UNIVERSITY OF OSLO
Department of Informatics

Evaluation of resource distribution and discovery in the QuA middleware with focus on the peer-to-peer broker

Master thesis

30 credits

Øyvind Nordang

2nd June 2008

Abstract

QuA is a reflectiv middleware architecture with a implementation broker assisting the service planner in service planning by performing resource discovery. QuA supports pluggable core services and one of these services is the peer-to-peer broker. The peer-to-peer broker is a distributed approach to the normal server/client way of managing resource discovery. The resources are distributed among the participants in the peer-to-peer network making the network more resilient to resource loss than the normal client/server approach.

This thesis evaulates the peer-to-peer broker by looking at the distribution of resources and the disk space and bandwidth used by this. By making the searchable domain for each resource larger by using more disk space we see if the resource discovery time can be improved and how big the disk and bandwidth overhead is by doing this.

The evaluation of the peer-to-peer broker shows that by partitioning the searchable domain we do get some improvements in the resource discovery time, but at the expense of a large disk space usage and bandwidth overhead. Also the size of the searchable domain for a resource can become very large.

Acknowledgements

The last five months has been both very educational and challenging. Never before have I written something this extensive, and the road has been both fun and frustrating at times, but I have had support all the way.

First of all, I would like to thank Professor Frank Eliassen, my supervisor on this thesis. He has given me insightful and invaluable information about the topics related to this thesis. Also a thank to Johannes Oudenstad, the author of the thesis this work is based on, who has been avialable for questions I have had along the way.

Also I would like to thank my friends and family for beleving in me and giving me love and support. Whenever I lacked motivation I could always count on them to give me kind and encouraging words. And a special thanks to my girlfriend who has put up with me during these times. She has been of invaluable support to me.

Preface

This thesis is the final part of a two 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 30 credits in the Norwegian study weighing system, which means that it is worth one semester of studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As computing systems become larger and more complex, the idea of self-adapting system is beginning to make its way in to the system design. A self-adapting system is able to reason about itself at run-time and, when necessary, make changes to itself in order to better satisfy the current environment requirements. Traditionally distributed systems where build from the client-server architecture, but many distributed applications are now built from *peer-to-peer* (P2P) architectures. These systems consist of equal, autonomous peers joining when they want to and often leave without a warning and do not depend on a central server.

The Quality of Service Aware Architecture (QuA)[5] developed at Simula Research Laboratory is a self-adapting middleware[1] which supports planning-based adaptation. In QuA resources are called services and are planned and reconfigured by a component called the Service Planner. The Service Planner uses a broker (trader) to find the components needed to build the service and to find different configurations when reconfiguring the application. QuA has two brokers, the *Implementation Broker* and the *P2P Broker*. The P2P broker, which we are interested in, is a peer-to-peer implementation of the standard QuA broker. This means that there is no central server acting as a trader, but all participants are equal users and servers.

Resource discovery is an important part of self-adapting systems. The resources in QuA are called *service mirror*, which is an object reflecting the service behaviour, known as service type, and its implementation, called

[1]Middleware is software that serves as a link between two other applications.

blueprint. A service mirror also have properties described by key-value pairs which works as constraints on the type. Allowed property types in the map is determined by the QuA type specified by the service mirror.

The development of the P2P Broker was the main task for Johannes Oudenstad's thesis[4] called *"The design and evaluation of a QuA implementation broker based on peer-to-peer technology"* from the University of Oslo, Department of Informatics. He looked at the feasibility of using peer-to-peer technology to handle resource discovery in QuA and was motivated by the central server approach being vulnerable to error and that it does not scale well.

The peer-to-peer technology used by the new broker creates a structured overlay network where all participants are assigned an id from a circular name space. Each peer has control over a part of the name space equal to half the way to the closest neighbour in both directions of the ring. Each message sent out is assigned an id from the same name space and the peer which controls the part of the name space where the message id belongs receives the message.

In Oudenstad's thesis it was introduced a proposal on how to distributed the load generated by resource discovery at the cost of storage space. The idea is to make duplicates of a resource and give them different ids, with the result that the different duplicates end up at different peers.

The new broker did go through some testing, but because of limited time not as much as one would like. Motivated by this we investigate the performance of the P2P broker with focus on the possibility of distributing the load on each peer by distributing the queries as well.

1.2 Problem statement

This thesis focus on the evaluation of a peer-to-peer based way of discovering resources in QuA.

> *Is it possible to gain faster resource discovery in QuA by partitioning the searchable domain? Can this approach justify the bandwidth and disk space overhead?*

Partitioning the searchable domain means more of the same resource will be distributed based on all possible combinations of the search criteria. By doing this we automatically use more storage space and bandwidth.

1.3 Research method

To reach the goals defined in 1.2 the work has to be structured in a way that we can be sure that we get done in time. To do this we will rely on the scope defined in 1.4.

To get the basics of the technology that is used some reading and research is needed. This includes knowledge about how the P2P broker works, the theory behind peer-to-peer, distributed hash tables (DHT), Pastry and application testing.

Within *software engineering* (SE) it is a discipline called *software performance engineering* (SPE), it was first mentioned in 1981[6] by Connie U. Smith, which is one of the pioneers of SPE, and it has since been subject for many studies and books. SPE is a method for developing software that meets the desired performance goals. A more formal definition is given on Wikipedia[7]

> *Performance engineering encompasses the set of roles, skills, activities, practices, tools, and deliverables applied at every phase of the Systems Development Life cycle which ensures that a solution will be designed, implemented, and operationally supported to meet the non-functional requirements defined for the solution.*

There are two general approaches[8] under the SPE umbrella.

- The first and most common is purely *measurement-based.* It applies testing and tuning late in the development cycle when the system can be run and measured.

  This approach can again be divided in to two categories[9]

  - Internal techniques, which means that the measurement is done inside the system. One way to do this is by making changes to the source code and insert code that will measure performance. Typically this is used to measure time used in a loop or the time from one point in the code to another. Another way is to use runtime libraries, they are libraries used by the compiler to implement functions while the system is running.
  - External techniques observes the system from the outside and tries to gather measurement data without altering the source code. This can be done by using profiling, this is a way of dynamically analyze the system during runtime. Another way of doing

this is by using independent observers where the measuring point is separated from the system, typically used for network related measurements.

- The second one is *model-based* where measurement models are created early in the development cycle and used quantitative results from these models to adjust the architecture and design so that the measurement demands can be reached. A typical approach to this is the use of queuing networks, this is an approach where the computer system is represented as a series of queues.

Much have been written about SPE and the problems it is facing[10][11][12], but that will not be a subject in this thesis.

It is obvious which one of the two main approaches that have to be used here. The P2P broker is not finished, but to stay within the scope of this thesis we see the broker as late in the development cycle and will use the measurement-based approach together with the internal technique. This will mostly be done by inserting timers to time different aspects of the code and by logging the results of resource distribution and discovery.

The initial idea was to use a real network called Planet Lab[2] to test the P2P broker, and we had indications that we could get access to the network through one of the research groups at the University of Oslo, Department of Informatics. Unfortunately they no longer had access to Planet Lab but had planed to renew their subscription. But to do this they have to contribute to the network with a dedicated computer which have to meet certain specifications. So because it would take too long before everything were up and running we decided to abandon the Planet Lab idea and go back to testing on a local network.

1.4 Scope

Since this master thesis is done within a limited amount of time we have to set some limitations. Since this is a short master thesis the limitations has to be well defined and not span over a too large area. The introduction and problem statement gives an overview but here we will define the exact scope for this thesis. The main task for this thesis is to test and measure the performance of the P2P broker with focus on resource discovery. The limitations for this thesis are somewhat given by the limitations in Oudenstad's thesis.

[2]Planet Lab is a global research network that supports the development of network services.

P2P broker

The P2P broker is only a small part of a larger system but here we limit the scope to only be looking at the P2P broker since it is not a part of this thesis to suggest improvements outside the broker.

Testing

We will test the approach suggested in Oudenstads thesis where the query load is distributed and see if we gain any better quering time and if it can be defended by the extra amount of disk space that have to be used. This has not yet been implemented and we will have to make a working sample of this. To match the new method we also have to change how queries are build.

The code

When the P2P broker was created it used FreePastry version 1.4.4. Since then several new versions have come out and as this is written the version is at 2.0_03. It is likely to believe that several improvements have been done since version 1.4.4, but it will not be a part of the scope trying to upgrade.

