Peer-to-peer brokering of planning meta-data

Johannes Oudenstad  
Norwegian Defence Research Establishment,  
P.O. Box 25,  
N-2027 Kjeller, Norway  
johannes.oudenstad@ffi.no

Frank Eliassen  
University of Oslo,  
Dept of Informatics,  
P.O. Box 1080 Blindern  
N-0314 Oslo, Norway  
frank@ifi.uio.no

Eli Gjørven  
Simula Research Laboratory,  
P.O.Box 134,  
N-1325 Lysaker,  
Norway  
eligj@simula.no

Romain Rouvoy  
University of Oslo,  
Dept of Informatics,  
P. Box 1080 Blindern  
N-0314 Oslo, Norway  
rouvoy@ifi.uio.no

ABSTRACT

In self-adaptive systems, information about resources, i.e., services, nodes etc., in the system must be dynamically published, updated, and removed. Current middleware approaches use statically configured, centralized repositories for storage and retrieval of such information. However, in the context of peer-to-peer (P2P) environments, we cannot assume the existence of server nodes that are always available for hosting such centralized services. Thus, in our planning-based, adaptation middleware, we introduce the P2P resource broker, which is a meta-data advertisement service based on P2P technology. We use a structured P2P protocol that distributes the service meta-data over a set of nodes based on service type and property information. By exploiting type and property information, we enable filtering the query results in each node, thus potentially reducing the query response time considerably, even for large amounts of meta-data. Furthermore, the P2P service broker handles node failures by providing replication of the meta-data.

We present a working prototype P2P service broker and results from initial experiments. The results of the experiments show that the meta-data distributes well over the nodes in the network thus enabling scalability and otherwise confirm our claim about robustness to node failures.

Keywords

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 lower level OS resources. A self-adapting system is able to reason about itself at runtime, and if necessary make changes to itself in order to better satisfy the current requirements. However, while distributed applications traditionally were built from client-server architectures, many current distributed applications are built from peer-to-peer (P2P) architectures, where the system consists of equal, autonomous peers that only to a varying degree adjusts its activities to the requirements of the distributed application. Such a peer may be willing to share its resources, or it may not. Peers entering an application when suitable for themselves, and then leaving without warning, must be considered the common case rather than an exception. Many approaches to adaptation middleware assume the existence of server nodes that can be expected to almost always be available, and 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 on the middleware level.

In this paper, we focus on the problem of dynamically locating information about resources, such as services, nodes, etc., in the system as they become available. In the area of adaptation middleware, many approaches use statically configured, centralized repositories. 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 [4]. In the QuA middleware, adaptation is driven by meta-data associated to services (e.g., service performance or cost), thus the system aims to provide the best possible Quality of Service (QoS) to users under variable execution contexts. However, this requires quick and simple means for querying for meta-data in the system. The P2P broker 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 meta-data across the network nodes to improve the planning processing performance. Besides, the replication of meta-data using the P2P network ensures the meta-data high-availability in terms of access time, fault tolerance, and network participants connectivity. An evaluation of these properties as well as the distribution quality are provided to illustrate the benefits of our proposition.

In the remainder of the paper, we relate to existing work in the domain of resource brokering (Section 2), and we introduce the QuA planning middleware (Section 3). Then, we
present the design of our P2P-based implementation broker for the QuA planning middleware (Section 4), and an evaluation of its performances (Section 5). Finally, we discuss the perspectives of this work and we conclude (Section 6).

2. RELATED WORK
The discovery of meta-data 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 [9] and the ODP/CORBA trader [2]. 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 resource registry of Web Services (WS) [5]. A service provider may publish the services it is willing to share with others in a WS registry, which announces their availability. A service consumer may access the WS registry to retrieve the relevant announcements, which describes where and how the services may be invoked.

The main difference between Jini, ODP trader, WS registry and QuA is the way resources are modelled. But more importantly, neither Jini, ODP-trader nor WS registry have been designed for a peer-to-peer system architecture. On the other hand, resource discovery mechanisms in peer-to-peer systems aim to distribute the resources and thereby the load of the system on all participating entities. However, these mechanisms do not provide anything similar to a trading service. It is therefore interesting to investigate the feasibility of designing and implementing a broker component for QuA based on peer-to-peer technology.

