# 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
Conference: PDPTA'05; Presenting author (if accepted): Karsten Loesing

***Abstract*— This paper proposes a protocol for decentralized service discovery with guarantees. We use a peer-to-peer network based on the distributed hashtable Chord 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. We propose a way to evaluate our protocol by running simulations in comparison with a straightforward way of achieving the same goal in an unstructured, Gnutella-like network.**

**KEYWORDS: SERVICE DISCOVERY, PEER-TO-PEER, DISTRIBUTED HASHTABLE**

## I. INTRODUCTION

In the 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 by another peer, 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 first publishing service descriptions at so-called registries and second 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 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 their 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-conforming agent platforms [2] the so-called directory facilitators can be federated, so that query requests are forwarded to each other with a given TTL (time to live). As another example, UDDI allows connection of multiple registries hierarchically or in a peer-like fashion which is called registry affiliation [3]. Those approaches to connect 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 the 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.

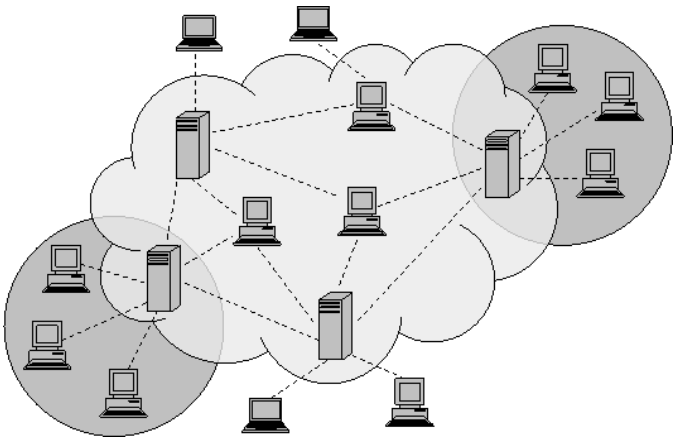

Fig. 1. Network consisting of nodes providing the registry service (inside cloud) and those that only make use of it (outside cloud). Nodes in circles stand for nodes belonging to administrative units.

## 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, that service providers want to publish, consist of (name, value) pairs describing the attributes of services to publish. Values of service descriptions can 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 as a service description. A discovery service must therefore allow publishing, modifying, and unpublishing tree-like complex data structures. There must not be any restrictions 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 directly impose requirements on the underlying P2P overlay network as the guarantee to find all available services does, as well. The overlay network 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 goal 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 in a way that queries can be step-wisely routed to the 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 that any query can 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 argument for our approach to use a structured P2P network is 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.

## 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 2.

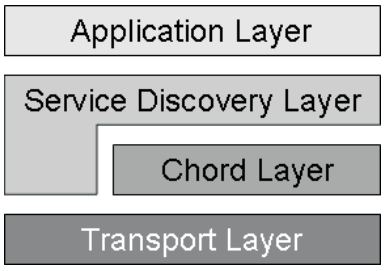


Fig. 2. Architecture of the P2P discovery service.

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 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 3).

```
( type = ticketservice;
  url = ( protocol = http;
          host = 81.200.194.40;
          port = 8080;);
  owner = DB;
  languages = {german,
               english,
               french};
)
```

Fig. 3. Example of a service description.

For each service description to be published a *registration ID* is created which can later 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. By this 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 4 shows an

example of the keys generated for the service description of figure 3.

```
"type.ticketservice"       ->  ADBCB6...17
"url.protocol.http"        ->  801D4F...F0
"url.host.81.200.194.40"   ->  922E25...59
"url.port.8080"            ->  9F5A36...23
"owner.DB"                 ->  71A8D1...30
"languages.german"         ->  A36923...42
"languages.english"        ->  BE2CD9...AD
"languages.french"         ->  B81C8B...D9
```

Fig. 4. Example of keys for a service description.

After service references have been stored within the Chord layer for each attribute, the service description itself is stored in a local data structure of the service discovery layer. The generated 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. As with publishing of services the attribute types may be primitive or complex types as well as sets of these. In order to permit multi-attribute queries, *templates* are used which incorporate all attributes belonging to one query.

Querying for all services matching a given template is done by choosing one of the attributes by random, calculating the hash key for it, 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.

From this description follows that 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.

### 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 the 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 achieved 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 this solution is preferable, because it ensures that service descriptions are up-to-date without producing significant traffic.

## V. Evaluation

Simulation of our protocol is one possible means to evaluate performance of it in comparison to other protocols, i.e. those which are not based on structured P2P networks like distributed hashtables. A protocol suitable for comparison must fulfill the same functional requirements as we stated above. That 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.

As simulation environment ns-2 [8] may be used which is a discrete event simulator working on packet level. In addition to this, GnutellaSim [9] might be useful which is an open-source library for simulation of P2P protocols and can be run with ns-2. The Gnutella protocol provided by GnutellaSim has to be modified to achieve unbounded flooding which is required for the comparison. Further an implementation of our protocol as well as a Chord layer has to be added to the library.

## 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. [10] proposes a way to apply inverted indices to use for keywords in file sharing applications. [11] 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 searching 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 [12] proposes a means 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. We proposed a way to simulate our protocol by comparing it with a straightforward way of achieving the same goal in an unstructured network.

## VIII. Current and future Work

Currently we are working on simulating our protocol according to the assumptions made in section V. Moreover a prototype for service discovery in a FIPA-conforming agent platform is under development. We also aim to improve efficiency of our protocol in the future and aim to incorporate means like a frequency dictionary of service attributes in the style of fusion dictionary [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


[1] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd ed. Addison-Wesley, 2001.

[2] *FIPA Agent Management Specification*, Foundation for Intelligent Physical Agents (FIPA), March 2004.

[3] *Introduction to UDDI: Important Features and Functional Concepts*, Organization for the Advancement of Structured Information Standards (OASIS), October 2004.

[4] *The Gnutella Protocol Specification v0.4*. [Online]. Available: http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[5] A. Oram, Ed., *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly, March 2001.

[6] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[7] L. Liu, K. D. Ryu, and K.-W. Lee, "Keyword fusion to support efficient keyword-based search in peer-to-peer file sharing," in *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, April 2004, pp. 269–276.

[8] [Online]. Available: http://www.isi.edu/nsnam/ns/

[9] Q. He, M. Ammar, G. Riley, H. Raj, and R. Fujimoto, "Mapping peer behavior to packet-level details: A framework for packet-level simulation of peer-to-peer systems," in *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems*, 2003. [Online]. Available: http://csdl.computer.org/comp/proceedings/mascots/2003/2039/00/2039toc.htm

[10] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Lecture Notes in Computer Science*, vol. 2672. Springer-Verlag GmbH, 2003, pp. 21–40.

[11] T. Lu, S. Sinha, and A. Sudan, "Panaché: A scalable distributed index for keyword search." [Online]. Available: http://www.pdos.lcs.mit.edu/6.824-2002/projects/

[12] D. Elenius and M. Ingmarsson, "Ontology-based service discovery in p2p networks," in *Proceedings of the MobiQuitous'04 Workshop on Peer-to-Peer Knowledge Management (P2PKM 2004), Boston, MA, USA, August 22, 2004*, 2004. [Online]. Available: citeseer.ist.psu.edu/711664.html