in during implementation testing. That means that all messages were safely delivered by FreePastry during the described experiments. The author believes that the reason why the mechanism was left with no work is that all experiments were run at a single machine, and that messages probably only disappear from FreePastry under special circumstances with high load on nodes operating over a network that experience packet loss.

Effects on service planning

From figures 7.9, 7.13, 7.11, 7.12, 7.10, 7.15, 7.16 and 7.14 we see that there are long intervals between logged responses on all graphs. These thin lines represent temporary, or if combined with 5 retries and 0 in resource count, permanent, unavailability of resources. Permanent unavailability means that the replication scheme ensuring robustness has failed, as will be discussed in the next section. However, temporary unavailability is also a problem, and it is an issue of self-organization. The FreePastry network takes some time to re-organize to node failure, and that is causing problems.

Consider a situation where the QuA service planner has planned a QoS sensitive application. The context changes, and the adaptation manager instructs the service planner to perform a re-planning of the service. The service planner asks the P2P-broker running in the QuA capsule for a component resource of some type. The P2P-broker tries to resolve the query, but changes in the FreePastry network membership forces the query to take a long time to resolve. The P2P-broker gets the job done, but not fast enough. Such a delay may force the QoS sensitive application to malfunction.

This is a distributed systems problem that is not easily solved. A definite solution to the problem will not be proposed here, but there are some alternative ways to go in the search for a solution:

- The waiting time can be reduced by decreasing the message timeout values for messages in the FreePastry implementation, but that will come at a cost. If it is set too low, messages will be re-sent even if an answer to the message is underway to often, and much resources will be wasted on sending duplicate messages.
- It is also possible to let the service planner be aware of such problems, and let it handle it by trying to plan the application without components of that type if a timeout for the query to the broker is reached.
- A third solution will be to let the P2P-broker handle it by informing the service planner if things take too long, and asking the service planner if

it wants to wait for the answer or not. This solution requires an extension of the ImplementationBroker interface of QuA.

8.3 Robustness

The robustness of a distributed system with regard to node or link failure is in this thesis seen as its ability to continue to operate correctly and continue to deliver the service that is expected of it. For the P2P-broker this means to be able to continue to discover the resources advertised to it at any time despite node and link failure.

The departing nodes test shows two things;

- How the network of P2P-brokers re-organize to respond to node failure as discussed above.
- How robust the network is to data loss during node failure.

In this section, we concentrate on robustness. Two experiments are presented running the departing nodes test. Figure 7.9 shows which nodes that responded to requests for resources over time in an experiment where a starting network of 100 nodes slowly dies, by letting one node die every 60 seconds on average. We see from the graphs that eventually, one node answers requests for all resources. That node is the only node that was not set to die in the experiment. The marks on each graph of the figures represent the actual time of reception for each message.

As mentioned in the previous section, there are some intervals between bursts of these marks on all graphs in figure 7.9 . These are areas of time where no response to messages sent for resources are received, i.e. areas of time where the resources were unreachable from the querying node. But as we see from the graph, the resources were available at later time. Actually, figure 7.11, 7.12 and 7.10 show that all resources were available right until the experiment was stopped, at which point all nodes except the querying node had died.

As the resources were only temporarily unavailable, we know that the replication scheme did not fail in its task. A probable explanation for the temporary unavailable resources is that FreePastry tried to send a message via a route that was unavailable because of a recent node failure, or possibly multiple node failures. When FreePastry discovered that the nodes had failed, it routed the message along a different path. This took some time, as messages had to time out at the FreePastry layer.

In the cases where the temporary unavailability of a resource matches the change of root node for that resource, it is probable that the root node died, and that it took time for Pastry to realize this and send the message to one of the replicating nodes. Implementing a caching functionality using the “forward” method of the “Application” interface of FreePastry, discussed in section 5.3, could provide a solution for this problem, reducing the delay in cases where one of the replicating nodes are on the routing path to the root node that just had died. This is an interesting subject for further work to improve the value of the P2P-broker implementation.

In the second experiment with the departing nodes test that is described in section 7.3, we increased the rate at which nodes “die” to try to force a situation where the the replication scheme crumbled, and see what happened.