This thesis is not based on programming, but still some code needs to be added. Modification of existing code or adding new code may be done where it is seen fit. This wil be documented inside the code and in this thesis if the modification is of such a magnitude that it is of importance to the reader.

Outdated resources

Resources referring to a dead node will not be taken in to consideration, this extends Oudenstad's scope of nodes not withdrawing their resource when leaving the distributed system. A suggested fix is to introduce timeouts for service mirrors leading the mirrors to be removed if they do not hear from their owner within a certain amount of time.

1.5 Summery of results

Through testing this thesis tries to evaluate the resource discovery in the QuA middleware with focus on the P2P Broker. After implementing a way of partitioning the searchable domain experiments were conducted to see if the resource discovery time improved and how much overhead got generated, both in bandwidth and storage space. The experiments show that by partitioning the search domain some improvements to the discovery time is gained, but it also shows a significant amount of overhead.

1.6 Thesis structural overview

This thesis have the following structure.

Chapter 2	gives some background information needed to understand the work of this thesis.
Chapter 3	gives a view on how the peer-to-peer broker is built and how it works. The explanation on how the partitioning of the searchable domain is going to be done is also presented here.
Chapter 4	gives some thoughts around the planning of the experiments and presents the test methods.
Chapter 5	presents the discoveries found in the experiments and discuss these.
Chapter 6	gives a conclusion on the findings and suggests further work.
Appendix A	contains a CD with the source code of both QuA and the P2P broker along with the raw data from the experiments.

Chapter 2

Background

This chapter will take a look at the work and the technologies this thesis is based on.

2.1 Peer-to-peer

The term peer-to-peer refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner. The resources encompass computing power, data (storage and content), network bandwidth, and presence (computers, human, and other resources)[13].

Peer-to-peer (P2P) computing provides an alternative to the traditional client/server architecture. With P2P computing, each participating computer, referred to as peer or node, has the role as client with a layer of server functionality. This allows the peer to act both as a client and server within the context of a given application. The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for dedicated servers. Most P2P systems builds a virtual network with their own routing mechanism. The topology of this overlay network and the routing mechanisms used have a significant influence on application properties such as performance, reliability, and scalability[13].

This section gives a summery of some of the protocols build with peer-to-peer technology.

There are several different architectures for P2P networks[14]:

Centralized A central server keeps track of what each peer is sharing along with information about the location of the peer (typically IP address).

The resource request is sent to the server to find out which node is holding the desired file. The disadvantage of this approach is that it scales poorly and also the central server is a single point of failure[1].

Decentralized but Structured These systems have no central server and is therefore called decentralized, but they have a great amount of structure in the overlay network[2]. In a highly structured system both the network topology and placement of files are precisely determined. This way queries in the system are routed very efficiently. An example of this approach is Pastry which is described in section 2.2.

Decentralized and Unstructured These are systems in which there is neither a centralized server nor any precise control over the overlay network or file placement.The placement of files is not based on any knowledge of the topology. To find a file the node queries its neighbours, typically with flooding where the query is propagated to all neighbours within a certain radius. These unstructured designs are extremely resilient to nodes entering and leaving the system. However, the current search mechanisms are extremely unscalable, generating large loads on the network participants.

P2P systems are often divided in to three generations[15], the first generation was launched by Napster, second generation refers to Freenet, Gnutella, Kazaa and BitTorrent. Pastry, Tapastry, CAN, Chord and Kademlia are all part of the third generation.

Napster

The popularity of P2P networking began with the launch of the Napster network in May 1999 and enabled users worldwide to share music with each other. As can be seen in figure 2.1 Napster is a large distributed storage system with a central index server. The server kept track of the files each peer was sharing along with the peers IP address but the actual files were stored on the participating peers. All requests were sent to the server which responded by sending a list of matching content back. The peer would typically pic a song from the list and a direct connection would be established between the peer holding the song and the peer requesting the song. At its peak

[1]Meaning where one part of a system will make the whole system fail.

[2]Overlay network is built on top of another network. In this case build on top of a physical network. Nodes in the overlay network are connected through virtual or logical links and one hop in the overlay network can be several hops in the underlying network.

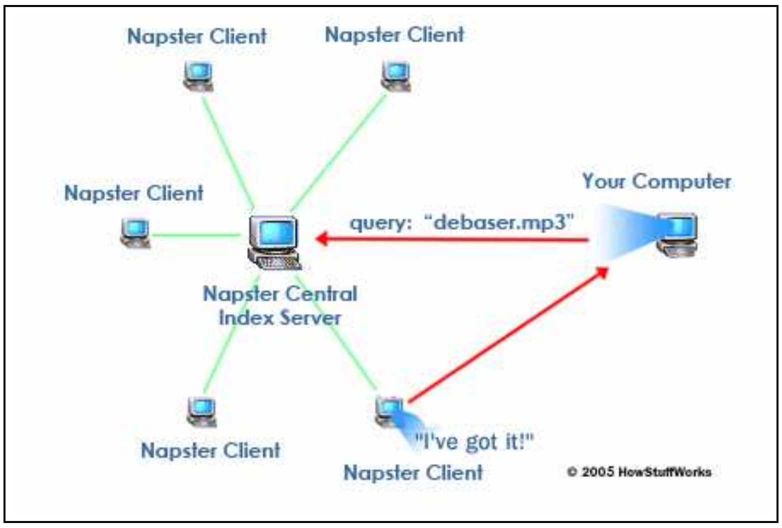


Figure 2.1: A view on how Napster works. Every user is connected to the central server. Taken from[1].

Napster had several millions users and thousands were swapping music files simultaneously[15].

In 2001 Napster was shut down because of ligal issues. Much of the music shared through Napster was copyright material and the copyright owners filed a law suit against the owners of Napster. Napster argued that they were not liable for what was shared since the actual transfer of files were solely between the peers. Their argument failed because the central server was seen as an important part of the file sharing process.[15]

Gnutella

Gnutella is an example of a unstructured and decentralized peer-to-peer protocol used to find files[16]. In figure 2.2 the blue dotted lines depicts a request for a file propagating through the system. The decentralized nature of Gnutella provides a degree of uncertainty, this is related to Gnutellas scalability. It does not scale well because the number of queries and the number of potential responses increases exponentially with each hop[13]. For example, if each node is connected to two others and the TTL[3] of the query is 7 (the default value in Gnutella) the number of queries sent will be 128. With the TTL mechanism there is a limit to how many nodes that receives the request, which means that there is no guarantee that the requested file will be found

[3]Time-to-live. How many hops a message will take before it dies

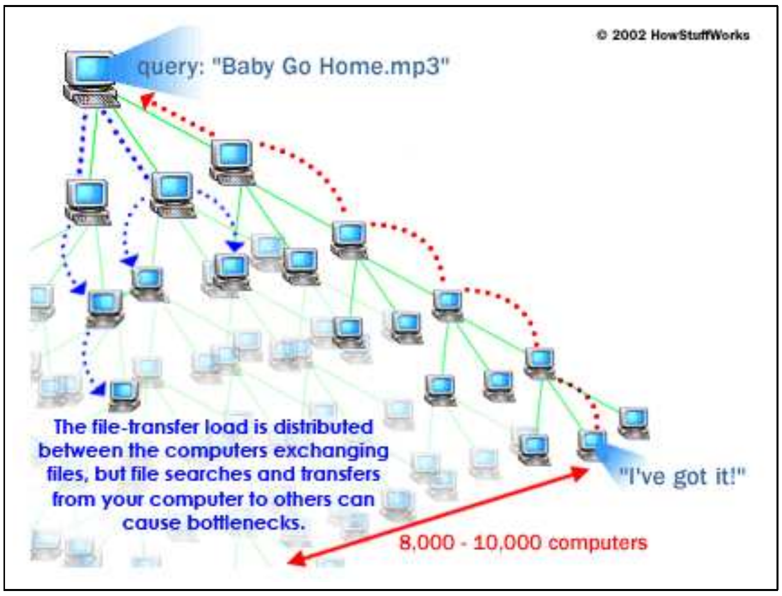

© 2002 HowStuffWorks
query: "Baby Go Home.mp3"
The file-transfer load is distributed between the computers exchanging files, but file searches and transfers from your computer to others can cause bottlenecks.
"I've got it!"
8,000 - 10,000 computers

Figure 2.2: An overview of a Gnutella network. Taken from[1].

even though someone in the network has it.

