Distributed Service Discovery with Guarantees in Peer-to-Peer Networks using Distributed Hashtables

Sven Kaffille, Karsten Loesing*, and Guido Wirtz
Distributed and Mobile Systems Group
Otto-Friedrich-Universität Bamberg
Feldkirchenstraße 21, 96047 Bamberg, GERMANY
{ sven.kaffille|karsten.loesing|guido.wirtz} @wiai.uni-bamberg.de
Phone: +49951863{2812|2810|2527}; Fax: +499518635528

Abstract—In this paper we propose a protocol for service discovery in decentralized environments. Our protocol guarantees that service descriptions can be found by any node in the network. It is meant to be used by applications which cannot afford central components, e.g., due to ad-hoc formation. The protocol makes use of the structured overlay network Chord. Chord has logarithmic performance for looking up keys with respect to the number of nodes and guarantees to retrieve any entry stored in the network. This would not be possible in an unstructured overlay network without flooding the whole network. Service descriptions are decomposed into portions which can be efficiently distributed and retrieved. Though this implies additional costs for publishing of service descriptions, it improves efficiency of lookup operations. We implemented a Java prototype of our protocol as a proof of concept based on TCP/IP.

Keywords: Service Discovery, Peer-to-peer, Distributed HashTable

I. INTRODUCTION

In recent years the Web has changed from a rather consumer-oriented to a consumer-and-producer environment. Providing a service is meanwhile understood nearly as usual as consuming a service due to the empowerment of the edge of the Internet. Services are offered by numerous nodes in a decentralized way and service providers may join or leave at will. Examples cover peer-to-peer (P2P) file sharing systems in which every peer offers a service for downloading files to other peers, software agent systems which allow agents to be created at different places in order to serve users or other agents, and Web Services in which any node with an HTTP server running can create its own Web Service and offer it to other nodes in the Web.

All these services are of little use until they are advertised to service consumers by—hopefully computer-readable—service descriptions including their type of service as well as parameters describing service details. This is done by (1) publishing service descriptions at so-called registries and (2) allowing service consumers to query the registry database for services matching certain criteria. The latter may be divided into looking up a certain service with a known identifier which is called name service and searching for services with certain attributes which is known as directory service [1].

Usually registry services are located at central, well-known places running on nodes dedicated only for this task. While being algorithmically simple, this solution has a couple of drawbacks: First of all, central solutions generally do not scale. Server load linearly grows with the number of clients making it necessary to replicate servers whenever network size increases significantly. Further, any server is a single point of failure making the service unavailable when it is disconnected from the network. Apart from that, central or hierarchical structures for registries do not always correspond to the dynamic formation process of service providing environments. Next, servers have to be set up and managed in order to allow collaboration. This represents a barrier for spontaneous collaborations which might inhibit ad-hoc formation and should be avoided. In some situations it may be reasonable to interconnect multiple registries which have been formed apart of each other. This should be done rather in a coequal than in a master-servant setting. Another example is grouping of users which share a common interest or only want to cooperate in a closed group. In these situations a distributed realization of a registry service is more feasible than picking a single node or a fixed set of nodes to provide this task.

There exist means of connecting multiple registries in non-hierarchical fashions. For example in the agent world of FIPA-complying agent platforms [2] the so-called directory facilitators can be federated, so that query requests are forwarded to each other with a given time to live (TTL). Considering another example, UDDI allows connection of multiple registries hierarchically or in a peer-like fashion which is called registry affiliation [3]. Those approaches for connecting registries may be compared with unstructured, pure or hybrid P2P systems like Gnutella [4] and FastTrack [5]. They form decentralized nets of all nodes, respectively specialized super nodes, and forward queries with a given TTL. The problem of these approaches is that—unless a query traverses all (super) nodes in a network by means of flooding—no guarantees can be given for finding a certain resource. Though this might not be a problem in file sharing systems, it is more than just an inconvenience for service-oriented environments which require precise results.

Our approach aims to decentralize the registry in a structured way in order to provide the guarantee of finding any registered service description matching a given query. This
is done by forming an overlay network using the distributed hashtable Chord [6] and distributing the service information in it so that it can be queried by contacting only a logarithmic number of nodes.

Figure 1 gives an example of a registry network that makes use of our protocol. The registry service is distributed among the nodes in the cloud while the nodes outside of it just make use of it without contributing to it. The circles denote administrative boundaries, e.g. of corporations or universities.