From the figures 7.13, 7.14, 7.15 and 7.16 , we can see that in this experiment the “qua/vidoBuffer” resource changes host two times, and survives the experiment. The “qua/Component1” resource changes host once, but when that host dies, no replicating nodes are alive, and the resource is no longer available in the system. Similarly, the “qua/Printer” resource also disappears from the system, but it does so at an earlier stage, even before it has changed root node at all.

From the experiments, we have seen that the replication scheme works. It is capable of moving resources around to new nodes to try to keep the invariant that k+1 nodes should hold every resource in the system. However, it is obvious that as nodes come and go in a distributed system, there will be times where the invariant does not hold, and fewer than k+1 nodes hold some resource. The job of the replication scheme is to detect situations where this happens, and react to it, getting the system back into a consistent state. It is also clear that there is a limit to how frequent changes of node membership the replication scheme can handle, as it have to be based on communication between participants in the distributed system.

In [26], an implementation of Pastry is tested to see how it handles node failure. In the experiment described, 10% of the nodes in a Pastry network of 100000 nodes fail simultaneously. According to [26], the lazy repair algorithms of Pastry allows the network to, over time, completely recover from this drop. Still, it is uncertain if Pastry could handle a larger percentage of nodes suddenly dropping, but it is likely that it would be mentioned in the cited paper if results indicating that was available.

Further, the implementation of Pastry used in [26] is not the same as FreePastry, and the results are not 100% comparable. Even so, we can not expect any results from the experiments with the P2P-broker to be better than the results from [26], as the implementation of the replication scheme is reliant upon FreePastry’s ability to correctly route messages and update routing state information.

In section 6.5 we discussed the parameters that theoretically should be a base for understanding how frequent node failures the system should handle.

On one hand, we might think that the system should have handled node failures at a rate much closer to the limits set from those parameters, which would mean a death rate closer to one every 6 seconds.

On the other hand, it must be taken into account that the individual nodes’ time of departure is determined randomly, and that the total number of nodes in the system can be assumed to be very low at the end of each experiment. With high probability, clustering of the time of death for some nodes will happen during the experiment, and if this happens at the end of the experiment when only a few nodes are left alive, it is probable that the dying nodes will be close in node-id space.

Chapter 9

Conclusion and further work

In this chapter we will conclude the thesis and suggest further work. The conclusion will start by relating the thesis to the problem statement from section 1.2. Next it will cover the most interesting findings in the thesis. In the future work section, we will take a closer look at some things that are addressed in the thesis, but would need further work, and things that have not been addressed much but could be interesting topics in increasing the value of this research.

9.1 Conclusion

The purpose of the work of this thesis was to investigate the feasibility of using peer-to-peer technology in the resource discovery phase of operation of the QuA middleware. An answer to the main problem statement of the thesis has been reached by completing the three sub goals set in section 1.2.1:

- Through literature study we have found that, in principle, any structured overlay peer-to-peer technology can be used as basis for a peer-to-peer based implementation broker component in QuA.
- Based on structured peer-to-peer technology, a P2P-broker component has been designed, and implemented on top of an implementation of Pastry called FreePastry. The component is integrated with QuA, and is fully operational.
- Through experiments with the P2P-broker, we have seen how it operates in terms of scalability, self-organization and robustness.

We have seen in this thesis that it is possible to design and implement a peer-to-peer based implementation broker component, and through that shown that it is possible to solve middleware resource discovery with the use of peer-to-peer technology. Through experiments we have also seen that the implementation of the P2P-broker based on FreePastry is capable of leveraging Pastry's scalability and self-organization properties. In addition, replication has been built into the P2P-broker, and we have seen that it works as long as FreePastry is able to keep routing state up to date, but we have also seen that the replication scheme crumbles when FreePastry is put under pressure by killing nodes too frequently.

Although the P2P-broker implementation has inherited the self organization properties from FreePastry, we have seen that leaving nodes in some cases can provoke high latencies when asking for resources. This causes problems for service planning as described in section 8.2. Finding a solution to this problem is a topic for further work.

Experiences obtained with FreePastry show that the technology has a lot of promise, but some parts of the implementation seemed somewhat immature.

The Past replication manager delivered with FreePastry is not generic enough to provide replication for applications that assume the possibility of assigning more than one object to a Pastry key or Id. This thesis suggests a more generic solution that makes the replication manager able to support a wider range of applications by leveraging the pluggable replication policy of the replication manager. The solution requires a re-implementation of every class in the FreePastry replication package, as the data-carrying class IdSet has to be replaced with the more generic IdMap class proposed by this thesis.

