A TOTALLY DISTRIBUTED ITERATIVE SCHEME FOR WEB SERVICES ADDRESSING AND DISCOVERY

M. Milanesio, G. Ruffo and R. Schifanella
Department of Computer Science
University of Turin
Torino, Italy
{milane, ruffo, schifane}@di.unito.it

ABSTRACT
Both web services framework and peer-to-peer networks provide a discovery process, but as current P2P systems focus more on the discovery of content in the form of common files (possibly associated with metadata), a centralized UDDI registry serves content in the form of metadata describing web services. Thus, the intersection between P2P and Web services is clear, so that it recently gained lots of interest: there is the need of finding a way to exploit the high potential of web services without paying in terms of scalability or single-point-of-failures. Our aim in this paper is two-fold: on the one hand, we want to present a high level protocol for the addressing and the discovery of web services that is totally decentralized and based on a structured P2P topology. On the other hand, we introduce a way to solve the exact match key-based routing problem, that afflicts structured P2P systems. This model guarantees that all addressed web services will be discovered in a logarithmic number of hops, as it is based on a Distributed Hash Table. One of the key innovation is the possibility of customizing the categorization of web services using tags, which are becoming very common in the so called Web 2.0.

KEY WORDS
Distributed Computing, Web-based Systems, Web Services, Addressing, Discovery, Distributed Hash Tables

1 Introduction
In his recent essay “The Wealth of Networks”\(^1\), Yochai Benkler explains how many phenomena exploited by the latest generation of distributed applications are re-shaping our markets, our way of using technology, and ultimately our life. In particular, we are assisting to a Commons-based peer production trend, that is at the basis of phenomena like the so called collective intelligence and social production, that push for new disruptive technological frameworks characterized by a better scalability, a higher decentralization of services, and a loosely structured organization of information.

Moreover, the responsibility of operations like publication, management and provisioning of services or digital events (e.g., audio/video broadcasting) is rapidly moving from few and well-known entities and nodes to the “Long Tail” of Internet users. These former passive consumers of the technology are increasingly enjoying the wealth of the networks, becoming active parts of the distribution processes (e.g., Web 2.0, Peer-to-Peer networks, social networks, and so on). Hence, user generated taxonomies (e.g., folksonomy) are generally preferred to complex (even if more expressive) description languages and composite ontologies. Finally, events and services are characterized by a high level of transience: in such a context, addressing is a very difficult task, especially if the publisher wishes her services be reached by search engines, that require - at least - a permanent access from the Domain Name System (DNS).

Nevertheless, the highly decentralized and dynamic nature of Web Services fits badly with (1) a central registry repository, like UDDI, that can represent a single-point-of-failure, and (2) the actual Web search engines, that address static points of access (i.e., public IP addresses and/or DNS symbolic names).

In this paper, we face the problem of spreading the UDDI addressing service in a totally distributed manner, presenting I-KoD, a fully decentralized framework for publishing, addressing and discovery generic Web Services on a DHT-based system. From the providers side, publishers of a given services make use of a straightforward description language, based on open-ended labels organized in two levels. From the end users side, consumers can access to services by means of an old fashioned Yahoo-like style, iterating a key-to-key search scheme that brings the user from a general category to a specific service.

2 Related Works
Distributed Hash Tables (DHTs, [1][2][3][4]) are a class of distributed systems that partition the ownership of a set of identifiers among available nodes in a fixed topology, routing messages efficiently to the unique owner of any given key. All the primitives provided by a DHT can be summarized by the functions $\text{lookup}(key)$ and $\text{put}(key, value)$. The first returns the identifier of the node responsible for the key, and the second gives the responsibility of value to the node responsible for key.

