# Brokering Planning Metadata in a P2P Environment

Johannes Oudenstad[1], Romain Rouvoy[2], Frank Eliassen[2,3], and Eli Gjørven[3]

[1] Norwegian Defence Research Establishment
P.O.Box 25, N-2027 Kjeller
johannes.oudenstad@ffi.no
[2] University of Oslo, Department of Informatics
P.O.Box 1080 Blindern, N-0314 Oslo
rouvoy@ifi.uio.no, frank@ifi.uio.no
[3] Simula Research Laboratory
P.O.Box 134, N-1325 Lysaker
eligj@simula.no

**Abstract.** In self-adaptive systems, metadata about resources in the system (*e.g.*, services, nodes) must be dynamically published, updated, and discarded. Current adaptive middleware approaches use statically configured, centralized repositories for storing and retrieving of such metadata. In *peer-to-peer* (P2P) environments, we can not assume the existence of server nodes that are always available for hosting such centralized services. However, the metadata repository is the keystone of the adaptation middleware and the consistency of adaptations relies on its reliability.
To address this limitation in our QuA planning-based adaptation middleware, we introduce a P2P broker, which is a metadata advertisement service based on P2P technology. This P2P broker can be plugged into the QuA middleware to support the construction of self-adaptive applications in a P2P environment. We use a structured P2P protocol that distributes the service metadata over a set of nodes based on service type and property information. The P2P broker is therefore capable of handling node failures by providing replication of the metadata. We present a working prototype of the P2P broker as well as results from initial experiments. These results show that the metadata distributes well over the nodes in the network, thus enabling scalability and robustness to node failures.

**Key words:** Peer-to-peer systems, resource brokering, self-adaptive middleware, service planning.

## 1 Introduction

As computing systems become larger and more complex, the idea of self-adapting systems is spreading to many areas of computing and communication, such as multimedia applications, mobile applications, advanced communication protocols, and management of low level operating system resources. In particular, a self-adapting system is able to reason about itself at run-time and, when necessary, to make changes to itself in order to better satisfy the current environment requirements. However, while distributed applications traditionally were built from client-server architectures, many current distributed applications are now built from *peer-to-peer* (P2P) architectures. These

systems consist of equal, autonomous peers entering an application when suitable for themselves, and then generally leaving without warning. Generally, approaches to adaptation middleware assume the existence of server nodes that can be expected to almost always be available, and in particular to be continuously available for long periods of time. As these approaches do not fit the P2P paradigm, we investigate self-adaptive middleware that is able to exploit other types of architectures at the middleware level. Thus, this paper contributes to the integration of component- and planning-based adaptation middleware in a P2P environment.

In this paper, we focus on the problem of dynamically locating metadata about resources, such as services, nodes, in the system as they become available in the network. We present the design of a resource broker based on a P2P infrastructure, and describe how this broker is used by our planning-based adaptation middleware QuA [1]. In the QuA middleware, adaptation is driven by metadata associated to services (*e.g.*, service performance or cost). Thus, the middleware aims at providing the best possible *Quality of Service* (QoS) to users under variable execution contexts. However, this adaptation requires quick and simple means to query for metadata in the system. And, existing approaches to resource discovery, such as Twine [2], do not satisfy our requirements as they do not provide the strong association of types to resources, as required in the QuA middleware. Therefore, we propose to achieve this task with our P2P broker, which exploits diverse connectivity between participants in a network and the cumulative bandwidth of network participants. Using a P2P infrastructure, we benefit from the seamless distribution of the metadata across the network nodes to improve the planning processing performance. Thus, a main challenge for the design of the P2P broker is to find a mapping from the metadata associated to services with the goal of obtaining an even distribution over P2P nodes. Besides, the replication of metadata using the P2P network ensures the metadata high-availability in terms of access time, fault tolerance, and network participants connectivity.

In the remainder of the paper, we introduce the QuA planning middleware (cf. Section 2), and we discuss related work in the domain of resource brokering (cf. Section 3). Then, we present the design of our P2P broker for the QuA planning middleware (cf. Section 4), and an evaluation of its performances (cf. Section 5). Finally, we discuss the perspectives of this work before concluding (cf. Section 6).