9.2 Further work

Although the work of this thesis has been comprehensive, there are many topics and interesting subjects for further research. These topics have been divided into three groups. Some topics are design issues, some are implementation issues, while some issues are related to testing and experimenting with the existing implementation.

9.2.1 Design issues

Computers have varying capacity when it comes to physical resources like memory, computational power and storage space among others. In a peer-to-peer system, all participating entities are assumed to have the same capabilities, and responsibilities are distributed evenly between them. If a way could be found of allowing nodes to be part of a P2P-broker network without contributing to it, new ways of using the broker could be opened up. For instance, it would be possible to have a "backbone" of stationary computers providing a consistent resource discovery service, and let mobile users connect to the network to find resources without having to contribute storage space or processing power to the network.

In this thesis, a way of exploiting properties in dividing the search space when looking for resources of specific types and property values has been proposed. A theoretical view of how effective this solution will be in different circumstances has been discussed, but its practical use is not known. The author believes that, with more effort, even better ways of exploiting properties to optimize search time for resources can be found. For example, it might be possible to use multicast functionality to create hierarchical search spaces for service mirrors within the peer-to-peer network. Still, any solution will probably involve some kind of trade-off between total storage use in the network and the search time of individual queries.

When nodes leave the network unexpectedly, we have seen that high latencies in queries for resources can arise. This causes problems for service planning, and especially re-planning of QoS sensitive applications as discussed in section 8.2. To find a solution to this problem, one might have to look into the service planner, or at least an extension of the ImplementationBroker interface and role in the QuA architecture.

9.2.2 Implementation issues

Although the implementation of the P2P-broker presented in this thesis works, there are some issues that need improvement if it is to be deployed in "the real world".

- A more general way of calculating which keys that need replication at individual nodes is needed. The current solution works in the context of the current QuA implementation, but need work if the implementation broker role of QuA is to be extended, as described in section 5.7. For instance, QuA support for software evolution has revealed the need for functionality to remove single service mirrors describing services that are out-dated. The current solution is built on the assumption that individual service mirrors can not be removed, which is the case in the current implementation of QuA.
- To improve the efficiency of the replication scheme, a smarter way of retrieving service mirrors for replication is needed. As of now, nodes fetching replication objects for a key fetches all objects for a key, even if they already had some of them in storage. Smarter ways of doing this are sketched in section 5.7, and would significantly improve the efficiency of the replication scheme in a real deployment with QuA.
- As was discussed in chapter 8, the distribution of resources on nodes is better when we look at the total of root node responsibilities and replication responsibilities than when we just look at root node responsibilities. It was argued that this means that distribution of used storage space is quite good, but distribution of messages regarding resources, i.e. query messages resulting from "getMirrorsFor" calls to a P2P-broker, has a less good distribution. By using the "forward" method of the FreePastry "Application" interface, it is possible to build in a caching mechanism on the the nodes replicating resources for a key so that the k+1 nodes holding resources for a key also share responsibilities of answering messages for that key.

9.2.3 Testing issues

The experiments with the current implementation of P2P-broker described in this thesis are based on tests designed to be run on a single computer, and are run on a single computer. To learn more about how the P2P-broker would operate in a real distributed environment, it would be interesting to either deploy it in a large scale network like Planet-LAB, or to build a comprehensive network simulation environment to test the P2P-broker in. From such experiments it would be possible to find out:

- To which degree the P2P-broker is scalable in practice.
- If it is self-organizing in practice.
- The replication scheme works in a real networking environment, and the P2P-broker is robust in practice.

Further, there was no time in the work of this thesis to find out if the alternative configuration (as described in section 4.3.2 has its uses. By creating different scenarios and experiment with the P2P-broker in both configurations, it would be possible to learn more about the contexts where the alternative configuration could be useful. In addition it might be possible to observe where the effort should be concentrated to try to exploit properties to make queries faster.

Appendix A

Enclosed CD

On the very last page of this thesis (the inside of the back cover), a CD is appended where the contents include:

- A README file.
- The source code for a QuA build which includes the P2P-broker developed as part of this thesis.
- The FreePastry library files needed to run the P2P-broker.
- Other library files needed by the FreePastry library.
- The raw data from the experiments presented in chapter 7.