\(^{1}\)http://www.benkler.org/
Storage on the DHTs can be either direct (i.e., content is mapped in responsible node(s); good for small amount of data) or indirect (i.e., meta information is stored instead of content; more feasible). Given a message and a key, the DHT routes the message to the node with the node-ID that is numerically closest to the key. The message routing has a cost that is logarithmic in the size of the network: each node maintains a routing table that contains information on a logarithmic subset of the entire system, and each message is forwarded through a logarithmic number of hops to its destination. This feature grants scalability to the structured P2P systems.

Several issues concern DHTs: from consistency [5], to latency in routing and location-awareness [6]. Two major issues are the bootstrapping and the key-based routing. Bootstrapping relies on known nodes, and this is a problem when a node joins the system for the first time, or when the known nodes are unavailable, and the key based routing can be simple when keys are known in advance, but this cannot be assumed at an application level. As a consequence, many solutions have been proposed: from the insertion of meta-information and meta-keys to the parsification of the query string (such as in eMule with Kademlia support). The impoverished query language due to the key based routing (i.e., no support for complex queries) is focused in [7], where authors give a entity-relational view to the P2P search mechanism. The query mechanism in [8] relies on indexes, stored and distributed across the nodes of the network: given a broad query, a user can obtain additional information about the data items that match the original query, recursively querying the DHT layer. The whole system creates multiple indexes, organized hierarchically, which permit users to access data in many different ways. Indexes are distributed across the nodes of the network and contain key-to-key (or query-to-query) mappings.

Web Services [9] are interoperable software components used in application integration and development: a web service is located and addressed through a protocol (UDDI) and described via a specific language (WSDL), and it involves three actors: the provider, the broker and the client. One of the main obstacle affecting the web service discovery mechanism is heterogeneity between services [10]. In the cited work it’s suggested a higher level approach in addressing different web services in order to overwhelm this variety, either technological, ontological or pragmatic. Another issue can be found in the centralized (even if replicated) UDDI registry that lacks of scalability and introduces a single point of failure in the system. Many distributed approaches have been proposed, in order to increase flexibility. The P2P system in [11] supports complex queries containing partial keywords and wildcards, and it guarantees that all existing data elements matching a query will be found with bounded costs in terms of number of messages and number of involved nodes. The whole system is based on a DHT and, for preserving locality purposes, it organizes keys (mapping data elements) in a multidimensional keyword space exploiting the lexicographical distance of data elements. In [12], the problem of scalability in the search mechanism is addressed imposing a deterministic shape on P2P networks, exploiting the so-called Semantic Web Services, an ontology based combination of Semantic Web and web services: authors propose a hypercube graph topology, which allows efficient broadcast and search, using global ontologies to partition the network topology into concept clusters that can be queried specifically. Ontologies (i.e., the explicit specifications of the conceptualization of domains) are also the core of [13]: in order to effortlessly discover, consume and provide services as they become available in a dynamic network, the so-called ubiquitous computing is introduced, organizing services in a logical manner. In the end, the work in [14] presents a distributed ontology repository that uses the DHT to store the ontologies, represented in OWL².

Our work starts from these experiences, lightening the framework both in the DHT usage and in the Web Service discovery mechanism. We do not use ontologies like [13, 14], nor lexicographic distance as in [11]. We propose a flat indexing space based on tags, without an entity-relational view of the network such [7].

3 Protocol Description

In this section, we briefly present I-KoD (Iterative Key-based routing on Dhts), a high level application protocol for web services addressing and discovery. We are not taking in consideration the structure of the WSDL, neither how to invoke a specific web service: what we want to score is the decentralization of the UDDI registry, by means of a structured P2P system that plays the role of the registry. Our system is based on top of a DHT-based structured overlay network used for routing messages and a storage layer, on top of the DHT, to store resources (i.e., WSDL files, as explained later). The key components of our layered system are: (a) a routing infrastructure, i.e., the DHT; (b) a storage layer; (c) an application level iterative scheme, for the insertion and the retrieval of the indexed resources; (d) a set of web services, provided and requested by users.