3. THE QU A MIDDLEWARE
The QuA middleware [4] supports planning-based adaptation, which means that applications are specified by their behavior, and 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, 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 (i.e., type) and implementation (referred to as a blueprint). Each service mirror has a map of (property name, value) pairs, where the range of property types that are 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.

The task of service planning, is to plan the initial configuration or the reconfiguration of a service. The planner is responsible for evaluating alternative service mirrors in order to find and select the service implementation with highest utility that satisfies 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 implementation broker is a trader-based discovery service. The resources traded in the broker are the service mirrors discussed above. Component developers and application developers alike must advertise the service mirrors to the broker. The broker has a responsibility of hosting all mirrors advertised to it in a repository. In the service planning phase, the planner asks the broker for service mirrors matching a type description and property constraint (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 mirrors can be stored and obtained. Capsules themselves are advertised as service mirrors that 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 platform.

A challenge for the design of the peer-to-peer QuA broker is how to map service mirrors to nodes in the peer-to-peer system, and how to provide efficient filtering both on type of functionality and properties. E.g., filtering should preferably be performed at the node responsible for storing the resource.

4. P2P BASED BROKER DESIGN
In this section, we present our design for the peer-to-peer based implementation broker, from now on called P2P broker. We start by presenting an overview of the concept in Section 4.1. In Section 4.2, we discuss how service mirrors are mapped to connected nodes in the system. The replication scheme that makes sure that the service descriptions have high availability even with node departure and failure is discussed in Section 4.3.

4.1 Overview
In QuA, both service blueprints and capsules are described by service mirrors. In the planning and re-planning phases of applications, these descriptions are accessed frequently. In the planning phase the QuA planner uses the trading features of the broker to filter out the most usefull mirrors.

The basic approach of our design is to create a peer-to-peer network and distribute the service mirrors evenly on participating nodes. The network is self-organizing, and the nodes will at all times agree upon which nodes that are responsible for the different service mirrors. Service mirrors are also
replicated to additional nodes. In this way, the meta-data is made highly available and independent of any central entity.

Our design is independent of any specific technology, but as it is important for service planners in the planning phase to be able to discover all possible service mirrors describing services of a specific QuA type, it is necessary to use a structured peer-to-peer overlay. In our prototype implementation we have used an implementation of the Pastry technology.

In structured peer-to-peer overlays, each participating node is assigned a unique id from a global identifier space. Every node in a structured peer-to-peer overlay is typically responsible for a contiguous area of the identifier space and receives all messages sent to any id in this space. This area usually consists of a set of ids that are numerically closer to the node’s own key than to the key of any other node. Every time a node joins or leaves the network, the neighbouring nodes in the id space are affected by this as their area of responsibility grows or shrinks.

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

### 4.2 Mapping service mirrors to nodes

To be able to distribute the data 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 peer-to-peer technologies that handle the creation of keys that fit in the id 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[3] persistent storage project, that structured peer-to-peer 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 service mirrors conforming to a type and property constraint 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 easy to find all the relevant service mirrors when they are needed.

Both CAN[6], Chord[7], Pastry[8] and Tapestry[10] create keys and Ids with a hashing function to ensure even distribution of Ids in the identifier space. These hashing functions will always create the same id from the same input.

The most intuitive way of mapping service mirrors to nodes uses only the QuA type specified by the service mirror as basis for key generation. Both nodes advertising and querying for service mirrors find out which QuA type the mirror specifies and uses the string representation of the type as a basis for calculation an overlay specific key for the resource. That basis will be known as keybase.

A consequence of this approach is that all service mirrors associated to the same QuA type, will be mapped to the same key and hence the same node. Unfortunately, this means that when there are many service mirrors specifying the same QuA 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. Worse yet, each time a service planner asks the broker for capsules that can host the components that constitute the service, all those queries will end up as incoming messages at the node responsible for that key.

If, in its request to the broker the service planner specifies a property constraint such as a version constraint or location constraint, 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 a reality for every QuA type that is specified by many service mirrors, and that are often asked for by service planners.