## 2 Foundations of the QuA Planning Middleware

The QuA middleware [1] supports planning-based adaptation, which means that applications are specified by their behavior, and are planned, instantiated, and maintained by the middleware in such a way that the behavioral requirements are satisfied throughout the application life-time.

Central to this middleware is *mirror-based service reflection* [3], which supports introspection and intercession on a service through all the phases of its life-cycle, including pre-runtime. 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 service mirrors. Thus, the task of *service planning* consists in planning the initial configuration or the dynamic reconfiguration of a service. The planner is responsible for evaluating alternative service mirrors in order to find and select the service implementation with the highest utility that satisfies both the functional and qualitative specifications of a service request. Service mirrors can be advertised to and obtained from a pluggable middleware *broker service*. The QuA broker is a trader-based discovery service. The resources traded in the broker are the service mirrors discussed above. Component and application developers alike must advertise the service mirrors to the broker. The broker has a responsibility of hosting all the service mirrors advertised in a repository. In the service planning phase, the planner asks the broker for service mirrors matching a type description and property constraints (if any), and the broker is responsible of returning the service mirrors that match the description.

An instance of the QuA platform consists of a small *core* that may be extended with specialized, domain-specific services. A QuA *capsule* represents the local runtime environment that a QuA platform instance depends on. A capsule hosts one or more *repositories*, where *blueprints* referred to by service mirrors can be stored and retrieved. Capsules themselves are advertised as service mirrors, and can be discovered by service planners looking for nodes to interpret QuA blueprints and instantiate services from it. For this purpose, each blueprint specifies a dependency to the required type of QuA platform, such as a QuA:Java or QuA:Smalltalk platforms.

Thus, a challenge for the design of the P2P broker is to map service mirrors to P2P nodes, and to provide efficient filtering both on type of functionality and properties.

## 3 Related Work

The discovery of metadata in QuA is based on required service types and potentially required static properties of implementations of those service types. We limit the discussion of related work to resource discovery approaches similar to that of QuA—*i.e.*, resource discovery through some form of marketplace often referred to as a trader or a broker. We therefore focus on systems where resources are traded based on type conformance and matching of properties.

Two representative systems for resource discovery that are similar to the approach of QuA are Jini [4] and the ODP/CORBA trading service [5]. Jini is able to operate in a ubiquitous environment, as it has mechanisms for discovering the trading function—*i.e.*, the lookup server—dynamically. Once a binding has been established to the lookup service, Jini trading operates in a similar way to that of the QuA broker. The ODP-trader is part of a middleware framework and also operates similarly to the QuA broker. A more recent trading-like resource discovery service is the *Universal Description Discovery and Integration* (UDDI) registries of Web Services [6]. A service provider publishes the services it is willing to share with others in a UDDI registry, which announces their availability to interested customers. A service consumer accesses the UDDI registry to retrieve the relevant announcements, which describes where and how the services can be invoked. The main difference between Jini, ODP trader, UDDI registry, and QuA is

the way resources are modeled. In ODP, resources are modeled as service offers, while in UDDI resources are modeled as WSDL documents. But more importantly, neither Jini, ODP trader, nor UDDI registries have been designed for a P2P system architecture.

Twine [2] builds on the *Intentional Naming System* (INS), which focuses on resource discovery in the mobile domain. Resources in Twine are represented by a resource description consisting of <*attribute*,*value*> pairs. Twine creates trees of these pairs as hierarchical structures of attribute types are possible. On resource advertisement, *strands* are constructed from the trees 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. Twine relies on Chord [7] to distribute responsibility for resources among participating nodes. However, Twine is not intended for use in component-based middleware, and has no need for strong association of types to resources.

JXTA [8] uses messaging for advertisement of resources fitting a P2P environment, allowing for creation of module types in advertisements. In our context, it is crucial to find all advertised metadata describing services of a given type in the network. Even if the introduction of the *Shared Resource Distributed Index* (SRDI) makes it more likely that information that belong together are grouped to one rendezvous peer, JXTA provides no guarantees for finding all the published pieces of metadata describing a specific type. Furthermore, nodes willing to take the role as *supernodes* (rendezvous peer or gateway peer) might not be available in all situations.

