A Peer-to-Peer Approach to Web Service Discovery

Article *in* World Wide Web · July 2003

DOI: 10.1023/B:WWWJ.0000017210.55153.3d · Source: CiteSeer

2 authors:

Cristina Schmidt
Universidade de Mogi das Cruzes

Manish Parashar
University of Utah

A Peer-to-Peer Approach to Web Service Discovery *

Cristina Schmidt and Manish Parashar
The Applied Software Systems Laboratory
Department of Electrical and Computer Engineering, Rutgers University
94 Bret Road, Piscataway, NJ 08854, Tel: (732) 445-5388, Fax: (732) 445-0593
{*cristins,parashar*}*@caip.rutgers.edu*

Abstract

Web Services has emerged as a dominant paradigm for constructing and composing distributed business applications and enabling entreprise-wide interoperability. A critical factor to the overall utility of Web Services is a scalable, flexible and robust discover mechanism. This paper presents a peer-to-peer (P2P) indexing system and associated P2P storage that supports large-scale, decentralized, real-time search capabilities. The presented system supports complex queries containing partial keywords and wildcards. Furthermore, 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 nodes involved. The key inovation is a dimension reducing indexing scheme that effectively maps the multidimensional information space to physical peers. The design and an experimental evaluation of the system are presented.

1 Introduction

Web Services are emerging as a dominant paradigm for distributed computing in industry as well as academia (e.g. the Open Grid Services Architecture standard for the "Web Services Grid" [15]). Web Services are enterprise applications that exchange data, share tasks, and automate processes over the Internet. They are designed to enable applications to communicate directly and exchange data, regardless of language, platform and location. A typical Web Service architecture consists of three entities: service providers that create and publish Web Services, service brokers that maintain a registry of published services and support their discovery, and service requesters that search the service broker's registries.

Web Service registries store information describing the Web Services produced by the service providers, and can be queried by the service requesters. These registries are critical to the utlimate utility of the Web Services and must support scalable, flexible and robust discovery mechanisms. Registries have traditionally had a centralized architecture (e.g. UDDI [16]) consisting of mutiple repositories that synchronize periodically. However as the number of Web Services grows and become more dynamic, such a centralized approach quickly becomes impractical. As a result, there are a number of decentralized approaches that have been proposed. These systems (e.g. [9, 10]) build on P2P technologies and ontologies to publish and search for Web Services descriptions.

In this paper we present a methodology for building dynamic, scalable, decentralized registries with real-time and flexible serach capabilites, to support Web Service discovery. Web Services can be described in different ways, according to existing standards (e.g. WSDL, ontologies), and can be caracterized by a set of keywords. We use these keywords to index the Web Service description files, and store the index at peers in the P2P system using a distributed hash table (DHT) approach. The proposed P2P system supports complex queries containing partial keywords and wildcards. Furthermore, 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 nodes involved. The key innovation of our approach is a dimension reducing indexing scheme that effectively maps the multidimensional information space describing the Web Service to a one dimensional index space that is mapped on to the physical peers.

The overall architecture of the presented system is a DHT, similar to typical data lookup systems [6, 11]. The key difference is in the way we map data elements[1] to the index space. In existing systems, this is done using consistent

*The work presented in this paper was supported in part by the National Science Foundation via grant numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), and by DOE ASCI/ASAP (Caltech) via grant number PC295251.

[1] We will use the term 'data element' to represent a piece of information that is indexed and can be discovered. A data element can be an XML file describing a Web Service.

hashing to uniformly map data element identifiers to indices. As a result, data elements are randomly distributed across peers without any notion of locality. Our approach attempts to preserve locality while mapping the data elements to the index space. In our system, all data elements are described using a sequence of keywords. These keywords form a multidimensional keyword space where the keywords are the coordinates and the data elements are points in the space. Two data elements are "local" if their keywords are lexicographically close or they have common keywords. Thus, we map documents that are local in this multi-dimensional index space to indices that are local in the 1-dimensional index space, which are then mapped to the same node or to nodes that are close together in the overlay network. This mapping is derived from a locality-preserving mapping called Space Filling Curves (SFC) [1, 8]. In the current implementation, we use the Hilbert SFC [1, 8] for the mapping, and Chord [11] for the overlay network topology.