II. REQUIREMENTS ON DISCOVERY SERVICES

The requirements on a discovery service imposed by service providers and consumers can be divided into functional and non-functional requirements.

Functional requirements of service providers comprise convenient methods and data structures to publish, modify and unpublish their services. Service descriptions consist of (name, value) pairs describing the attributes of services to publish. Values of service descriptions may contain primitive data types like integer, string, etc., complex data types, or sets of one of these types. Complex data types are composed of (name, value) pairs just like service descriptions. A discovery service must therefore allow publishing, modifying, and unpublishing tree-like complex data structures. There must not be any restriction on the number of attributes a service description can consist of.

Service consumers querying a discovery service need methods to look up services and a convenient data structure to describe templates for services which they are looking for. This data structure must be defined analog to the one used for service descriptions. The discovery service must return all service descriptions matching a query giving service consumers the guarantee to find any available service. Further it should give users the possibility to specify an upper bound for the number of returned services and provide a means for iterating over them.

Beyond functional requirements the discovery service has to satisfy non-functional requirements. Some of these are independent of applying a discovery service in a distributed manner. These are e.g. low response time, reliability, and scalability (depending on the number of services published and number of queries). Some requirements apply only to a distributed discovery service (especially in a P2P environment) as, for example, low bandwidth consumption and low number of messages for publishing, modifying, unpublishing, and querying of services.

III. OVERLAY NETWORK - REQUIREMENTS AND ASSUMPTIONS

The non-functional requirements as well as the guarantee to find all available services directly impose requirements on the underlying P2P overlay network. It has to allow storage and retrieval of service descriptions. Therefore two types of P2P networks could be applied: unstructured and structured networks. Unstructured P2P networks like Gnutella [4] and FastTrack [5] would achieve this by creating local indices on all nodes and forwarding queries through the network until either a result was found, or a given TTL value has run off. Advantages of this approach are simplicity of algorithms and arbitrary complexity of queries. But drawbacks which make this solution unfeasible for a discovery service are bad scalability and incapability of giving guarantees whether a certain service is available, or not.

Structured P2P networks like Chord [6] take another approach. Here, the index used for service discovery is already distributed in the network when registering the service. This is done by routing queries step by step to that node which is responsible for holding the required information. Advantages of structured overlays are efficient lookups which only involve a logarithmic number of nodes in the network. Further, they guarantee any query to be answered even if service provider and consumer reside at two distant edges of the network. A drawback is the requirement of maintenance of a certain network structure in case of joins, leaves, or failures of nodes, which usually is more expensive than in unstructured P2P networks. Another problem is that distributed hashtables do not support searching by themselves, but only looking up data bound to concrete hash values.

The main reason for our approach to use a structured P2P network is the fact that queries are guaranteed to be answered without having to flood the whole network. It has to be evaluated by simulation whether maintenance costs play a crucial role compared to costs of publishing and querying. Choosing the right data structure to publish and retrieve service information in the network is one of the crucial tasks of our approach.

In Chord [6] any piece of information has to be uniquely identifiable by a key. This is achieved by applying a hash function on any data which is to be stored in the network. This key is used for storage as well as for retrieval of data. Consequently, in order to find any information the full key has to be known in advance assuming that one has to know exactly what to look for. Searching capability has to be implemented separately of this lookup mechanism or by using additional
data structures, e.g. inverted indices. All nodes in the overlay network are assigned unique identifiers (ID) of the same key space. Any node is responsible for the data keys within the range of the next smaller node ID in the network up to its own node ID. Therefore every node has to know its predecessor which is propagated and updated by maintenance messages. The ordering of nodes in successor-predecessor relations forms the so-called Chord ring. In addition to predecessor references every node stores a so-called finger table—a skip list containing $i$ references to nodes of which the node IDs are at least the $i$-th power of two greater than their own node ID. By this, queries can be forwarded at least half the way closer to their destination in every step only needing to know a logarithmic number of nodes in the network. This leads to logarithmic performance for storing and retrieving any item stored in the ring. Whenever nodes join or leave the network routing information have to be updated. This is done by a stabilization protocol which has been proven to be correct even in case of multiple node joins or leaves at the same time [6]. In order to prevent data loss because of node failures data should be replicated on multiple nodes in a network, e.g. by copying it on the $k$ next nodes following a node responsible for a specific data item. Figure 2 demonstrates an example of a lookup procedure in Chord.