In order to address this issue, we may associate more than one key to each resource advertised. By trading between more storage space usage for each resource, and distributing it on participating nodes, the search space and thus time for queries which specifies needs through property constraints can be reduced. In addition, as will be explained below, the requests for service mirrors for specific QuA types will be distributed over more keys, and implicitly over more nodes, ultimately ensuring a better distribution of queries. We will do this by creating more than one keybase in the cases where the advertised service mirror specifies values for enumerable properties, where enumerable property is defined as “a property that has an enumerable value range”.

In general, a service mirror can be characterized by the tuple $T, p_0, \ldots, p_n$ where $T$ is the type specified by the service mirror, $p_0, \ldots, p_n$ are property attributes, and each $p_i$ draws its value $v_i$ from an enumeration domain $D_i$. From that information, we can define keybases in the advanced mapping method to in general be the form:

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

where $x_i \epsilon \{v_i, s\}$, $v_i \epsilon D_i$, $T$ is the type of the service, $v_i$ represents the value for the $i$th property type for the service mirror, $s$ is the empty string representing a wildcard, and the $+$ marker indicates string concatenation. To ensure that keybases based on properties are generated in the same way in all participants of the network, an ordering of the property values have to be made. The ordering is based on the names of the property types for the QuA type specified for
the mirror, in increasing order.

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

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

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

where \(v_i \in D_i\), \(T\) is the type of the service, \(v_i\) represents the value for the \(i\)th property type for the service mirror in increasing order, and the + marker indicates string concatenation.

We illustrate this idea by the following example (see figure 1). The network of P2P-brokers may be seen as a distributed hash table, where each node has responsibility for a unique part of the key-range. To simplify, we will assume that there is only one enumerable property type for a capsule, specifying the type of code hosting capabilities it has. For simplicity, we also assume that a capsule may only host either Java or Smalltalk code, not both, and nothing else. In other words, the value range of the hosting capabilities property is enumerated to Java and Smalltalk, and it is not possible to have multiple values.

In our example there are three possible keybases that can be created from the type and the enumerable property, of which any announced mirror at most can produce two. The three keybases are *QuA-capsule, QuA-capsule:Java* and *QuA-capsule:Smalltalk*. Now, as we have more than one keybase for each service mirror, we associate one key with each keybase (again consult figure 1). Imagine that *QuA-capsule* gives key one, *QuA-capsule:Java* gives key two, and *QuA-capsule:Smalltalk* gives key three. If node A advertises its own capsule as a resource, and that capsule has capabilities of hosting Smalltalk code, both node C and node B would assume responsibility of hosting that resource, but under different conditions. Node C would now respond to all queries for a capsule without any preferences to hosting capabilities. Node B would respond to all queries for a capsule that has capabilities of hosting Smalltalk code. 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 observant 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 keybases anyway. The only negative effect would be that one node has multiple service mirrors of the same type without really needing to, thus it will use unnecessary storage space.

The multi-key mapping technique described above have two main advantages with regard to filtering of service mirrors in the broker:

- If the service description used when querying for service mirrors specifies values for a number of enumerable properties, then the multi-key advertising ensures that when this request ends up at the node responsible for the given key, none of the advertised service mirrors that match the QuA type but not the specified property values will be in the set that has to be searched through to find the set of mirrors that match all properties specified by the service description.

- Because of the multi-key advertising, the different service requests that have matching QuA types, specify different values for some enumerable properties will end with different keys, and (most often) end up at different nodes, increasing parallelism of meta-data filtering during service planning.

4.3 Replication

As nodes join and leave a network of P2P-brokers, the areas of responsibility for nodes change. As a result, when a P2P-broker at one node is asked for service mirrors conforming to a specification, the node that is responsible for the corresponding key might not be the same as the one that got the advertisement message in the first place. For the system to be self-organizing, a mechanism has to counter this problem. In the same way, when nodes that have responsibility for certain service mirrors leave the network, that information will be lost if it is not taken care of by some mechanism. If not, the design will not ensure meta-data robustness to node failure.

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

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

Now consider figure 2 showing how replication would work in a Pastry network. The figure illustrates the basic idea. A resource gets advertised at node A. Node A calculates the Id that corresponds to the service mirror and sends an advertising message to the network. Node B is the node responsible for that Id, so node B receives the message. The first thing node B does is to replicate the service mirror by sending a message to the replicating nodes R.