For sake of simplicity, the reader can refer to the Pastry ring with the Past storage layer built on it. We will focus on the component (c).

Each DHT offers a set of functions for building correct identifiers for resource indexing. For simplicity, reader can think of a single function defined as \( I : \{0,1\}^* \rightarrow \{0,1\}^h \) (where \( h \) is the length of the current DHT identifiers, e.g., \( h = 128 \) if Pastry DHT is used). Moreover, we define two high level functions, namely insert and get (based on the DHT layer primitives lookup and put), as follows:

²http://www.w3.org/TR/owl-features/
In order to find the node \( n \), append \( \text{value} \) to the list of tags referred by (the tag indexed by) \( \text{key} \).

**get(key):** A \( \text{lookup(key)} \) is performed for finding node \( n \). At an application level, \( n \) will return the value associated to \( \text{key} \). The value can be an element or a list of elements.

The system provides a mapping for web services spread among the peers and it is characterized by a hierarchical scheme in which the services are organized in a straightforward user-defined semantic way rather than e.g., through an ontology [14]. Through this scheme, for each inserted web service, the WSDL location can be reached, and thus the web service itself can be invoked. A possible hierarchical indexing scheme is shown in Figure 1.

In order to establish the hierarchical scheme, web services are organized into classes (e.g., Entertainment, News, Health, ...). Each class is divided into types (e.g., in the Entertainment class, it is possible to find web services of type TvOnDemand, Radio, Sports, ...). Finally, there are several services of the same type (e.g., more than one streaming service concerning Sports).

As shown in Figure 1, a web service is completely and univocally identified by the path followed from the root to the leaf of the tree: a web service has its own class, type, and provider. It’s important to underline that, at this moment, we are not facing WSDL-related issues (i.e., parameters, IP address of the server that hosts the web service and so on). The following resources take part in our protocol, either as keys to be spread over the DHT, either at the application level, as tags to be used for generating the tree in Figure 1.

- \( \text{init} = \text{"root - list"} \). When bootstrapping, each node must contact the responsible for the identifier indexing this high level resource, containing the list of available tags for classes actually available in the systems.

- A set of tags \( C = \{ c_0, c_1, \ldots, c_l \} \) containing \( l \) classes, \( c_i \) is the \( i^{th} \) (tag for) class \( i \) (e.g., Entertainment);

- \( \forall i, \exists m > 0 \text{ such that } T_i = \{ t_0, t_1, \ldots, t_m \} \) is the set of tags for types associated to class \( c_i, t_j \) is the \( j^{th} \) type of the \( i^{th} \) class (e.g., TvOnDemand);

- \( \forall j, \exists n > 0 \text{ such that } S_j = \{ s_0, s_1, \ldots, s_n \} \) is the set of tags of inserted web services for type \( t_j \) (e.g., MyTv), each one of them is described through a WSDL file (e.g., for the \( k^{th} \) service, \( \text{wsdl}_k \));

- A set \( P = \{ p_0, p_1, \ldots, p_r \} \) of providers. For each provider \( h, p_h \) can be either her public key or other identification schemes (e.g., OpenId url³).

With the assumptions on \( m \) and \( n \) we want to underline the fact that the existence of a class of web services suppose necessarily the existence of at least one type, and so on, recursively until the leaf.

While tags (members of sets \( C, T = \bigcup T_i \) and \( S = \bigcup S_j \)) are human readable strings, for each one of them a DHT identifier has to be computed, starting from the complete path from the root (thus allowing different services with the same name under different categories). For example, service tag \( s_0 = \text{MyTv} \) will have an identifier build through the application of function \( I \) with argument “init[Entertainment][TvOnDemand][MyTv].”