IV. DESIGN OF THE DISCOVERY SERVICE

The design of the proposed P2P discovery service is covered in this section. This service consists of three layers on top of which an application may be built. The first and lowest layer is the transport layer on top of which the second layer, the P2P overlay network Chord, resides. The third layer is the local discovery service layer which is implemented above the Chord layer. This layer provides methods to publish, unpublish, modify, and query service descriptions. It also maintains a local data structure which stores all service descriptions that have been published by the local node. Parts of the service discovery layer rely directly on the transport layer for direct communication with other nodes of which the addresses are already known by the service discovery layer. Finally, an application may be built on top of the service discovery layer. The architecture is shown in figure 3.

These layers are intended to be implemented on every node in the discovery service network. Hence, every participating peer can publish, unpublish, and modify service descriptions that it wants to provide to and query service descriptions provided by other peers. Alternatively, it is possible that services can be provided to (and provided by) nodes not directly participating in the P2P discovery service. This is done by connecting to one or more nodes which are part of the discovery service.

The following sections show how service descriptions are published, unpublished, modified, and queried with help of the provided architecture.
A. Publishing service descriptions

The service discovery layer includes a data structure for representing a service description. Within this data structure various attributes describing a service can be set. These attributes consist of (name, value) pairs. Values can be primitive or complex types, or sets of one primitive or complex type. Primitive types include integer, string, etc. Complex types may contain primitive or complex types as well as sets of them as subtypes. In this way a tree-like structure of types can be built (e.g. figure 4).

\[
\begin{align*}
\text{type} & = \text{ticketservice}; \\
\text{url} & = (\text{protocol} = \text{http}; \\
& \quad \text{host} = 81.200.194.40; \quad \text{port} = 8080;); \\
\text{owner} & = \text{DB}; \\
\text{languages} & = \{\text{german, english, french}\} \\
\end{align*}
\]

Fig. 4. Example of a service description.

For each service description to be published a registration ID is created which later can be used to uniquely identify the published service. Further, keys are generated for the attributes of the service description:

- The keys for attributes with primitive type are generated by calculating the hash value of the concatenation of name and value. For each of these keys a so-called service reference consisting of the local node’s address and the registration ID is stored in the Chord layer using the calculated hash value as key.
- Attributes having a complex type are decomposed into their subtypes. For each of these a service reference is stored in the Chord layer. In order to retain the tree-like structure of the service description the names of attributes of a complex type are preceded by the name of the complex attribute itself. Hence, each attribute is assigned a fully qualified name which allows unique identification of attributes in a description.
- At last, attributes consisting of a set of types are stored one by one as described above. The names of the items in the set are also preceded by the attribute name of the set. The order of values in a set is not mapped to Chord. If such an order is desired, a complex type should be used instead.

The presented mapping of service descriptions to Chord keys preserves the hierarchical structure of attributes, but discards the ordering of its elements. Figure 5 shows an example of the keys generated for the service description of figure 4.

After service references have been stored within the Chord layer for each attribute, the service description itself is stored in the Chord layer using the generated registration ID. The registration ID is returned to the application for later referral to the service description.

B. Querying service descriptions

In order to search for a service, a node has to know the schema—the tree-like hierarchy of attribute names—of the service description of the service it is looking for and at least one value of a service attribute. The attribute types may be primitive or complex types as well as sets of these, comparable to publishing of services. In order to permit multi-attribute queries, templates are used which incorporate all attributes belonging to one query. Figure 6 shows an example of a service description template suitable for retrieving the service description given in figure 4.

\[
\begin{align*}
\text{type} & = \text{ticketservice}; \\
\text{url} & = (\text{protocol} = \text{http}; ); \\
\text{owner} & = \text{DB}; \\
\end{align*}
\]

Fig. 6. Example of a service description template.

Querying for all services matching a given template is done by choosing one of the attributes by random, calculating its hash key, and looking up all available service references for it in the Chord layer. Performance can be improved, if only the service references for the least frequent key are queried which requires an additional data structure maintaining keyword frequency (e.g. see [7]). The node addresses of the returned service references are then used to retrieve the complete service descriptions. It can be tested locally, if these descriptions also match the other attributes contained in the query template. Only service descriptions matching all attributes of the template are returned to the application layer.