When a peer has the requested file, the answer message follows the same route as the request message. This is show by the red dotted line in the figure. When a peer receives a positive answer and wants to download the file the two peers connect to each other outside the Gnutella protocol.

The Gnutella protocol itself does not provide a fault tolerance mechanism. It relies on enough peers being connected so that a query will propagate far enough to find a result, but does not guarantee anything.

BitTorrent

In July 2001 Bram Cohen came out with the BitTorrent protocol. It is widely popular and several clients have been made supporting the protocol. A file is seen by BitTorrent as a number of identically-sized segments. To download a file with the a BitTorrent client you often need to know of a web page that offers the download of torrent files. These are meta-data files containing information about the file, like the checksum[4] of each segment and which tracker to connect to. A tracker is a central entity which helps coordinating communication between peers. The client receives a list from

[4]The checksum is created by using the SHA1 hashing algorithm

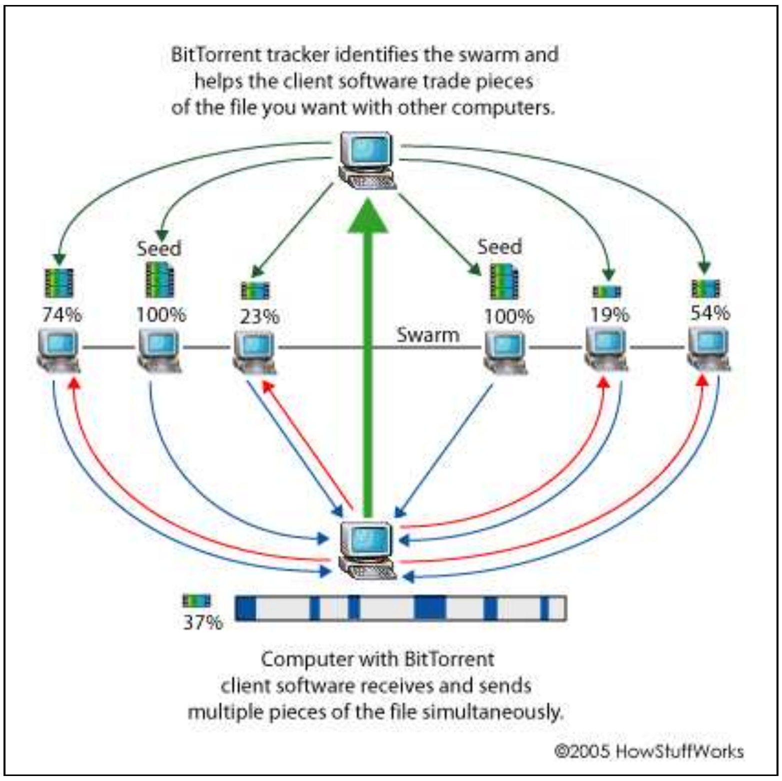

BitTorrent tracker identifies the swarm and helps the client software trade pieces of the file you want with other computers.
Seed
Seed
74%
100%
23%
Swarm
100%
19%
54%
37%
Computer with BitTorrent client software receives and sends multiple pieces of the file simultaneously.
©2005 HowStuffWorks

Figure 2.3: How a BitTorrent swarm works. The client receives segments from the other peers and sends information back to the tracker. Taken from[1].

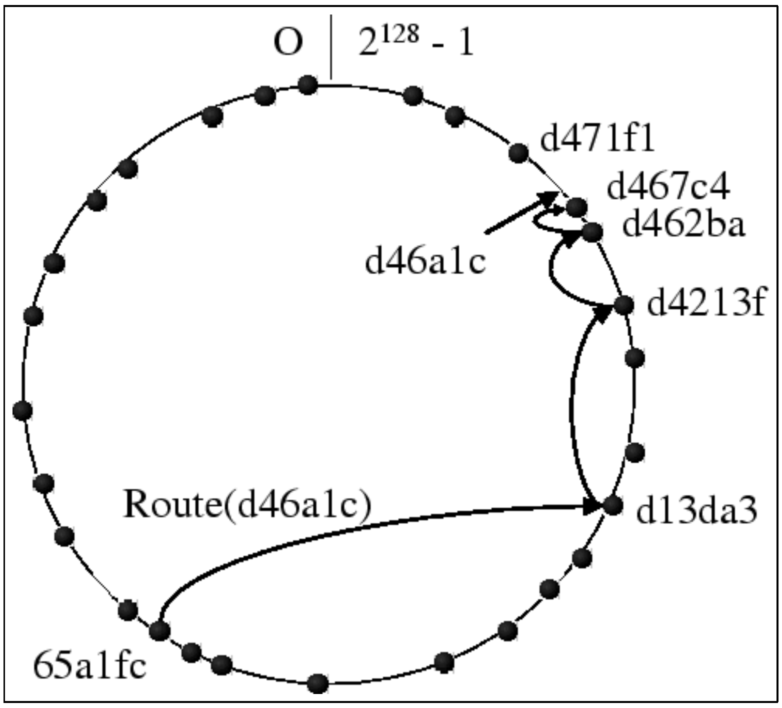


Figure 2.4: Routing a message from node $65a1fc$ with key $d46a1c$. The dots depict live nodes in Pastry's circular name space. From [2].

the tracker with other peers currently transferring pieces of the file specified in the torrent and connects to these. The file transfer starts and the client informs the tracker of the new segments it has received. The BitTorrent protocol tries to keep the availability of each segment as high as possible by asking for the least popular segments first. To cope with people only downloading and not sharing (hit and run) the protocol uses a tit-for-tat scheme, which means that peers prefers to send data to other peers it is receiving data from.

2.2 Pastry

Pastry[17] is a self-organizing routing overlay middleware which uses a *distributed hash table* for locating nodes and objects. It is responsible for routing requests from any clients to a host that holds the object which the request is addressed. The object of interest may be placed and reallocated to any node in the network without client interference, objects may also be duplicated on to other nodes to ensure availability. The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, using knowledge at each node to locate the destination object. All nodes are assigned a unique 128-bit GUID[5], usually a SHA1 hash, and forms a circular name space where GUID 0's lower neighbour is 2^{128}-1. In the highly unlikely event that two nodes get the same GUID Pastry detects it and takes proper action. The algorithm calculating node ids tries to keep the nodes in the overlay network evenly spread throughout the id space.

2.2.1 Routing in Pastry

Each node has a list of leaf nodes called leaf set, a list of neighbour nodes and a routing table[15].

The routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries each and each node maintains its own tree-structured table containing GUIDs and IP addresses for a set of nodes spread throughout the entire range of the 2^{128} possible GUID values, with increased density of coverage for GUIDs numerically close to its own. GUIDs are viewed as hexadecimal values and the table classifies GUIDs based on their hexadecimal prefix. The value of b is a trade-off between number of entries in the routing table and the maximum number of hops required to rout between any pair of nodes[17]. The neighbour list represents the M closest peers in terms of the routing metrics. The neighbour list is not used directly by the routing algorithm but it is used for maintaining locality principals in the routing table. The leaf set consist of the L closest peers by GUID. The leaf set ensures reliable message delivery and is used to store replicas of application objects.

Consider a message with id P addressed to a node with id D. First node D checks if the message id falls within the range of the node ids covered in its leaf set. If so, the message is routed directly to the destination node. If the id is not covered in the leaf set the routing table is used and the message is forwarded to a node that shares a common prefix with the id by at least one more digit. If no nodes in the routing table shares at least one more digit with the message id than the node itself the message is routed to a node who share the same amount of digits as the message and is numerically closer to the message id than the current node. With this approach the message will always get closer to the destination node D.

Figure 2.4 show the routing of a message with key $d46a1c$ from node

[5]In this thesis GUID, key and id all refer to the hash-value given by Pastry ranging from 0 to 2^{128}-1

$65a1fc$. With the use of the routing table the node sees that the node with the longest common prefix compared to the message key is $d13da3$. This lookup happens at each node receiving the message. In a network with N participating nodes the routing algorithm will correctly route a message addressed to any node in a maximum of $O(logN)$ steps.

2.2.2 Self-organizing

As mentioned Pastry is self-organized, this means if a node falls out without notice due to host failure or departure or a new node joins Pastry is able to connect the circle again. A node is considered failed if its immediate neighbours are unable to communicate with it. When this happens the leaf sets containing the failed node needs to be updated.