The README file contains further information on CD-content and how to use the software including:

- An overview of all of the contents of the CD.
- How to compile and configure the P2P-broker.
- How to get hold of SUN's JDK, and Apache's Ant, needed to compile and run the QuA with the P2P-broker.

Appendix B

Definitions and explanaitions of terms

This appendix contains definitions, explanaitions and clarifications of some terms that falls into one or more of these groups:

- Have different meanings in different contexts.
- Are introduced by this thesis.
- Are of special importance to this thesis.

The definitions, explanations and clarifications presented here are valid for the context of understanding the work of this thesis.

Alternative configuration An alternative way of mapping resource responsibility to nodes in the P2P-broker based on properties(See section 4.3.2).

Component A common definition of a component which originates from Clemens Szyperski is *"A unit of composition with contractually specified interfaces and explicit context dependencies only"*. See component based middleware, section 2.2.

Id An Id is a unique identifier in the global identifier space of a peer-to-peer system. When the term Id and not key is used, the term most commonly refers to a node. However, in this thesis, nodeId is the most commonly used term for an identifier for a node.

idBase A string representation that is the basis for building a key for a resource. The inBase is extracted from metainformation on the resource. Same as keybase.

idspace The global identifier space of a peer-to-peer network.

id range A subset of the idspace of a Pastry peer-to-peer network. Each node is responsible for an id range.

Key A key is a unique identifier in the global identifier space of a peer-to-peer system that has been associated with a resource or object.

keybase A string representation that is the basis for building a key for a resource. The keybase is extracted from metainformation on the resource. Same as idBase.

keyspace The global identifier space of a peer-to-peer network.

leaf set In Pastry networks: At an arbitrary node X, the set of nodes with the nodeIds closest to node X in the global identifier space.

Middleware A layer of software that lies between the operating system layer and the application layer, hence the name. See section 2.2.

Node In this thesis, what is referred to as a node is most often an entity that runs an instance of the P2P-broker application. In some cases in the implementation discussion, the meaning of the term is only the actual running FreePastry code, but it will be denoted as "the underlying node".

nodeId The unique identifier of a node in the global identifier space of a peer-to-peer system. In chapter 5, this may be the term used to refer to an object of the NodeId class of FreePastry.

Normal configuration The most basic and intuitive way of mapping resource responsibility to nodes in the P2P-broker (See section 4.3.1).

P2P-broker The name of the implementation of the peer-to-peer based implementation broker for QuA presented in this thesis.

P2PBroker The name of the class that implements the ImplementationBroker interface of QuA.

Peer Two entities are considered to be peers if they are equal in responsibilities and rights.

Peer-to-peer technology Technology for distributed systems that assumes that all participating entities are peers in their architectural models. Not all technology classified as peer-to-peer have a *true* peer-to-peer architectural model as explained in section 2.5.

QuA Name A global identifier in the QuA middleware. Has an url-like structure, and can be used in bindings.

Resource In computing, a resource can be anything from a piece of software, to a hardware resource. Resources are often modelled as *services* in middleware.

Resource discovery The process of discovering resources. More information in section 2.4.

Responsible node See explanation of *Root node*.

Root node A node is said to be root node for resources that it is responsible for in a peer-to-peer network. That is, messages sent to a key representing a resource are sent to the root node for that key. That node is in this thesis said to be responsible for the resource that the key represents.

Service discovery In this thesis, the same as resource discovery. The variation in terms is often based upon the modelling of resources as services in middleware.

Service mirror A metadata construct for services used in QuA to provide reflection in all phases of a components lifetime. See section 2.3.

Service planning The phase where services are planned for instantiation or reconfiguration in QuA. Works closely with the implementation broker. See section 2.3.

Bibliography

[1] Freepastry 1.4.4 javadoc.

[2] http://freepastry.org.

[3] S. Amundsen, K. Lund, F. Eliassen, and R. Staehli. Qua: Platform-managed qos for components architectures. In *Proceedings of Norwegian Informatics Conference (NIK)*. Tapir, 2004.

[4] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[5] Magdalena Balazinska, Hari Balakrishnan, and David Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of the International Conference on Pervasive Computing (Pervasive 2002)*, pages 195–210, Zurich, Switzerland, August 2002. Springer-Verlag Berlin Heidelberg.