Following from this description, templates may only contain complete (name, value) pairs. It is not possible to support wildcard usage in the value part of an attribute, because searching for them cannot be done efficiently in the Chord layer. Searching for ranges of values is not possible at the moment, neither. Both issues might be addressed in future work.

Figure 7 shows the entries of a Chord ring after publishing the service given in figure 4 and performing a query on it.
Fig. 7. Procedure of publishing and querying a service. **Publish:** The service discovery layer generates a registration ID by hashing the given service description (here: 79 for the service description given in figure 4) and a key for each entry of the service description (cf. figure 5). Tuples of key and registration ID are stored as service references which is marked by ①. The service description itself is stored in the Chord ring using its registration ID which is indicated by ②. **Query:** The service discovery layer assembles a template consisting of the demanded attributes (cf. figure 6). It generates the key for one of the attributes and queries the Chord ring for all service references belonging to the key, e.g. ③. The two service descriptions belonging to the registration IDs D1 and 79 are looked up which is shown by ④. If the retrieved service descriptions match the rest of the template, they are returned to the application.

**Remark:** Node IDs are shortened to the two highest digits for the sake of readability.

C. Modifying and unpublishing service descriptions

Sometimes it may occur that a previously published service description has to be modified. Therefore the attributes which have changed or have become obsolete are removed from the Chord layer and attributes which are new or have changed are added to it. This is done by calculating the keys for the affected attributes and delegating the addition or removal of service references to the Chord layer. Since the content of the service references stays the same, unchanged attributes are not affected by the modification. This procedure ensures that there is only network traffic generated for changed attributes.

If a service description has to be unpublished, the service references belonging to the affected service have to be deleted. Therefore the keys of all service attributes are generated and passed to the Chord layer for removal. Further, the service description is removed from the local service description data structure.

D. Handling departure and failure of nodes

Nodes joining and leaving the P2P network as well as crashing nodes are handled by the Chord layer, transparently to the service discovery layer. But the service discovery layer has to take care of the content which is stored in the Chord layer. When a node leaves the network, the service discovery layer of that node has to ensure that all service references referring to the leaving node are removed from the underlying Chord layer.

If a node crashes, the service references of the crashing node have to be removed, too. There are three possibilities to achieve this:

- A leasing concept could be employed making every node responsible to renew the leases for its service references in a regular interval. Unfortunately, this would lead to the same number of messages as if all service descriptions would be published again, but would be repeated whenever leases have expired.
- A node would recognize that a node has crashed while querying for a service description. This is why all nodes storing possible relevant service descriptions have to be contacted directly. If a node does not respond to this request, the querying node can remove the affected service reference from the Chord layer. This solution would lead to lots of properties being stored in the Chord layer which are never used for a query.
- Service descriptions could be replicated by the publishing node to \( k \) other nodes. This could for example be accomplished by exploiting the routing information used by the Chord layer for internal replication purposes. If one of these nodes detects that a publishing node has crashed, it could initiate the removal of all service references published by the crashed node.

Though being most complex the last solution is preferable, because it ensures that service descriptions are up-to-date without producing significant traffic.
V. IMPLEMENTATION OF THE DISCOVERY SERVICE

We implemented our protocol described in the last section for testing and evaluation purposes. This implementation is written in Java 5.0 [8] and is based on TCP/IP as well as on a Java implementation of Chord. Figure 8 shows the most important classes of both the discovery and the Chord layer.

The abstraction of the concrete transport layer is done by use of proxy and endpoint classes. Every remote node in the network is locally represented by a proxy instance. An endpoint instance handles incoming network connections for the local node. Proxies representing a remote node contact the endpoint of that node by means of a TCP/IP socket. With help of these proxies and endpoints as well as the application of factories the TCP/IP protocol could easily be exchanged by another transport protocol (e.g. Java RMI).

The functionality of the Chord overlay network is made accessible by a single interface called Chord. It provides methods to join, create, and leave a Chord overlay network. In order to participate in a Chord overlay network, a user has to provide a URL which contains the protocol to be used and the location of the node. Furthermore, a URL of another node in the overlay network, the so-called bootstrap node, has to be provided to connect to an existing Chord overlay network. If this URL is omitted, a new overlay network is created by the local node. Additionally, Chord provides methods to store, delete, and retrieve key-value pairs. Keys are instances of the class Key. Values are represented by classes implementing the interface java.io.Serializable, so that they can be transferred over the network.