When a node with id X joins it initializes its state tables and inform other nodes of its presence. This is done by the new node sending a request for a join message to to a node it already knows of, lets assume this is a node with id A. The knowledge of such a node can be acquired through multicast or methods outside the system. Node X asks node A to route a join message with the id equal to X. All nodes on the path to X which the message visits send their state tables to X. The new node inspects the received information and builds its own state tables based on this.

After a node failure the leaf set L needs to be repaired. The node that discovered the failure looks for a node close to the failed node in L and requests a copy of that nodes leaf set. The new leaf set will contain a sequence of ids that partly overlaps those in L and the node discovering the failure will pick an appropriate node from the new leaf set to replace the failed node. Other neighbours are then informed of the failure and they perform a similar procedure.

Repairs to the routing table does not take place even though the leaf set had errors. The routing of a message will still work when the routing table is not up to date as long as an acceptable amount of entries are correct, failed routing attempts only results in the node trying a different entry from the same row in the routing table.

2.2.3 Locality metrics

Even though a node is numerically close to another node it is not certain that they are close to each other in physical distance. To address this issue Pastry tries to minimize the possibility for unnecessarily transport paths on

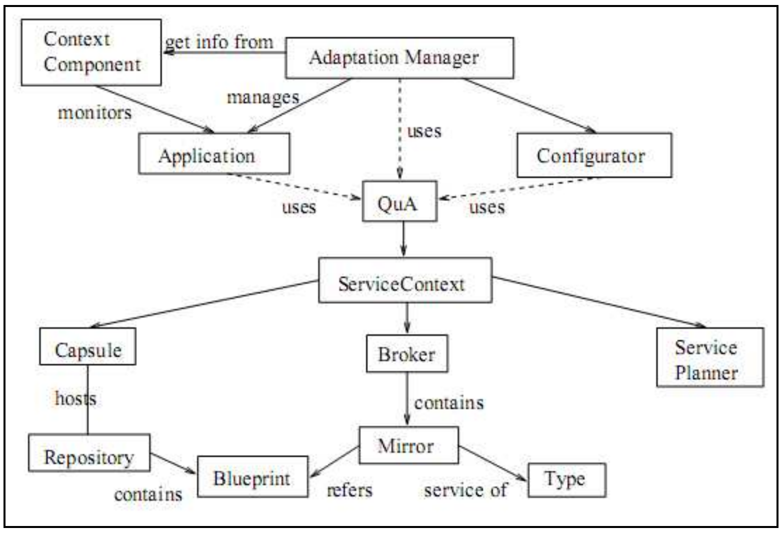


Figure 2.5: The QuA middleware. Taken from [3]

the underlying network by using a locality metric based on network distance (e.g number of IP hops or measured latency) to select appropriate neighbours.

2.3 QuA

Component middleware and adaptive middleware are both intended to increase software reuse, yet the established middleware architectures do not have support for quality-of-service (QoS) aware applications. Without this application developers are forced to program platform-specific knowledge into application components, causing the components themselves to be platform dependent[5]. QuA is a middleware component architecture developed at Simula Research Laboratory and aims at ensuring platform independence even for QoS aware components by building QoS support in the middleware architecture.

QuA is a self-adaptive middleware[6] meaning that the system can adapt itself to changing requirements and environments, providing dependability, robustness and availability with minimal human interaction.

QuA uses planning-based adaptation which means that applications are

[6]Middleware is a layer of software between the operating system layer and the application layer

specified by their behaviour, and planned, instantiated and maintained by the middleware so that the behavioral requirements are satisfied throughout the application life-time[18]. This middleware approach enables a clean separation between adaptation related concerns, for example deciding when and how to adapt. The adaption manager collects information from a pluggable context component that monitors the application. When the adaptation manager decides that change is needed. it calls the *Service Planner* for a (re-)planning of the service.

Service Planning

A central part in the approach of the QuA middleware is a process called *serviceplanning*, whose responsibility is to plan the initial configuration or the reconfiguration of a service[18]. It is based on searching through and evaluating a set of alternative service mirrors in order to find and select a service implementation that satisfies the users QoS demands. To do this the service planner queries a broker for the set of mirrors that fit the requirements. Next, the service planner tries to resolve any dependencies defined. A mirror is said to be fully resolved if it has an architecture with no dependencies or all dependencies resolved[18]. For each fully resolved mirror, the QoS predictors are calculated. Each resulting prediction is used as input to the utility function in the behaviour specification. The result of the of service planning is normally the mirror ranked highest by the planner.

An adaption manager decides when a service needs to be replanned, which means to go back to the planning phase and reconsider the architecture that was selected, or one or several of it resolved dependencies[18]. The replanning is based on the running service's current service mirror, if a service configuration that has a better utility in this context compared to the current one, the current service will adapt into the new service.

Service Mirror

A reflective system is one that performs computation about itself, and that provide inspection and control through its reflective interface[19]. QuA uses a variant of this called mirror-based reflection which is an improvement of traditional reflection. In mirror-based reflection, meta-level functionality is implemented separately from base-level objects using meta-objects called mirrors. So instead of querying an object, the reflective system is queried for the object's mirror, which contains the object's meta-data. Access to the meta-level

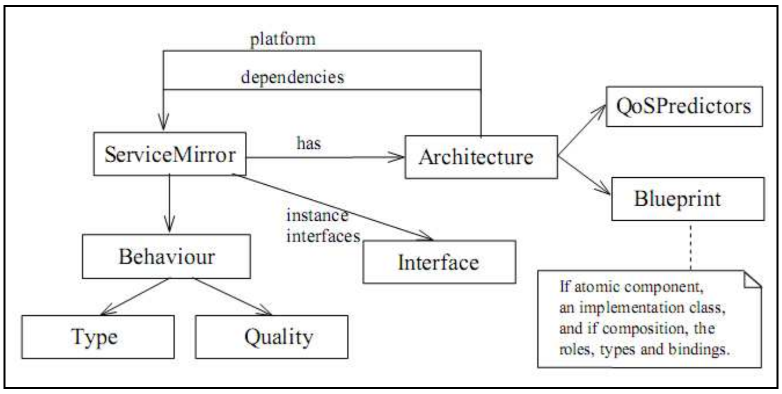


Figure 2.6: Service mirror. Taken from [3]

is defined as a middleware service, like traditional middleware services such as instantiation and binding. In QuA these mirrors are called *service mirror*. Figure 2.6 shows a conceptual view of a service mirror. The *Behaviour* part consists of *Type*, which reflects the type of service and defines the functional behaviour of the service in terms of its provided interface, and *Quality*, which are used as constraints on the type defining a set of QoS dimensions.

Chapter 3

P2P Broker

3.1 P2P broker

The P2P broker is implemented by using a Pastry implementation in java called FreePastry[20] version 1.4.4. The FreePastry project is a cooperation between Rice University of Houston, U.S.A. and Max Plank Institue for Software Systems, Saarbrücken,Germany. Distribution of FreePastry is under a BSD-like licence and the source code as well as a pre-compiled version is freely downloadable. Pastry acts as a structured overlay network and uses distributed hash table (DHT) for locating nodes and objects, which means that it is possible to access any node within a maximum of *O(log N)* hops, as described in section 2.2.1.

The basic concept of the broker, which is shown in figure 3.2, is to build an overlay network between all the nodes configured to use the P2P broker as an implementation broker in QuA by using peer-to-peer technology to share the load of hosting service mirrors. The P2P broker consist of 3 layers[4].

QuA layer This part contains the logic needed to communicate with other parts of QuA. When invocation from QuA is received this layer is responsible for creating a message indicating the type of invocation and labelling it with the id-base[1] indicating responsibility for the resource type. The id-base is created by a pluggable id-base generator.