Replication like this has already been implemented with PAST[7][3] 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 advertised service mirrors

Although our design ensures that the meta-data is highly available even with nodes joining and leaving the network due to the replication scheme, we cannot be sure that the 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 metadata describing these services will be 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 have to be kept at a minimum in any case. We are currently working on the details of a solution that allows the server planner to ask the P2P-broker to remove service mirrors that has been discovered to contain outdated information.

Further, we believe that the problem of outdated information on where to find blueprints can be solved by creating a peer-to-peer based blueprint repository. By combining this with a policy to only use a local implementation broker to advertise and query for instantiated and running components, we believe that we have an efficient and satisfactory solution for highly available meta-data.

### 5. Evaluation

To be able to evaluate our design, we have completed a Java implementation with which we have performed some initial experiments. In this section we present our results and evaluation. We have used the FreePastry[1] Java implementation of the Pastry technology.

Our aim is to evaluate the following three aspects of the performance of the P2P-broker: scalability, self-organization and robustness to data loss by node failure. For the initial experiments reported in this paper, we ran the entire test setup on one computer. As all nodes are run on one computer, using time as a metric to test the P2P-broker solution compared to any other solution would not make sense. Therefore, for the test of scalability, we use a distribution experiment to observe if the system is able to leverage the facilities in the Pastry network to distribute the service mirrors relatively evenly on networked nodes. We rely on the good results on messaging efficiency shown by the Pastry team[7] that messaging between nodes will be scalable. The design presented above will distribute the load of storing service mirrors, resolving queries, and filtering service mirrors on nodes that have responsibility for hosting the corresponding service mirrors. Thus, we expect that the system as a whole will be scalable if the responsibility for service mirrors are well distributed.

Figure 3 shows the results of an experiment where 81000 service mirrors were announced specifying different QuA types to a network of 300 nodes. For different node IDs along the x-axis, the y-axis shows in the blue graph the number of service mirrors each node is responsible for, replicating in the green graph and either responsible for or replicating in the red graph. Although the graph shows that the responsibilities for service mirrors are not completely evenly distributed, it shows that it is shared among all participating nodes. The reason for this distribution can be explained by the way Pastry distributes responsibility for keys to nodes. E.g., with only 300 nodes randomly assigned IDs in an ID space of $2^{128}$, the nodes will form clusters in the id space. The nodes at the edges of such a cluster will get responsibility for a high number of keys, while those at the center of the cluster will have neighbours close by on both sides, and will have a smaller keyspace to be responsible for.
To show that the system is able to reorganize when new nodes are joining, we have performed a series of experiments where an initially small network are joined by a stream of new nodes. A small number of resources are announced to the initial network. Immediately upon joining the network, the joining node tries to discover all service mirrors. Through logging, we are able to see how the responsibility for responding to the queries shift as new nodes join the network. The service mirrors remained available throughout all experiments.

Similarly, we have performed experiments where we have started each experiment with an initial network with a high number of interconnected nodes. Subsequently each node leave the network unexpectedly at a random time. A small number of service mirrors were advertised to the initial network. In these experiments, a number of nodes are constantly querying for resources to observe their availability. As nodes responsible for certain service mirrors left the network, we observed that the closest node in the ID space answered query as expected. When we increased the frequency at which nodes left the network, we reached a point where the replication scheme could not keep up the pace. By examining the logging files, we found that this happened when a number of nodes equal to our replication factor (i.e. 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 presented the design and implementation of a peer-to-peer implementation broker for the QuA planning-based adaptation middleware. The QuA middleware uses the implementation broker to retrieve meta-data in the form of service mirrors describing implementation alternatives conforming to a service specification. The specification 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 implementation broker for QuA was designed for a traditional client-server architecture, this paper has investigated the feasibility of implementing the QuA implementation broker using peer-to-peer technology. A particular challenge addressed in this paper was how to map service mirrors to nodes in the network so as to achieve an even distribution of meta-data over the nodes. The latter will enable better scalability of query processing. This paper has also shown how the QuA broker can be made robust to node failures.

Our initial experiments have been conducted on a single machine. In our future work we will perform experiments on a distributed deployment to investigate the performance of the peer-to-peer broker using a time metric.

7. REFERENCES