### 3.1 Inserting a Web Service

In order to explain the publishing of a web service, we suppose to start from an empty system, i.e., a system with \( \Omega \) nodes in which no web service has been inserted yet, and thus no classes and no types were given (at time \( t = 0 \) the tree in Figure 1 is empty). A service provider \( p_i \in P \) (identified by \( id_{p_i} \) on the DHT) builds its web service, tagged \( s_i \), and describes it through a WSDL file (namely \( \text{wsdl}_i \)), with the associated identifier \( id_{\text{wsdl}_i} \), computed through \( I \): \( p_i \) decides that suitable tags for \( s_i \) could be class \( c_i \) and type \( t_i \) (for instance, let’s assume \( c_i = \text{Entertainment}, t_i = \text{TvOnDemand} \) and \( s_i = \text{MyTv} \)). In order to publish \( s_i \) on the system, \( p_i \) has to perform the following steps:

**Step 1.** \( \text{get(id_{init})} \). \( p_i \) will get the list of available classes (empty, in our example) from which she can choose a suitable tag for class.

**Step 2.** Performing three times the \( I \) function with parameters “init||\( c_i \)”,”init||\( c_i||t_i \)” and “init||\( c_i||t_i||s_i \)” provider \( p_i \) will compute the three DHT identifiers \( id_{c_i}, id_{t_i} \) and \( id_{s_i} \) respectively, which represent the keys the tags will be indexed with on the DHT. As keys are fairly distributed on the DHT, we can suppose three different nodes to be responsible for those keys (e.g., \( N_{c_i}, N_{t_i} \) and \( N_{s_i} \)).

³http://openid.net/
Step 3. \( \text{insert}(i_d^s, [id_s^t, \text{TvOnDemand}]) \): \( p_i \) contacts the application layer of node \( N_{c_i} \), inserting the mapping \([id_s^t, \text{TvOnDemand}]\).

Step 4. \( \text{insert}(i_d^s, [id_s^s, \text{MyTv}]) \) will cause the insertion of the entry \([id_s^s, \text{MyTv}]\) on node \( N_{t_i} \).

Step 5. \( \text{insert}(i_d^s, [id_p^s, id_{wsdl}^s]) \) is the insertion of the identifier of the node responsible for the WSDL on the node responsible for the tag \( s_i \).

Step 6. Finally, on the storage layer of the node responsible for the identifier \( id_{wsdl}^s \), the WSDL file \((wsdl^s)\) of the service has to be inserted.

As providers insert their services, the \text{insert} function will cause either the insertion of a new tag (as in the above description) or simply a new type (service) added on the node responsible for the higher level tag. Considering the number of operations on the DHT layer, in terms of primitives \( \text{lookup}( \text{key} ) \) and \( \text{put}( \text{key}, \text{value} ) \) (both used by the \text{insert}), a distinction should be made whether the provider finds suitable tags for his web service or not. In both cases, anyway, the provider has to perform at least one DHT \text{lookup} (if it does not find any suitable class tag). A second \text{lookup} has to be done in case a suitable class is given but not a suitable type.

Furthermore, the provider has to perform a DHT \text{put} operation (Step 6) for inserting its WSDL on the storage layer of the DHT. If the provider needs to update a web service, if the changes concern only the web service itself and not its tags, what has to be done are Step 5 and Step 6. In case of deletion, the removal of the mapping in Step 6 will cause, as the system runs, the removal of the paths that lead to that resource.

In the end, a further refinement could be done in the identifier building function. In order to allow different services with the same name but under different classes or types, top level tags could be combined with the XOR operator together with the service tag.

3.2 Searching for a Web Service

A simple interface is given to the user. For simplicity, let’s assume \( l \) classes and \( m \) types for each class, \( n \) services for each type. Once joined the system, the available classes of web services are given, and thus user can choose the preferred class. More specifically, the joining of the system returns to the user a list of classes \( c = \{c_0, c_1, ..., c_l\} \), from which she can choose the desired \( c_i \), and thus compute the \( i_d^s \) identifier, as in Step 1 of the insertion procedure. All the identifiers mentioned in the following will be used in the DHT \text{lookup} function, as for the insertion case, for finding the responsible node(s). The iterative scheme evolves with the following steps.