This layer is also responsible for storing service mirrors that belongs to the node, and for finding relevant service mirrors as response to a request from another node. Replicated service mirrors are also stored in this layer, which gives a quicker recovery when a node goes down

[1]The string which the id is generated from

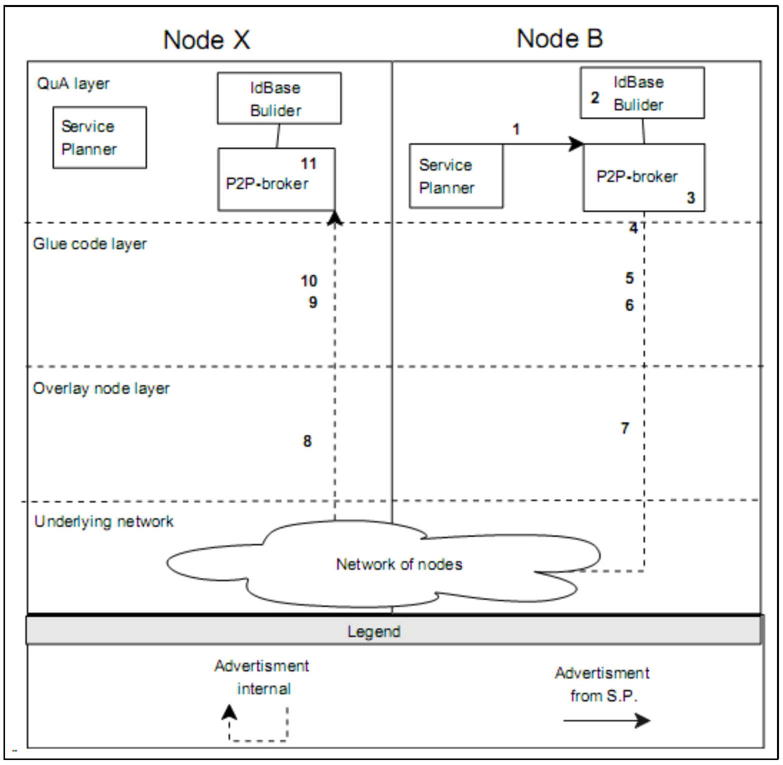


Figure 3.1: Layer responsibility when a service mirror is advertised. Taken from [4].

and the neighbour node that assumes responsibility already have the service mirror in the QuA layer.

Glue layer The glue layer binds the QuA layer onto the peer-to-peer layer. The responsibility of this layer is to hide the QuA layer from the underlying layer making it easier to change the underlying peer-to-peer technology. This layer receives messages from the above layer, creates and id based on the id-base from the above layer and creates a message wrapper that can be handled from the underlying layer. When messages arrives from the underlying layer, this layer unwraps the message and sends it to the QuA layer. Calculaton of when to update replicated resources is also handled at this layer.

Peer-to-peer layer This layer is responsible for keeping the network organized and for routing messages to the right node. This features are supplied by a third party, in this case by FreePastry.

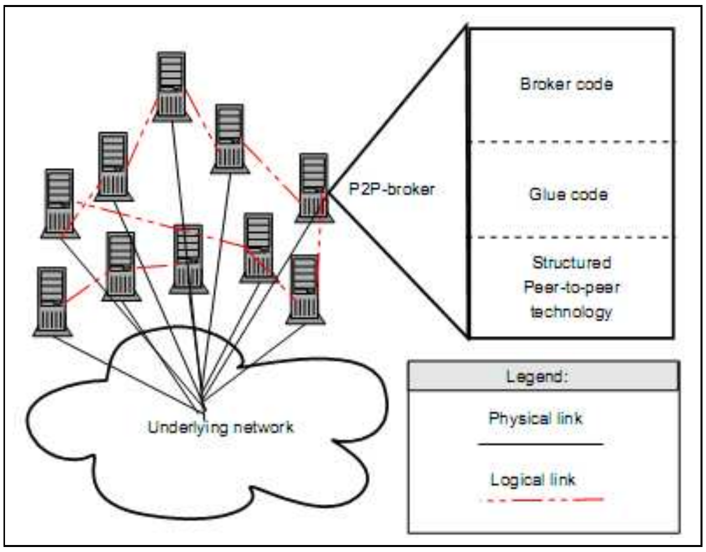


Figure 3.2: Basic concept of the P2P broker. Taken from [4].

3.2 Service mirrors

In QuA each service is represented by a service mirror, which is an object reflecting the service behavior (known as service type) and its implementation (known as blueprint). Each service mirror has a map of <name,value> property pairs, where the list of property types allowed in the map is determined by the QuA type specified by the service mirror. The property type determines the value range of a property of that type and the matching operators that can be applied for filtering mirrors.

3.3 Distributing resources

As described in section 2.2 Pastry is used to create a circular overlay network with strict rules about where the nodes are placed in the id space that range from 0 to 2^{128}-1. Section 2.2 also shows how the routing of messages from one node to another works. Each message is assign an id from the same id space and the message ends up at the node with the id closest to the message id. With this knowledge we can use it to divide responsibility of resources between participating nodes.

In Oudenstad's master thesis[4] it was presented two ways of creating a service mirrors id-base, referred to as key-base. The first one called *basic id-*

base builder and the other one *property id-base builder*. The first one creates a message id based only on the service type. This leads to all mirrors that have the same service type get the same id, even if they have completely different property values, and thus will end up on the same node. This is what we are trying to avoid, and this is what the second implementation is for. The proposed approach is to duplicate a service mirror so that it will be distributed onto more nodes. As we know messages are associated with the node that has the id which is numerically closest to the message's id in the id space. The id is generated by a string, and the same string always gives the same id. So the id generated for the service mirrors has to be different for each duplicate which again means that a unique string has to be produced for each duplicate. By taking advantage of the service type and property values of a service mirror we can create a number of strings that can be used to create ids equal to the number of unique unordered combinations of service type and property values. Each string will be created on the form

$$\texttt{id-base} = T + x_0 + \cdots + x_i + \cdots + x_n$$

where T is the type of service, x_i is an ordered list of the associated properties used as constraints and the operator $+$ indicates string concatenation. The reason for the second approach is that by giving up storage space we hope to gain faster querying time. By doing this we also gain another advantage. With the *property id-base builder* we introduce matching on properties as well. With the *basic id-base builde* only the service type is compared to see if a request match any of the node's service mirrors, leading to all mirrors matching the type being sent to the quering node even though only a few of the service mirrors are possible candidates. With the new method we can save bandwidth by doing a better filtering earlier in the request process by also removing service mirrors which do not match with desired properties as well as type from the result set sent back to the quering node.

As an example consider a node that offers a printer service with the QuA type *printer* and with two properties, *papersize* and *price* where *papersize* is the paper size which the printer uses, for example A4, and *price* being the price per page it costs to use the printer, we set this to 2. With the *property id-base builder*, based on the assumption that we are interested in all combinations of type with and without properties. This gives us four possible strings to build ids on as shown in table 3.1. It is not guaranteed that each version of the service mirror ends up on different nodes, one node could be in control of the id space that all the duplicates ids lies within. But the chance of this not happening increases with the amount of nodes participating.

1	printer
2	printer,A4
3	printer,2
4	printer,A4,2

Table 3.1: All possible combinations of a service mirror containing two properties

The reader may already have noticed that if the value of both properties are the same, mirror two and three in the table will get the same id. As described in section 3.6 the consequence of this is that only three mirrors will be created because the algorithm creates all possible unsorted combinations leading string two and three to be seen as duplicates and one of them is removed, this again results in one node answering for two query possibilities. But besides that it will not lead to any problems.

3.4 Replication

To show resilience to node and network failure with respect to loss of data a replication mechanism[4] is needed. The solution takes advantage of one of the key features of a structured peer-to-peer system, the placement of the nodes in the global id space relative to each other is highly organized. Each node always knows who its closest neighbours are. Also the strict algorithm for routing messages, always routing to the node responsible for the id space which the message id lies within. The combination of this two properties gives the possibility of using the closest neighbours to any given node as the replicator nodes. If node X unexpectedly leaves the network one of its immediate neighbours, node Y, will be in control of the id space node X controlled. This leads to node Y receiving all messages node X would have received if alive. Node Y is now responsible for the the service mirrors it already has as well as the ones from node X.

If both node X and node Y is alive and a node Z joins and get in between node X and node Y in the id space, the responsibility of the resources at node X and node Y is recalculated and node Z receives any resources which lies within the id space it is responsible for.

3.5 Queries

A query in the P2P broker is build as a service mirror with type and properties. Consider the example in section 3.3. If we are only interested in printers where the paper size used is A4 and do not care how much the cost is the service mirror used as a query will contain the type printer, which is mandatory, and the <key,value> pair <papersize,A4>.

We know that if we specify the same type and properties in your query-mirror as an already existing mirror the id of the two match and the query is sent to the node holding the desired mirror. The node that receives the request search through all mirrors to find any match. Both type and properties are matched and only service mirrors matching all the desired criterias are returned.