It is therefore interesting to investigate the feasibility of designing and implementing a broker component for QuA based on P2P technology.

## 4 Design of the P2P-based Broker Service

In QuA, both service blueprints and capsules are described by service mirrors, which are frequently retrieved in the planning and re-planning phases of applications. During these phases, the QuA planner uses the trading features of the broker to filter out the most useful service mirrors.

Our design approach consists in creating a P2P network and distributing the service mirrors evenly on participating nodes. The network is self-organizing, and the nodes will at all times agree upon which nodes are responsible for the different service mirrors. Service mirrors are also replicated to additional nodes, which makes the metadata highly available and independent of any central entity.

The main design issues of the P2P broker include choice of P2P technology and its integration into the QuA architecture (cf. Section 4.1), the mapping of service mirrors to P2P nodes (cf. Section 4.2), and the replication scheme that makes service mirrors highly available (cf. Section 4.3).

### 4.1 Choice of P2P Technology

While ideally the design should be independent of any specific P2P technology, it is important for service planners to be able to discover all possible service mirrors describing services of a specific type. For this reason it is preferable to use a structured P2P overlay. Unstructured P2P technologies, such as the well known Gnutella protocols, make use of broadcasts of query messages to find resources in the network. To avoid these broadcasted messages strangling the network, a maximum number of hops is usually specified for these messages. Because of the way these systems construct their networks, this feature provides no guarantee for finding all resources that exists in a network matching a specific query. This is in conflict with the requirement of finding *all* service mirrors matching a service type and a set of properties when requested by the service planner. In structured P2P overlays, each participating node is assigned a *unique identifier* (UID) from a global identifier space. Every node is typically responsible for a contiguous area of the identifier space and receives all messages sent to any UID in this space. This area usually consists of a set of UIDs that are numerically closer to the node's own UID than to the UID of any other node. When a node joins or leaves the network, the neighboring nodes in the UID space are affected by this as their area of responsibility grows or shrinks.

By mapping service mirrors to UIDs, we effectively assign responsibility for them to nodes in the network. The node that is responsible for this area of the UID space will at any time be responsible for that service mirror. In our design, a collection of P2P brokers constitute a distributed system that works as a whole to provide a service that is common and equal to all interconnected instances of the software.

Figure 1 depicts four instances of QuA deployed on four nodes of a P2P network. Each instance is composed of a Core hosting two pluggable services: the P2P broker and the Service planner. The Service planner uses the local P2P broker to retrieve service mirrors that are suitable for an adaptation. The local P2P broker interacts with networked P2P brokers to find all the relevant service mirrors. The P2P broker itself is composed of three parts. The P2P part contains the implementation broker logic. Communication with other P2P brokers is maintained by the Overlay part, while the Glue part glues those parts together and assists in calculating replication of the data.

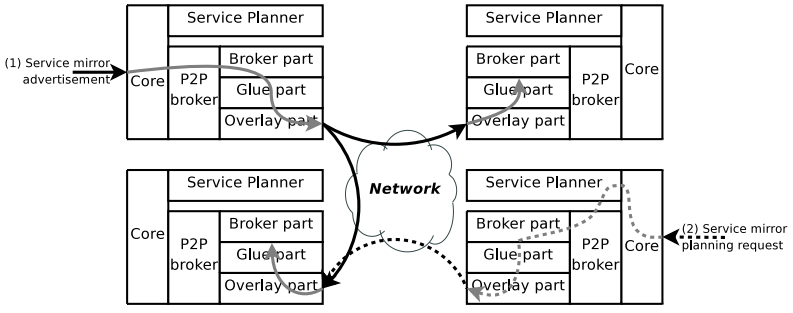


**Fig. 1.** P2P broker and planner services in QuA.

### 4.2 Mapping of Service Mirrors to Nodes

To be able to distribute the metadata evenly on the nodes participating in the network, we need a way to map service mirrors to nodes. As there already exists hashing mechanisms in the P2P technologies that handle the creation of keys that fit in the UID space based on some input (*e.g.*, a string of characters) the problem boils down to extracting data from the service mirrors to use as basis for the generation of keys. It has been shown by many projects, including the PAST persistent storage project [9], that structured P2P overlays can be used to effectively distribute storage of large quantities of data between nodes. Our problem is different because the QuA planner searches for all service mirrors conforming to a service type and property constraints and not for a specific mirror. In practice, this means that instead of creating unique keys for each service mirror (like *e.g.*, PAST creates a key for each individual file), we need to group service mirrors together in a way that makes it easier to find all the relevant service mirrors when they are needed.