[6] Mirion Bearman. *Tutorial on ODP Trading Function*. Faculty of Information Sciences Engineering. University of Canberra. Australia, 1997.

[7] Clip2. *The Gnutella Protocol Specification v0.4*. http://www9.limewire.com/developer/gnutella_protocol 0.4.pdf.

[8] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, 1989.

[9] George Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Consepts and design. 4th. ed.*, chapter 1 : Characteristics of distributed systems, pages 1–25. Addison-Wesley / Pearson Education, 2005.

[10] George Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Consepts and design. 4th. ed.*, chapter 20 : CORBA case study, pages 847–854. Addison-Wesley / Pearson Education, 2005.

[11] George Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Consepts and design. 4th. ed.*, chapter 16 : Mobile and ubiquitous computing, pages 657–719. Addison-Wesley / Pearson Education, 2005.

[12] George Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Consepts and design. 4th. ed.*, chapter 2 : System Models, pages 47–61. Addison-Wesley / Pearson Education, 2005.

[13] George Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems : Consepts and design. 4th. ed.*, chapter 10 : Peer-to-peer systems, pages 397–430. Addison-Wesley / Pearson Education, 2005.

[14] Gregory Craske, Zahir Tari, and Kiran R. Kumar. DOK-trader: A CORBA persistent trader with query routing facilities. In *International Symposium on Distributed Objects and Applications*, pages 230–240. IEEE CNF, 1999.

[15] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003. Springer Berlin / Heidelberg.

[16] Peter Druschel and Antony Rowstron. PAST: a large-scale, persistent peer-to-peer storage utility. In *Eighth Workshop onHot Topics in Operating Systems, 2001.*, pages 75 – 80. IEEE CNF, May 2001.

[17] F. Eliassen, E. Gjørven, V. S. W. Eide, and J. A. Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In Nalini Venkatasubramanian Geoff Coulson, editor, *The 5th annual Workshop on Adaptive and Reflective Middleware (ARM 2006)*, pages 1–6. ACM Press, 2006.

[18] E. Gjørven, F. Eliassen, K. Lund, V. S. W. Eide, and R. Staehli. Self-adaptive systems: A middleware managed approach. In *2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006)*. Springer, 2006.

[19] Oliver Heckmann, Axel Bock, Andreas Mauthe, and Ralf Steinmetz. The eDonkey File-Sharing Network. In *Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications, Informatik 2004*, September 2004. ftp://ftp.kom.e-technik.tu-darmstadt.de/pub/papers/HBMS04-1-paper.pdf.

[20] Jeff Hoye. *The FreePastry tutorial, version 1.3.* http://freepastry.org/FreePastry/tutorial/.

[21] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications, Surveys & Tutorials, IEEE*, 7, 2005.

[22] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM Press.

[23] Jan Newmarch. Jan newmarch's guide to jini technologies, 2006. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml.

[24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[25] Jordan Ritter. Why gnutella can't scale. no, really., 2001. May be found at: http://www.darkridge.com/ jpr5/doc/gnutella.html.

[26] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001.*, volume 2218, pages 329–350. Springer Berlin / Heidelberg, 2001.

[27] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM Press.

[28] Antony Rowstron, Anne marie Kermarrec, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication : Third International COST264 Workshop, NGC 2001, London, UK, November 7-9, 2001.*, volume 2233, pages 30–43. Springer Berlin / Heidelberg, June 25 2001.

[29] Sourceforge. *The Gnutella Protocol Specification v0.6*. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

[30] Richard Staehli, Frank Eliassen, and Sten Amundsen. Designing adaptive middleware for reuse. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 189–194, New York, NY, USA, 2004. ACM Press.

[31] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[32] SUN Microsystems. *Jini Architecture Specification v. 2.0*, 2003. http://www.sun.com/software/jini/specs/.

[33] SUN Microsystems. *Jini Discovery and Join Specification v. 3.0*, 2005. http://www.sun.com/software/jini/specs/.

[34] SUN Microsystems. *Jini Lookup Service Specification v. 1.1*, 2005. http://www.sun.com/software/jini/specs/.

[35] Zahir Tari and Gregory Craske. A query propagation approach to improve CORBA trading service scalability. In *International Conference on Distributed Computing Systems*, pages 504–511. IEEE CNF, 2000.

[36] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41 – 53, Jan 2004.