Sometimes when requesting a resource it is not found, this can of course be because the resource is lost. This happens when a node holding the resource and all the replicator nodes dies before the responsibility for the resource can be recalculated. This is unfortunate, but there is nothing that can be done when the resource is already gone. The retry mechanism will not be of any help here, but there is another reason to why a request result yields zero resources. This is when a node leaves unexpectedly but at least one replicator node is still alive and the responsibility have to be recalculated. This can take some time and if a request arrives before anyone have taken the responsibility for the resource it will not be found. This is why a retry mechanism has been implemented. It retries four times with 15 seconds interval (both parameters are configurable).

3.6 Partitioning the searchable domain

As mentioned in Oudenstad's thesis[4] the *id-base builder* was not extended to support the QuA property model because of limited time. To be able to do the tests an automatic way of generating property combinations had to be made. That led to a working sample of the `buildAllIdBases` method inside the `PropertyIdBaseBuilder` class. What the method does is to take a list of property values and create all possible unordered combinations of properties. The task is solved by using recursion[2] and with no optimization this will rapidly become a sever problem. Note that it is not a part of this thesis to use an optimized approach to solve this, but at least one easy change can be done to make the number of function calls go down drastically. One thing

[2]Recursion in terms of programming is a procedure that may call itself as a subroutine.

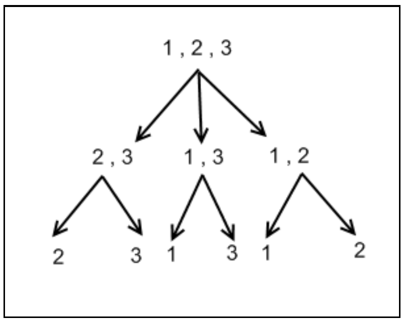


Figure 3.3: All possible unordered combinations with the numbers 1, 2 and 3 with the use of a non-optimized recursive method.

we know is that the last row in figure 3.3 where all values are separated will always occur as long as at least on property is defined. With this knowledge we can do this operation outside the recursive method and by this we will remove the most expensive row in the tree. By expensive we mean the amount of iterations through the recursive method. An evaluation of the property id-base builder can be found in section 5.1.

Chapter 4

Planning the experiments

In this chapter we take a look at the planning of our experiments. Some experiments were already conducted by Oudenstad and documented in his thesis[4]. These were related to measuring the scalability, robustness and self-organizing abilities and the P2P Broker.

The rest of this chapter gives a description of why and how we want to conduct each test together with a list of the parameters involved.

4.1 Scenario

To demonstrate the self-adaptive behavior of QuA, an application called Personal Media Server (PMS)[18] has been created by the developers of QuA. The application can be viewed as an in-house personal proxy server for storage and delivery of multimedia content. The PMS receives media data from external providers, for instance, a regular TV-broadcast via cable or an audio stream from an Internet radio provider. A client device, such as a laptop or handheld computer, may be used to connect to the PMS and initiate streaming from any location, assuming Internet connectivity.

If we see the PMS application in combination with the P2P broker, new possibilities opens up. With no need for a central server each participating node can be a consumer and distributor of the media content. From this we can describe at least two scenarios.

1. As we get more and more media devices at home with the ability to interact with its surroundings(TV, PDA, laptop, video game consoles) a likely scenario would be to incorporate an adaptive middleware such as QuA and a media application like PMS on each device. Since the home is not a mobile environment a central server can be used in stead. But

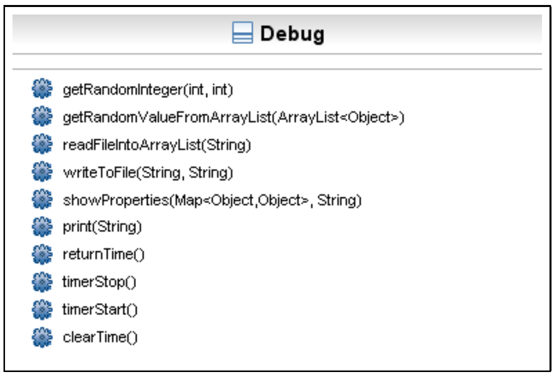
then we get all the problems which a peer-to-peer system can avoid, especially the single point of failure. Also with a central server everything would have to be placed on the media server before other devices can stream it. With a peer-to-peer approach everyone can in theory stream from everyone at any time assuming they are all connected to the same overlay network.

2. Consider a group of people at a technological conference attending the presentation of a long awaited product. Everyone is excited and eagerly records the event. After the presentation one of the attendants is a little disappointed with the quality of the video captured because he was too far away from the stage. So after the presentation he picks up his cell phone and logs on to the application. Here he finds media content from the presentation he attended, and by browsing through it he finds a peer which was much closer to the stage. He choses this peer and starts receiving video content.

4.1.1 Conclusions from the scenario

These two scenarios have a lot in common, but the main differences are that with the first one we can assume that each node stays for a longer period of time. With an overlay network where each peer is more likely to be an individual person peers may come and go more often without any warning. The other thing to point out is the amount of peers we are likely to see in each scenario. With the first one we can assume a small amount of participants limited to the most common devices in a household, like cell phones/PDA, gaming consoles, stationary and portable computers and TV. The peer-to-peer technology used promise scalability, robustness and self-organizing, but how well will this work with only a few nodes? Is it overkill to use an overlay network capable of hosting $2^1$28 different peers when only a few peers can be expected?

Everything we are testing with the second scenario in mind can also be applied to the first scenario, except for the amount of peers. So we will keep our main focus on the second scenario. We will look at the distribution of resources and see how they are placed in the overlay network, distribution time, quering time and availability of resources when nodes are leaving unexpectedly.


Debug

getRandomInteger(int, int)
getRandomValueFromArrayList(ArrayList<Object>)
readFileIntoArrayList(String)
writeToFile(String, String)
showProperties(Map<Object,Object>, String)
print(String)
returnTime()
timerStop()
timerStart()
clearTime()

Figure 4.1: The debug class. Shows all useful methods

4.2 Limitations

Outdated information is not taken care of, when a node suddenly dies there is no way to remove the outdated service mirrors. This will potentially cause the search for a service mirror to take longer time because a node has to go through outdated mirrors as well as active mirrors. But for these tests it does not matter if we find outdated resources since we do not care where the resource comes from.

4.3 Test helpers

To help us run the tests we need a way to distribute the service mirrors, query for them, log results and debug code. The distribution and query part is done by having one dedicated node for each of the two tasks. Logging and debugging is done with the help of a debugging class.

Debugging

For testing and debugging a class was created containing several functions making the tasks easier. The class consists of a few helper functions as can be seen in figure 4.1.

print This is a small but very useful method. All it does is print to console what ever you send to it. The great thing about it is that the output can be uniform. For example putting a time stamp in front of the output or writing out which thread is using the method. Also to turn off all output just comment out the line writing to console.

writeToFile This is just your typical write-to-file function. It takes a filename and a string and appends the string to the end of the file. Useful for logging and makes it easier to create fast logging to file with only the call of a function.

showProperties This function takes a `java.util.Map` and prints its key-value pair to console. It is intended to use for printing out the service mirror properties.

readFileIntoArrayList Takes each line in a file and puts it into a list. This is used in combination with `getRandomValueFromArrayList`.

getRandomValueFromArrayList Takes a list and returns the value of a random key.

getRandomInteger This method was added because of the need to set a lower limit to the random integer. Returns a random integer between the lower and higher limit.

timerStart, timerStop, returnTime, clearTime These methods are used for time measurement. By taking the end time and subtract the start time the elapsed time between two points in the code can be measured. `returnTime` returns the result in milliseconds after `timerStart` and `timerStop` are called. Note that this way of code measuring is only useful if the time measured is considerably larger than the time it takes to call the timer methods.

Distributing resources

The distribtion of service mirrors is done with a dedicated node. Since we do not care where the resources come from, only where they end up, we can have one node take care of this task. At start-up the node takes three parameters from the user. These are number of service mirrors to be sent out, how many to be sent each time and the delay in milliseconds between each time. This enables the possibility of starting more distribution nodes with different user input for better random simulation. Note that with one service mirror with n properties and $n>0$, $\left(\sum_{k=1}^{k=n} \frac{n!}{(n-k)! * k!}\right) + 1$ service mirrors are generated and distributed (see section 5.1). All service mirrors that are sent out from the distribution node are logged for later use with the query node.