Both CAN [10], Chord [7], Pastry [11], and Tapestry [12] create keys and UIDs with a hashing function to ensure an even distribution of UIDs in the identifier space. These hashing functions always create the same UID from the same input. Thus, an intuitive way of mapping service mirrors to nodes uses only the service type specified by the service mirror as basis for key generation. Both nodes advertising and querying for service mirrors find out which service type the service mirror specifies, and uses the string representation of the type as a basis for calculation an overlay specific key for the resource (known as *key-base*). A consequence of this approach is that all service mirrors associated to the same service type, will be mapped to the same key and hence to the same node. Unfortunately, this means that when there are many service mirrors specifying the same service type, such as when there are many instances of a specific capsule type, they will all be mapped to and thus hosted on the same node. Furthermore, each time a service planner asks the broker for capsules that can host a service, all those queries will end up as incoming messages at the node responsible for that key-base.

Besides, if in its request to the broker, the service planner specifies a property constraint, such as a version or a location constraint, then the receiving node has to search through all the service mirrors to find the capsules that match the constraint. If several service planners plan concurrently, this may be time consuming. This problem will become cumbersome for any service type referenced by many service mirrors, and often retrieved by service planners. In order to address this issue, we associate more than one key to each resource advertised. By using more storage space for each resource, and distributing it on participating nodes, the search space and thus the time for queries can be reduced. In addition, as explained below, the requests for service mirrors for specific service types will be distributed over more keys, and implicitly over more nodes, ultimately ensuring a better distribution of queries. We achieve this by creating more than one key-base when the advertised service mirror specifies values for *enumerated properties—i.e.*, properties that have an enumerated type.

In general, a service mirror can be characterized by the pair $[T, < p_0, \ldots, p_n >]$ where T is the type specified by the service mirror, $< p_0, \ldots, p_n >$ is an ordered list of the enumerated properties, and each $p_i$ draws its value $v_i$ from an enumeration domain $D_i$ . Then, we can define a set of key-bases in the advanced mapping method, where

each key-base is defined as:

$$\text{key-base} = T + x_0 + \cdots + x_i + \cdots + x_n$$

where $T$ is the type of the service, $x_i \in \{v_i, \emptyset\}$ (where $v_i \in D_i$, while $\emptyset$ is the empty string representing a wild card), and the operator $+$ indicates string concatenation. To ensure that key-bases based on enumerated properties are generated in the same way in all participants of the network, property values are alphabetically ordered based on the names of the property types for the service type associated to the mirror.

The P2P broker receiving the initial advertisement of a new mirror goes through the property set of the service mirror and finds all enumerated properties that are specified. Then, by following the property type ordering, it creates key-bases for all possible combinations of the service type and values of the enumerated property types where the values are either the value specified by the service mirror, or a wild card. Key-bases for each combination of property value or wild card are created in order to later match the key-base generated from any broker query when requesting a particular service mirror. This means that a P2P broker that receives a resource query for a service type and some properties, creates *one* key-base based on the service type wanted and the enumerated properties specified in the service mirror. This key-base will be in the form:

$$\text{key-base} = T + v_0 + \cdots + v_i + \cdots + v_n$$

where $T$ is the type of the service, $v_i \in D_i$ ($D_i$ is the enumerated domain of property $i$), and the operator $+$ indicates string concatenation.

We illustrate this idea by the following example (see Figure 2). The network of P2P brokers can be seen as a *distributed hash table*, where each node has responsibility for a unique part of the UID-space. To simplify, we assume that there is only one enumerated property type for a capsule, specifying the type of code hosting capabilities it has. We also assume that a capsule, whose type is `QuA-capsule`, exclusively hosts either Java or Smalltalk code. In other words, the value range of the property *hosting capabilities* is strictly enumerated to `Java` and `Smalltalk`.