Step 1. \( \text{get}(i_d^s) \) returns a list \( t = \{t_0, t_1, ..., t_m\} \) containing all (the tags for) types available for that class. User chooses \( t_j \), and computes \( i_d^s \).

Step 2. \( \text{get}(i_d^s) \) returns a list \( s = \{s_0, s_1, ..., s_n\} \) containing all the services of that type, from which \( s_k \) can be chosen (computing \( i_d^s \)).

Step 3. \( \text{get}(i_d^s) \) returns the identifier of the node responsible for the WSDL (or directly the URI) of the desired web service (see for reference Step 5 in Section 3.1)

Step 4. Using this last information the user can contact the node responsible for the WSDL, download it from the storage layer and thus invoke the web service.

It’s easy to notice that, in the worst case (i.e., user has no information at all on the status of the system) the whole procedure will take up to four \text{lookup} operations on the DHT layer.

4 Discussion

I-KoD does not use a typical DHT key-to-value mapping, but rather a key-to-key one. Locally, a host can store a list of indexes that is fills as the system runs, and this will permit to some user to search in a different way from the one proposed in Figure 1: the distinct paths that can lead to a same resource will travel across different indexes (i.e., different keys). For example, suppose (with reference to Figure 1) a user searching for service \text{MyTv}. If the user has no information at all about the inserted services, its path to the service will follow the default steps described before. If the user knows of some provider that offer the type of services known as \text{MyTv}, she will follow another path to reach the same \text{node}: looking for the provider identifier, it will retrieve all the services offered from that service provider, and thus the service \text{MyTv}, among others, without following the whole path from the root to the leaf. Finally, it is important to underline that when a service is updated, or a new one is inserted, the new mapping has to be inserted and indexed. This will not concern higher level brokers, i.e., nodes responsible for more generic keys, as this information will sooner or later reach all nodes in the alternative paths that lead to the resource. What we have is a sort of Domain Name System (DNS) that works, instead of names, with dynamic and transient nodes and services. Top level servers are responsible for top level keys, while intermediate keys are held by intermediate nodes. An application level caching must be provided in order not to overload intermediate nodes, that are responsible for a huge number of keys.

Another aspect is the tagging of web services. Tagging-based systems enable users to add tags (i.e., freely chosen keywords) to web resources for categorizing purposes. Tagging is not only an individual process of categorization, but implicitly it is also a social process of indexing, a social process of knowledge construction. Users share their resources with their tags, generating an aggregated tag-index so-called folksonomy\(^4\). There are two

\(^4\)One-word neologism from taxonomy and folk.
main approaches of information access in a folksonomy: Information Filtering (IF) and Information Retrieval (IR) [15]. We exploit the second one in our approach. User can actively query the system, but, instead of inserting keywords for searching, she will follow a given classification (see Figure 1), that can be updated and extended by service providers themselves, with the insertions of new tags.

It is important to underline that this folksonomy could affect the growth of the ariety in the tree of Figure 1, as users join/use the system and insert new tags for classes or types of web services and cause the size of top level lists to grow a lot. We can reasonably suppose this not to happen: in real folksonomy-based systems the tag popularity follows a power-law distribution $P(k) \sim k^{-\gamma}$, where the constant $\gamma$ is called the exponent of the power law. Roughly speaking, wrt our system, there are very few hot tags (i.e., a small number of very long lists), while most of tags are poorly used. We studied the real case of Last.fm\(^5\), in which, starting from the equations in [16], we calculated the exponent $\gamma = 2.15$ and the error $\alpha = 0, 115$ of the tag distribution in that system. Starting from this, we can suppose that normally a user would follow and use the already given tags, but, if the user will add a new tag, this one will be located on the tail of the distribution, and thus this should have a minimum effect on the overall functionality of the system, as the number of tags will tend to be (with high probability) bounded.