Quering for service mirrors

For queries a query node is used. This node takes two parameters from the user, the upper and lower random query time boundary in seconds. To pick service mirrors to ask for the log file created by the distribution node is used. This is done by picking a random line from the file, prosess it, build a service mirror containing the type and properties to ask for and send the message off as a query. The log file is red before each query so the query node can start quering while the distribution node is sending out service mirrors. Also, the quering have to be manually started by the user so that the query node can be started without asking for service mirrors immediately. This is useful when all nodes have to connect to the overlay network before any service mirrors are sent out.

4.4 Setup

All tests were done on a single computer. The operation system used was GNU/Linux distribution Red Hat Enterprise Linux WS release 4 (Nahant Update 6) and the Java SE Runtime Environment version used was 1.6.0_06. It ran on a Intel Core2 2,4 GHz with 3GB RAM. Running all the nodes on a single computer uses a lot of socket handlers, approximately 30 per node when a lot of nodes are connected. The defualt number of open sockets allowed was 1024, which is not enough when simulating over 50 nodes. To raise this value, the unix command `ulimit -n <num allowed sockets>` was used.

For a better control of the test both resource discovery and resource distribution is done from dedicated nodes. This way we are able to inject specialized node in to an existing overlay ring and control their behaviour more easily than by having each participating node distribute and search for resources. If we were testing this with a real application it would of course be each node's responsibility to distribute and search for resources, but in our simulation we do not care where the resources come from because it has no impact on which node the resource ends up at. When searching for resources we are interested in the time used from the query is sent to the answer arives and aspects affecting this time. In a real network with network latency the search time can be affected by which node is sending the request due to the distance between the nodes in the underlying network. In our tests we use a local network and can assume no network latency which means that the placement of the nodes in the underlying network compared to each other will not affect the measurements.

The tests will look at the usage of the new distribution method described in section 3.3 compared to the existing method mentioned in the same section. This main task can be divided in to smaller parts.

1. We need to know the time spent creating all the duplicate resources. Even though section 3.6 states that the solution created is not fully optimized it will give us a hint on how this part of the process performs. This will probably be the biggest bottleneck when distributing resources.

2. When asking for resources it is of interest to know how long it takes from a request is sent until an answer is received. This process can be measured at the quering node to see the round-trip time and at the node receiving the request to see how long the search for relevant resources takes.

3. By testing with both the *basic id-base builder* and the *property id-base builder*, described in section 3.3, we can test to see if we gain any faster quering time with the property id-base builder.

4.5 Tests

To see if the property id-base builder gives a better quering time we need several nodes concurrently asking for resources. This will simulate different peers asking for service mirrors to do a service planning. The same test will be done with the basic id-base builder to see how long it takes when several peers are asking for the same resource. From this we hope to see if the property id-base builder gives any better quering time.

Chapter 5

Results

5.1 Evaluating the property id-base generator

As mentioned, making a fully optimized way of creating all property id-bases is not a part of this thesis. But the current approach is a starting point and it is interesting to see how it performs. The method creating all the combinations is only triggered when there are two or more properties specified in a service mirror. If zero properties are given the id-base will of course only contain the service type, and with one property two id-bases will be made, one with only the service type and one with both service type and property value. More of this can be found in section 3.6.

We have tested the id-base generator with service mirrors containing from two to ten properties. The property combinations are found by using recursion and the time spent calculating all combinations has been recorded by placing timers right before and after the call to the recursive method. Table 5.2 shows some facts about the method. By using knowledge from the mathematic branch combinatorics we can set up some mathematical facts about the recursive method which will help us in the evaluation process. The number of iterations through the method can be found by using

$$\left(\sum_{r=2}^{r=n} \frac{n!}{r!}\right)$$

where n is number of properties, $n>1$ and r being the range from 2 to n. With three properties with values (1,2,3) all possible combinations are shown in figure 3.3. The table row Combinations is calculated by using the formula

$$\left(\sum_{k=2}^{k=n} \frac{n!}{(n-k)! * k!}\right)$$

where n is number of properties, $n>1$ and k being the range from 2 to n. This gives us all possible unordered combinations without duplicates. Note that this is the number of unique combinations generated by the recursive method. To get all combinations we need to add the number of combinations we calculate outside the recursive method and the combination only containing type. By making some small alterations to the formula above we get $\left(\sum_{k=1}^{k=n} \frac{n!}{(n-k)!*k!}\right)+1$, which gives us all possible combinations.

It is clear that this approach is far from optimal, and for seven properties and more the time significantly increases. If we imagine this used on a small handheld device with minimal computational power this will be very costly in terms of cpu power and memory consumption. The default maximum allowed memory with the java virtual machine was set to 284MB which resulted in an out of memory exception. By increasing the maximum allowed memory to 512MB we were able to use ten properties, and the peak of resource consumption was for cpu usage 47% and memory 12% of the computers total. We can also see the amount of service mirror duplicates when we have six properties or more becomes very large, this will potentially take a lot of extra bandwidth to distribute.

Improvements

The most obvious improvement is the method generating the id-base strings. But the recursive method can be sufficient if we instead set a limit to how many duplicates that can be generated. But this makes the process quering for mirrors more complex since we need to know which property combinations that were used to create duplicate mirrors and which were not. Another solution would be to limit the maximum allowed properties than can be specified for a service mirror, but this puts a constraint to the usage.

5.2 Evaluating the use of the property id-base approach

The idea behind the property id-base approach is that by using more storage space we hope to get a better quering time. Section 3.3 gives detailed information about this. For this to have any effect one condition have to be met, the placement of service mirrors in the overlay network have to be fairly even distributed so that the query load also can be evenly distributed. We see from figure 5.1 that the placement of the nodes in the id space is evenly distributed, as promised by Pastry[17]. In figure 5.2 we see the distribution of 100 service mirrors with three properties. All property values and service types are different making it a total of 800 service mirrors. As we can see only four nodes are without any service mirrors and some nodes are holding more than the others. This is no surprise since we have 100 nodes and 800 service mirrors but most nodes have between 0 and 10 service mirrors. This shows that the distribution of resources is also evenly distributed, with of course some abnormalities. But if we look at the second scenario in section 4.1 we will not have that many different service types. We can imagine 100 user which each have his/her own device with one out of four possible applications. The four application have their own service type and each service type have two properties each with a random value between 1 to 5. Then we get a distribution as shown in figure 5.3. Again we have sent out 100 service mirrors simulating one resource per node. Since we have 2 properties we get 4 duplicates per service mirror giving us a total of 400 mirrors. From this we got 99 unique ones. This number will probably vary for each time the test is executed since the property values are random. The four dots to the right in the figure are the nodes holding all the service mirrors with the id-base only made by service type. If the basic id-base builder was used these had been the only dots on the graph.

We see that the resources are more or less evenly distributed in the id space, but do we gain any better query time by doing this? Table 5.1 shows the time from a query is sent until the result set is received. As we can see when many nodes are requesting resources at the same time the time before the result is received increases, but not drastically. The first row in the table can be a user asking for a resource from a node which holds only one mirror and that no one else is quering. The second row could be a node with six resources and with nine other nodes quering it.

The amount of nodes asking for a resource and the amount of resources a node is holding does not seem to give that big of an impact on the query time. Another thing to consider when using the property id-base builder is the extra bandwidth and storage space used. This is tested by serializing the message object sent from one node to another and by serializing a service mirror. Every message containing a service mirror takes 2.11KB and a service mirror by itself takes 1.51KB. If we see the overlay network as one distributed unit this gives, for a service mirror with 4 properties, a bandwidth overhead of 158.25KB when using the property id-base builder. This includes the bandwidth used by the replication process. The extra disk space used is 113.25KB. Lets imagine a network with 100 nodes, each distributing at least

Avg. resources	**Avg. time**	**Query nodes**
1	177	1
6	210	9
25	137	1
100	382	9

Table 5.1: Shows the average of number of returned service mirrors, the average time in milliseconds for the query node to get answer and how many query nodes that were concurrently asking for resources.