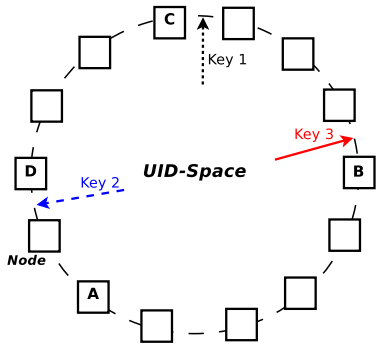


**Fig. 2.** Mapping of service mirrors to nodes.

In our example, there are three possible key-bases that can be created from the type and the enumerated property, of which any announced mirror at most can produce two. The three resulting key-bases are `QuA-capsule`, `QuA-capsule:Java`[4] and `QuA-capsule:Smalltalk`. Now, as we have more than one key-base for each service mirror, we associate one key with each key-base (cf. Figure 2). For example, key `1` is `QuA-capsule`, key `2` is `QuA-capsule:Java`, and key `3` is `QuA-capsule:Smalltalk`. If node `A` advertises its own capsule as a resource, and the 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. Further, 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 reader may notice that it is possible for more than one key belonging to a service mirror to be associated with one node. However, the probability of this happening decreases linearly with the number of nodes joining the system. Further, the broker is able to distinguish resources based on key-bases anyway.

The multi-key mapping technique described above has two main advantages with regards to filtering of service mirrors in the broker. Service description including enumerated properties are processed by the responsible node that match the requested properties. This ensures that service mirrors exhibiting at least one incompatible property are excluded from the research. And, because of the multi-key advertising, the different service requests (most often) end up at different nodes, increasing parallelism of metadata filtering during service planning.

### 4.3 Replication of Service Mirrors

As nodes join and leave a network of P2P brokers, the areas of responsibility for nodes change dynamically. As a result, when a P2P broker of a given node is asked for service mirrors conforming to a service description, the node that is responsible for the corresponding key might not be the same as the one that processed the advertisement message initially. In the same way, when nodes responsible for service mirrors leave the network, that information will be lost if it is not taken care of by some replication mechanism. If not, the design will not ensure metadata robustness to node failure. Thus, we support built-in replication in the system to handle both the self-organization and the metadata robustness concerns.

A key feature of any structured P2P network is that the placement of nodes in the global identifier space relative to each other is organized. This means that it is always possible to calculate who is the closest neighbor in a given direction in the UID-space. These systems also have strict algorithms on routing messages, always routing to the node responsible for the given area that the destination UID of the message lies within. The combination of these two properties gives the opportunity to use the immediate

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

neighbors of a given node, node B, as its replicating nodes. If node B unexpectedly leaves the network, one of its immediate neighbors, node R1, will become in control of the area of the UID-space node B was responsible for (cf. Figure 3). In effect, node R1 will receive all messages regarding the objects that node B just recently had. In some cases where node B had multiple resources associated with different keys, the responsibility of resources may be spread amongst several immediate neighbors. In any case, replicating data on the nearest neighbors to node B solve the problem. In addition, each node must recalculate its immediate neighbors and area of key-responsibilities frequently so that replicas remain up to date.

In a special case, node B may leave and a node E may join the network with an UID that lies between node B and node R1 in the key-space. In this case, constant monitoring of the routing state of node R1 discovers the arrival of node E. By recalculating the responsibility-area of UIDs, R1 and E are able to figure out which keys node E is responsible for, and arrange for R1 to send the relevant data to it.

Now consider Figure 3 that basically illustrates how replication would work in a Pastry network. A resource gets advertised at node A. Node A calculates the UID[5] that corresponds to the service mirror and sends an advertising message to the network. Node B is the node responsible for that UID, so node B receives the message. The first thing node B does is to *replicate* the service mirror by sending a message to the $k$ replicating nodes R ($k$ is known as the replicating factor).

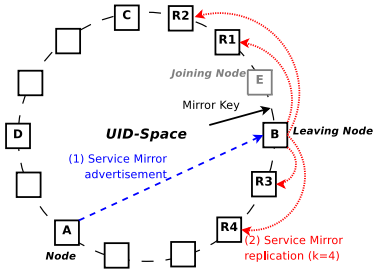


**Fig. 3.** Replication in the P2P broker.

Replication like this has already been implemented with PAST [9,11] over the Pastry technology, and it can be shown that it is possible to implement over a Chord, Tapestry, or CAN network, even though the CAN identifier space differs significantly from the others (at least for a high number of dimensions).