Furthermore, the load on each node responsible for the hot keys will not affect the distribution of the identifiers on every node. With reference to Figure 1 suppose node $N_{ci}$ responsible for the key indexing Entertainment (e.g., $c_i$, that should be a highly populated family of web services). When a user will perform $get(c_i)$, $N_{ci}$ will respond with a list of type tags $[ids_i, ts_i]$ pairs associated, at an application level, to class $c_i$. Starting from the following example based on the Pastry DHT, it is important to notice that, wrt the amount of used disk space on each node, the state of the node will not be affected so much by the gathering of the identifiers in those lists. With the $\gamma$ and $\alpha$ coefficients calculated as told before, we computed analytically the average load on the three most loaded nodes (i.e., the nodes responsible for the keys indexing the three most popular type tags $t_i, i \in [0, 2]$): results are summarized in Table 1, where the list size (both in number of elements and in KB) is presented for the three most overloaded nodes.

This size can be further reduced (on each node) with an heuristic approach, as follows. In Pastry, each node $r$ maintains a leaf set, namely $LS_r$. Each leaf set has a constant $d$ dimension that is $2 \times 2^b$, with $b$ fixed as a configuration parameter (usually $b = 2$). Each node is responsible for a part of the DHT, namely $KV$. For replication and caching purpose, at a storage layer the keys are spread in the leaf set, and we can think of $KV = KV_1 \cup KV_2 \cup \ldots \cup KV_b$; that is, the set of all the $[key, value]$ pairs the node $r$ is responsible for is divided in the leaf set. We can build the set of the nodes contained in the leaf set so that $KV_i \cap KV_j \neq \emptyset; \forall i, j \in LS_r$, so we can investigate the dimension of $KV_i$, considering a probability $p$ for each node in the leaf set to be responsible for some keys of the other nodes in the leaf set.

$$|KV_i| = \left\lceil \frac{|KV|}{d} + p \cdot (|KV| - \frac{|KV|}{d}) \right\rceil$$

We emulated\(^7\) this scenario, and, in Figure 2, as the size of the system and the number of web services increase, the results are presented. The variation of the parameter $p$ gives us a snapshot of the load on the leafset of the most loaded node (i.e., the responsible for the most-used-tag).

When $p = 0$, each $KV_i$ is disjoint from the others, and $|KV| = \lceil |KV|/d \rceil$. In this case the system presents

\(^{5}\)http://www.lastfm.it/charts/music/tag/

\(^{6}\) $KV_i$ is the subset of $[key, value]$ pairs for which the node $n_i \in LS_r$

\(^{7}\)We used OverlayWeaver, available at http://overlayweaver.sourceforge.net/