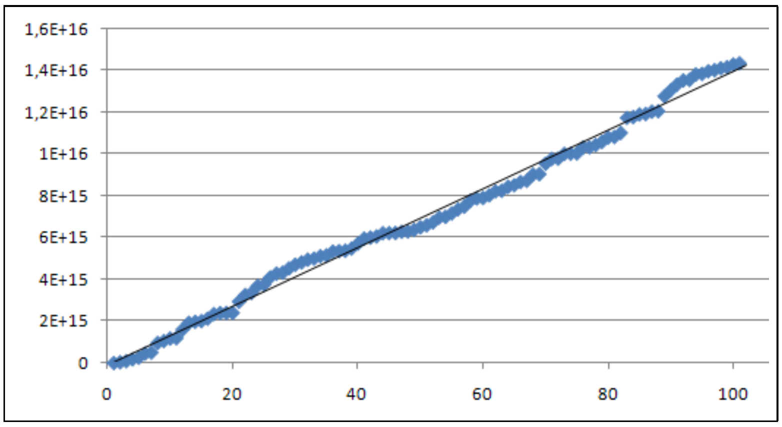


Figure 5.1: 100 nodes spread out in the id space. Y-axes shows the node id in 10 base. X-axes is the number of nodes.

one service mirror each and about 10 applications concurrently replanning a service. In a worst case scenario we get a bandwidth overhead of 15.5MB and disk usage overhead of 11MB.

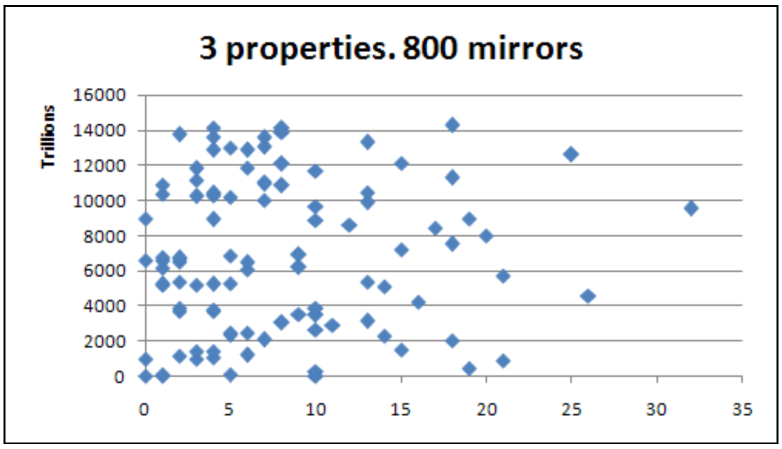


Figure 5.2: 800 different mirrors spread out over 100 nodes using the property id-base model. Y-axes shows the node id in 10 base. X-axes is the number of service mirrors a node is holding.

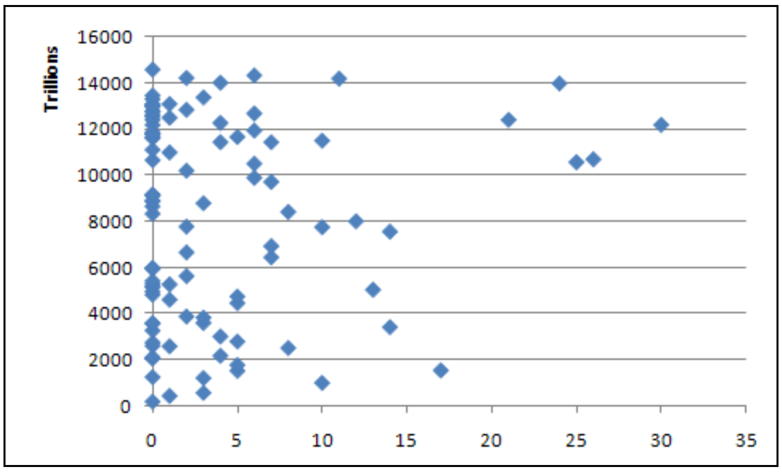


Figure 5.3: 400 mirrors spread out over 100 nodes using the property id-base model with 4 service types and 2 properties. Y-axes shows the node id in 10 base. X-axes is the number of service mirrors a node is holding.

Properties	**Combinations**	**Function calls**	**Time**
2	1	1	0
3	4	4	0
4	11	17	1
5	26	86	2
6	63	517	9
7	120	3620	49
8	247	28961	160
9	503	260650	591
10	1013	2606501	7404

Table 5.2: Number of properties, number of unique combinations, iterations through the recursive method and time in milliseconds using the recursive method for generating all possible property combinations.

Chapter 6

Conclusion and further work

6.1 Conclusion

It is not easy to give a conclusion based on the environment the experiments were conducted. Both network latency and the specifications of the device running the QuA middleware with the P2P Broker will potentially have an impact on the results, unfortunately we were not able to test this.

Experience obtained by testing the P2P Broker shows that the nodes are evenly distributed even when only 100 nodes are connected. Also the distribution of resources among the nodes is also fairly evenly distributed.

By implementing the property id-base builder we have shown that the suggested approach on how to distribute the query load is in fact feasible, and that the amount of resources created is $\left(\sum_{k=1}^{k=n} \frac{n!}{(n-k)! * k!}\right) + 1$, where n is number of service type constraints and k being the range from 1 to n. We only gain a couple of hundrer milliseconds in query time at the cost of a lot of storage space and bandwidth. But this does not mean that the property id-base approach is not usefull in some contexts. In an overlay network with a lot of replanning and a lot of peers it might be an advantage to have a large searchable domain to relieve the load on a peer caused by queries. But in a network with few participants the time gained is minimal compared to the extra storage space and bandwidth used. It comes down to how much disk space is available on each peer, how much storage space a peer is willing to use and how much bandwidth utilization that can be tolerated. Related to our scenarios, which both have a small amount of nodes, using the property id-base builder will result in a lot of used storage space and minimal time saved when quering.

6.2 Further work

Even though the most important tests have been conducted there are more interesting aspects about the P2P Broker that can be tested. But some things we did not get time to do and some things are not a part of this thesis' scope but can be of interest.

- We did not get time to create a fitting way of testing with nodes concurrently joining and departing in the overlay network.
- Testing with the use of a real network, like Planet Lab, to see how network latency affects the resource discovery and distribution.
- Testing QuA with the P2P Broker on actual or simulated handheld devices to see how it performs with limited amount of computational resources.

Appendix A

Enclosed CD

On the last page of this thesis a CD is appended where the contents include:

- A README file.
- The source code for a QuA build including the P2P broker used in the experiments.
- The raw data from the experiments.

Bibliography

[1] Carmen Carmack. How bittorrent works, 2005. [Online; accessed 15-May-2008].

[2] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks, 2002.

[3] Frank Eliassen, Eli Gjørven, Viktor S. Wold Eide, and Jørgen Andreas Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In *ARM '06: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM '06)*, page 1, New York, NY, USA, 2006. ACM.

[4] Johannes Oudenstad. The design and evaluation of a qua implementation broker based on peer-to-peer technology. Master's thesis, University of Oslo, Department of Informatics, 2007.

[5] 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.

[6] Connie U. Smith. Increasing information systems productivity by software performance engineering. In *Int. CMG Conference*, pages 5–14, 1981.

[7] Wikipedia. Performance engineering — wikipedia, the free encyclopedia, 2008. [Online; accessed 28-April-2008].

[8] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Douglas P. Konkin, Gregory M. Oster, and Richard B. Bunt. Exploiting software interfaces for performance measurement. In *WOSP '98: Proceedings of the 1st international workshop on Software and performance*, pages 208–218, New York, NY, USA, 1998. ACM.

[10] Rob Pooley. Software engineering and performance: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 189–199, New York, NY, USA, 2000. ACM.

[11] Daniel A. Menascé. Software, performance, or engineering? In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 239–242, New York, NY, USA, 2002. ACM.

[12] Robert F. Dugan. Performance lies my professor told me: the case for teaching software performance engineering to undergraduates. *SIGSOFT Softw. Eng. Notes*, 29(1):37–48, 2004.

[13] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. *Technical Report HPL-2002-57*, 2002.

[14] 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.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: Concepts and design*, chapter 10, pages 398–430. Addison-Wesley, 4th edition, 2005.

[16] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network, 2001.

[17] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[18] Eli Gjørven, Frank Eliassen, Ketil Lund, Viktor S. Wold Eide, and Richard Staehli. Self-adaptive systems: A middleware managed approach. In *Self-Managed Networks, Systems, and Services*, pages 15–27. Springer Berlin / Heidelberg, 2006.

[19] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[20] FreePastry. Pastry - a scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications, 2008. [Online; accessed 27-Mai-2008].