### 4.4 Outdated Information in Service Mirrors

Although our design ensures that the metadata is highly available even with nodes joining and leaving the network due to the replication scheme, we can not be sure that the

[5] Or possibly multiple UIDs from multiple key-bases if running the advanced mapping method, but in this example it is only one key.

information in the service mirrors do not get outdated. As nodes that host blueprints or even instantiated and running components that have been advertised to the P2P broker leave the network, the service mirrors describing these services will become out of date. Likewise, service mirrors describing capsule resources will also contain outdated information when the nodes hosting those capsules disappear from the network.

We assume that, on average, nodes stay for a while once they have joined the network of P2P brokers. As QuA is used for planning and frequent re-planning of adaptive applications, the time used to plan and re-plan applications has to be kept at a minimum in any case. Whenever the service planner discovers that a service mirror contains outdated information, it will ask the P2P broker to discard those service mirrors. Further, we believe that the problem of outdated information on where to find blueprints can be solved by creating a P2P-based blueprint repository. By combining this with a policy to only use a local broker to advertise and query for instantiated and running components, we believe that we have an efficient and satisfactory solution for highly available metadata.

## 5 Evaluation

The evaluation of the P2P broker reported in this section has been performed using the Java implementations of QuA and FreePastry [13]. We have conducted performance measurements on a desktop PC with the following software and hardware configuration: Intel Core 2 Duo 2,38 GHz processor, 3GB of RAM, Ubuntu linux distribution, Java Virtual Machine Sun JDK version 1.6.01 build 105. Our experiments focused on validating the following properties of the P2P broker: *scalability*, *self-organization*, and *robustness* to metadata loss by node failure. In particular, we evaluated the algorithm that maps service mirrors to UIDs in order to validate the even distribution of service mirrors among the P2P nodes (cf. Section 5.1). We ran the entire test setup on a single computer to disable the interferences created by the network. The P2P broker we designed balances the load of storing service mirrors, resolving queries, and filtering service mirrors on nodes that have responsibility for hosting the corresponding service mirrors. Replication factor is set to $4$, which means that the four closest nodes to the responsible node are told to replicate the resources. Thus, we expect that the system as a whole will be scalable if the responsibility for service mirrors are well distributed. As a consequence, we also evaluated the resilience of the system to node failures (cf. Section 5.2) before discussing our results (cf. Section 5.3).

### 5.1 Service Mirrors Distribution

The first experience aims at demonstrating the scalability of the distribution of service mirrors among distributed nodes. Thus, Figure 4 shows the results of an experiment where $81,000$ service mirrors specifying different service types are advertised in a P2P network of $300$ nodes. This configuration reflects the deployment of a set of QuA applications and their respective configurations. For different node UIDs along the X-axis, the Y-axis shows in the blue dotted graph the number of service mirrors each node is responsible for, the replicated ones in the green dashed graph, and either responsible

for or replicated in the red graph. Although the graph shows that the responsibilities for service mirrors are not perfectly evenly distributed, it shows that the metadata is shared among all participating nodes. This distribution is justified by the way Pastry distributes keys responsibility to nodes. For example, when $300$ nodes are randomly assigned UIDs in an UID-space of $2^{128}$, the nodes form clusters in the UID-space. The nodes at the edges of these clusters get responsibility for a high number of keys, while those at the center of the cluster will have neighbors close by on both sides, and will have a smaller key-space to be responsible for.