There are two mechanisms for representing service descriptions for use by the service discovery layer:

1) An instance of ServiceDescription can be used for representing a service description. This class offers a method to set attributes by name represented by a String and the value of the attribute.

2) Arbitrary classes can be used of which the attributes are extracted by using the Java Reflection API. Those classes have to provide one getter method for each attribute they contain. These methods must be named get concatenated with the name of the attribute.

Types of attribute values are restricted to the wrapper classes of primitive Java types, e.g. Integer, common Java value types, e.g. String, data structures implementing the java.util.List interface, or classes which themselves possess get methods containing more service attributes. Internally, all attributes are converted to instances of the class Property. They contain name and value of an attribute as well as methods to determine the hash key for storage in the Chord distributed hashtable.

For lookup purposes a data structure is provided for representing a query template. It is called Template and is used similar to ServiceDescription. Moreover it is possible to reuse the application-specific classes used for service description as template classes by means of Java Reflection. Therefore only those attributes which are relevant for the lookup must be set while the others are assigned to the wildcard value null.

The functionality of the discovery service is provided to an application by the Registry interface. It contains methods to lookup, register, and unregister service descriptions by using the data structures ServiceDescription, Template, and application-specific classes.
VI. RELATED WORK

Recently, some work has been done on enabling searching capabilities in P2P systems based on distributed hashtables which goes beyond looking up keys. [9] proposes a way to apply inverted indices to use for keywords in file sharing applications. [10] extends this model by adding mechanisms to improve query efficiency in such a system, namely query ordering, bloom filters, popularity information, and truncated results. [7] introduces a keyword dictionary and improves query efficiency by so-called keyword fusion. All this work has been applied to the search for multimedia data in file sharing systems rather than for service discovery in service-oriented environments.

In our approach we assume service providers and consumers to have a common schema for describing services. That means that service types and their attributes are known before using the registry service. In contrast to this, [11] proposes a method to add semantics into registry services, for example by returning similar services which do not exactly match the queried service description, by using ontologies.

VII. CONCLUSION

In this paper we proposed a protocol for decentralized service discovery with guarantees. We used a P2P network based on a distributed hashtable that provides a structured overlay network in order to avoid flooding the whole network. Service descriptions are decomposed into portions which can be efficiently distributed and retrieved. The implementation of the discovery and Chord layer will be made available over the internet as open source after they have been tested and evaluated more extensively.

Currently we are working on simulating our protocol. Simulation is one possible method to evaluate its performance in comparison to other protocols, i.e. those which are not based on structured P2P networks such as distributed hashtables. A protocol suitable for comparison must fulfill the same functional requirements as we stated above. This includes publishing service descriptions as well as sending queries containing service templates which are guaranteed to be answered by all matching services in the network. Therefore we assume a Gnutella-like network in which service descriptions are stored locally at each node and queries are sent through the network by flooding in order to achieve the same guarantees as our protocol does. The definitive disadvantage of this approach is that flooding is inherently inefficient. But an advantage of the Gnutella-like protocol may be that one query message can contain a template with an arbitrary number of attributes of a potential service. Usage of the discovery service does not depend on one of these two protocols. Therefore assumptions must be made on network size and dynamics, e.g. arrival and uptime of nodes, and on service usage, e.g. frequency of publishing and querying of services and number and kind of attributes contained in service descriptions and query templates. The data to be measured and compared in the simulation can be divided into quality measures, e.g. response time of queries, and impacts for the nodes, e.g. amount of storage, number of open connections, and traffic volume. Since the setting is the same for both protocols we expect our protocol to outperform the Gnutella-like approach.

Moreover a prototype for service discovery in a FIPA-complying agent platform is under development. We also aim to improve efficiency of our protocol in the future and intend to incorporate means like a frequency dictionary of service attributes in the style of the fusion dictionary proposed in [7] in order to decrease traffic of multi-attribute queries. Further we intend to support wildcard and range queries in the future. At last, security issues have to be taken into consideration, since any adversary would be able to add, modify, or delete service descriptions at will.

REFERENCES