---

Table 1. Lists size growth wrt classes and types.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
<th>First list size (# - KB)</th>
<th>Second list size (# - KB)</th>
<th>Third list size (# - KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>200</td>
<td>44 - 3.5</td>
<td>18 - 1.5</td>
<td>12 - 0.9</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>60 - 4.8</td>
<td>24 - 1.9</td>
<td>16 - 1.2</td>
</tr>
<tr>
<td>75</td>
<td>50</td>
<td>80 - 6.4</td>
<td>30 - 2.4</td>
<td>19 - 1.5</td>
</tr>
<tr>
<td>100</td>
<td>600</td>
<td>108 - 8.7</td>
<td>41 - 3.3</td>
<td>26 - 2.1</td>
</tr>
<tr>
<td>125</td>
<td>750</td>
<td>130 - 10.5</td>
<td>51 - 4.0</td>
<td>33 - 2.6</td>
</tr>
<tr>
<td>150</td>
<td>900</td>
<td>116 - 9.2</td>
<td>42 - 3.3</td>
<td>26 - 2.1</td>
</tr>
<tr>
<td>200</td>
<td>1200</td>
<td>146 - 11.7</td>
<td>56 - 4.4</td>
<td>36 - 2.9</td>
</tr>
<tr>
<td>300</td>
<td>1500</td>
<td>185 - 14.8</td>
<td>69 - 5.5</td>
<td>43 - 3.5</td>
</tr>
<tr>
<td>400</td>
<td>200</td>
<td>139 - 11.1</td>
<td>52 - 4.1</td>
<td>33 - 2.6</td>
</tr>
<tr>
<td>500</td>
<td>250</td>
<td>180 - 14.4</td>
<td>70 - 5.6</td>
<td>44 - 3.5</td>
</tr>
<tr>
<td>600</td>
<td>300</td>
<td>230 - 18.4</td>
<td>87 - 6.9</td>
<td>55 - 4.4</td>
</tr>
<tr>
<td>700</td>
<td>400</td>
<td>180 - 14.4</td>
<td>63 - 5.1</td>
<td>39 - 3.1</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>214 - 17.1</td>
<td>83 - 6.6</td>
<td>53 - 4.2</td>
</tr>
<tr>
<td>600</td>
<td>600</td>
<td>280 - 22.4</td>
<td>104 - 8.3</td>
<td>66 - 5.3</td>
</tr>
<tr>
<td>700</td>
<td>1000</td>
<td>194 - 15.5</td>
<td>75 - 5.9</td>
<td>46 - 3.6</td>
</tr>
<tr>
<td>800</td>
<td>1200</td>
<td>256 - 20.5</td>
<td>97 - 7.8</td>
<td>61 - 4.8</td>
</tr>
<tr>
<td>900</td>
<td>200</td>
<td>320 - 25.6</td>
<td>117 - 9.3</td>
<td>74 - 5.9</td>
</tr>
</tbody>
</table>

Figure 2. Number of pairs on the most loaded node, as the system grow.
a minimal replication degree: the load on each node is minimal, but some problems may occur in case of failure, as it would not be possible to re-build the whole $KV$. When $p = 1$, each node in the leaf set is responsible for every key the root node is responsible for, $|KV_r| = |KV|$. In this case the system presents a maximum degree of replication, as well as a huge overhead on every node in the leaf set.

We can see from the Figure 2 that having $0.2 \leq p \leq 0.3$ can lead to a load decrease for node $r$, together with a good replication degree. Our analysis is based on the fact that splitting the keys set a node $r$ is responsible for into the nodes in its leaf set grants reliability through the replication of the mappings, and scalability through load balancing in the leaf set. It is also clear that a small overhead is introduced in this mechanism, during the lookup phase: the reason is that the leaf set has to be queried in order to re-build the set of keys held by $r$ (i.e., the set $r$ is responsible for). This is not a real problem, thanks to the lookup properties of Pastry (i.e., prefix-based approach) and to the fixed dimension of the leaf set. The scalability of the lookup process is deeply discussed in [17]: the number of hops needed to retrieve a resource, logarithmic in the number of nodes, takes advantages from the caching of the resources at the application level (i.e., the caching of the WSDLs).

5 Conclusion and Future Work

In this paper, we introduced I-KoD, a fully decentralized addressing and discovery framework based on Distributed Hash Table. I-KoD allows a straightforward description language based on open-ended tags that are used for publishing the services and for letting the end users search the system using an iterative scheme. We implemented a prototype for emulative purposes using Pastry; nevertheless, we are implementing also a Kademia based and a Chord based version, in order to perform comparative analysis. In fact, from an applicative point of view, an important task is to minimize the overhead of lookup messages; although DHTs behave quite well, we wish to optimize the framework before engineering the middleware.

Acknowledgements

This work has been partially supported by the Italian Ministry for University and Research (MIUR), within the framework of the “PROFILES” project (PRIN).

References


[17] A. Rowstron and P.Druschel, Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility, Proc. of SOSP 2001, 188-201.