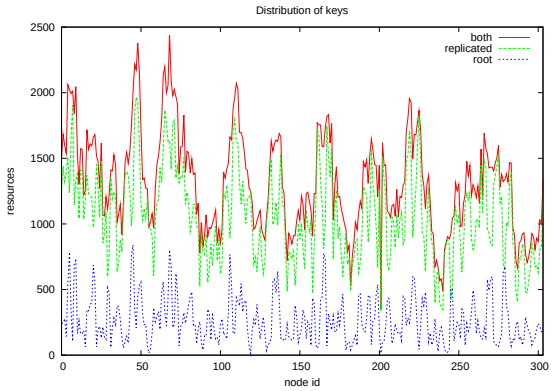


**Fig. 4.** Distribution of service mirrors over nodes.

### 5.2 Service Mirrors Availability

To demonstrate that the system is able to reorganise the distribution when new nodes are joining, we performed series of experiments where an initially small network is joined by a stream of new nodes. In Figure 5, we observe the evolution of $3$ resources (qua:/VideoBuffer, qua:/Component1, and qua:/Printer), which are initially advertised to a network composed of $11$ nodes. The X-axis depicts time-stamps, while the Y-axis shows UIDs in the UID-space in base $10$. $100$ nodes are individually joining this initial configuration following a rate of about $1$ node per second. Immediately upon joining the network, each joining node tries to discover all service mirrors. The responsible node for answering service mirror requests is allocated to the closest node with regards to the resource key requested. Monitoring shows how the responsibility for responding to the queries shift as new nodes join the network. Thin lines in the graph can be explained by the delay introduced by FreePastry to re-organize the distribution and re-allocate responsibilities.

Similarly, as depicted in Figure 6, we have performed experiments where we have started each scenario with an initial network composed of $100$ interconnected nodes. Subsequently, each node leaves the network unexpectedly at a random time ($60$ seconds on average). Initially, $3$ service mirrors were advertised to the network

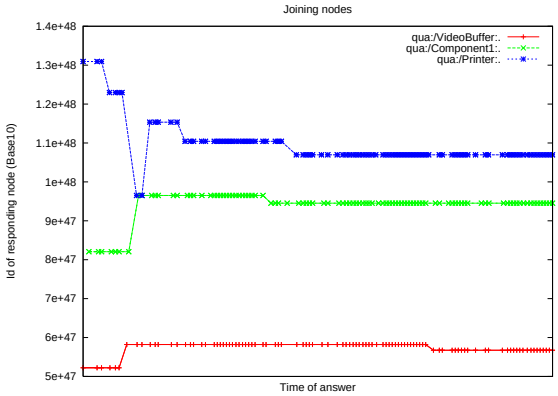


**Fig. 5.** Service mirrors responsibility when nodes join.

(qua:/VideoBuffer, qua:/Component1, and qua:/Printer). In these experiments, a number of nodes are constantly querying the three resources to monitor their availability. As nodes responsible for certain service mirrors left the network, we observed that the closest node in the UID-space answered queries as expected.

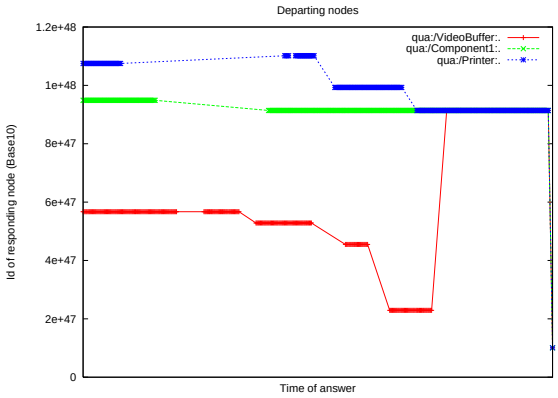


**Fig. 6.** Service mirrors responsibility when nodes leave.

### 5.3 Discussions

This section discusses the results we obtained with regards to the observed properties: *scalability*, *self-organization*, and *robustness* to metadata loss by node failure.

*Distribution and scalability.* As mentioned in Section 5.1, the service mirror distribution we obtained is not perfectly even. The reason is that the resources are assigned

keys that are fairly well distributed in the UID-space, but node identifiers are clustered. Given that the resources are replicated to the closest nodes in the UID-space, some nodes get responsibility for a larger range of keys than other nodes. Nevertheless, even if the distribution is not perfect, the P2P-broker seems to be able to take advantage of Pastry's properties with respect to distribution of resources. This make the P2P-broker able to scale as the numbers of nodes and resources increases.

*Self-organization.* In Figure 5 and Figure 6, thin lines depict the reorganization of responsibilities and replicas when the network topology evolves. These thin lines represent temporary unavailability of resources and forces the QuA planner to wait for the service mirrors. The temporary unavailability of resources is due to FreePastry, which tries to send a message via a route that is unavailable because of a recent node failure. When the node failure is discovered, FreePastry routes the message along a different path. This delay can be reduced by decreasing the message timeout values for messages in the FreePastry implementation. However, if the timeout is set too low, messages will be frequently re-sent even if an answer to the message is underway, and consequently much resources will be wasted on sending duplicate messages.

*Robustness.* Figure 5 and Figure 6 validate the replication scheme. In particular, it is capable of replicating resources to new nodes to try to keep the invariant that $k + 1$ nodes should hold every resource in the system. However, the dynamism of the P2P environment breaks regularly this invariant and the replication scheme has to detect these situations to restore the system back into a consistent state. The capacity of the replication scheme to handle node failures depends on the frequency of node failures. In particular, [11] has shown that the lazy repair algorithms of Pastry allows the network, over time, to completely recover from a drop of 10% of $100,000$ nodes. When increasing the frequency at which nodes leave the network, we reach a point where the replication scheme fails. Observations show that this happen when the number of nodes equals the replication factor—*i.e.,* the number of nodes replicating each service mirror—dropped out in a time interval close to the *keep-alive* refresh interval of FreePastry.

## 6 Conclusions

In this paper, we have introduced the design and the implementation of a P2P broker for the QuA planning-based adaptation middleware. The QuA middleware uses the broker to retrieve metadata in the form of service mirrors describing implementation alternatives conforming to a service description. The description includes the type of service required and constraints on other properties of the service implementation. The broker is also used to advertise new service mirrors.

While the original broker for QuA was designed for a traditional client-server architecture, this paper has investigated the feasibility of implementing the QuA implementation broker using P2P technology. A particular challenge addressed in this paper was the mapping of service mirrors to nodes in the network to provide an even distribution of metadata over the nodes. While enabling better scalability of query processing, this paper has also shown that the QuA broker can tolerate node failures.

Our perspectives include large-scale deployment of the QuA middleware to evaluate the performance of the P2P broker using a time metric (*e.g.*, response-time, throughput).

## Acknowledgements

The authors thank the partners of the MUSIC project and reviewers of the DAIS conference for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

## References

1. Eliassen, F., Gjørven, E., Eide, V.S.W., Michaelsen, J.A.: Evolving Self-Adaptive Services using Planning-Based Reflective Middleware. In: 5th Int. Middleware Workshop on Adaptive and Reflective Middleware (ARM). Volume 190 of AICPS., ACM (November 2006) 6
2. Balazinska, M., Balakrishnan, H., Karger, D.: INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In: 1st Int. Conference on Pervasive Computing (Pervasive). Volume 2414 of LNCS., Zurich, Switzerland, Springer (August 2002) 195–210
3. Bracha, G., Ungar, D.: Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In: 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Vancouver, BC, Canada, ACM (November 2004) 331–344
4. Sun Microsystems: Jini Architecture Specifications - v2.1. (2005) `http://www.sun.com/software/jini/specs`.
5. Bearman, M.: Tutorial on ODP Trading Function. Faculty of Information Sciences Engineering. University of Canberra. Australia. (February 1997)
6. OASIS: UDDI Version 3.0.2. (February 2005) `http://uddi.xml.org`.
7. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Int. Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM), San Diego, CA, USA, ACM (August 2001) 149–160
8. Sun Microsystems: JXTA Protocol Specification - v2.0. 2.5.3 edn. (October 2007) `https://jxta-spec.dev.java.net`.
9. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In: 8th Int. Workshop on Hot Topics in Operating Systems (HotOS). CNF, Schoss Elmau, Germany, IEEE (May 2001) 75–80
10. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Int. Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM), San Diego, CA, USA, ACM (August 2001) 161–172
11. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Int. IFIP/ACM Conference on Distributed Systems Platforms (Middleware). Volume 2218 of LNCS., Heidelberg, Germany, Springer (November 2001) 329–350
12. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22** (January 2003) 41–53
13. : FreePastry. Rice University, Houston, USA. and Max Plank Institute for Software Systems, Saarbrucken, Germany `http://freepastry.org`.