Note that locality is not preserved in an absolute sense in this keyword space; documents that match the same query (i.e. share a keyword) can be mapped to disjoint fragments of the index space, called clusters. These clusters may in turn be mapped to multiple nodes so a query will have to be efficiently routed to these nodes. We present an optimization based on successive refinement and pruning of queries that significantly reduces the number of nodes queried.

Unlike the consistent hashing mechanisms, SFC does not necessarily result in uniform distribution of data elements in the index space - certain keywords may be more popular and hence the associated index subspace will be more densely populated. As a result, when the index space is mapped to nodes load may not be balanced. We present a suite of relatively inexpensive load-balancing optimizations and experimentally demonstrate that they successfully reduce the amount of load imbalance.

The rest of this paper is structured as follows. Section 2 compares the presented system to related work. Section 3 describes the architecture and operation of the presented P2P indexing system. Section 4 presents an experimental evaluation of the system and Section 5 presents our conclusions.

2 Related Work

Currents approaches to Web service discovery can be broadly classified as centralized or decentralized. The centralized approach include UDDI [16], where central registries are used to store Web Service descriptions. The distributed approaches are based on P2P systems. Existing P2P systems developed for Web Service discovery include (1) the Hypercube ontology-based P2P system [9] that focusses on the discovery of Web Services in the Semantic Web, and (2) the Speed-R [10] system that makes use of ontologies to organize web service discovery registries and addreses scalabilty of the discovery process.

The system presented in this paper enhances the Web Service discovery with a flexible, P2P-based keyword search. In this system, Web Services (description files) are indexed based on the keywords describing them. The P2P indexing system used is based on the Chord [11] data lookup protocol. Chord [11] and other lookup protocols (CAN [6], Pastry [7] and Tapestry [13]) offer guarantees, but lack flexible search capabilities - i.e. they only support lookup using a unique identifier. Our system enhances the lookup protocol with the flexibility of keyword searches (allowing partial keywords and wildcards) while offering the same guarantees. The underlying overlay network in our system has to be structured, unlike Gnutella-like systems [14] which have unstructured overlays. These systems however do not offer guarantees and do not scale well.

3 System Architecture and Design

The architecture of the presented P2P indexing system is similar to data-lookup systems [11, 6], and essentially implements an Internet-scale distributed hash table. The architecture consists of the following components: (1) a locality preserving mapping that maps data elements to indices, (2) an overlay network topology, (3) a mapping from indices to nodes in the overlay network, (4) load balancing mechanisms, and (5) a query engine for routing and efficiently resolving keyword queries using successive refinements and pruning. These components are described below.

3.1 *Constructing an Index Space: Locality Preserving Mapping*

A key component of a data-lookup system is defining the index space and deterministically mapping data elements to this index space. To support complex keyword searches in a data lookup system, we associate each data element with a sequence of keywords and define a mapping that preserves keyword locality.

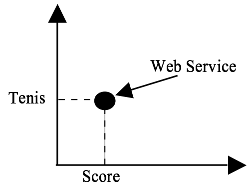

Figure 1: A 2-Dimensional keyword space. The data element *"Web Service"* is described by keywords *"tenis:score"*.

These keywords form a multidimensional keyword space where data elements are points in the space and the keywords are the coordinates. The keywords can be viewed as base-*n* numbers, for example *n* can be 10 if the keywords are numbers or 26 if the keywords are words in a language with 26 alphabetic characters. Two data elements are considered "local" if they are close together in this keyword space. For example, their keywords are lexicographically close (e.g. *comp*uter and *comp*any) or they have common keywords. Not all combinations of characters represent meaningful keywords, resulting in a sparse keyword space with non-uniformly distributed clusters of data-elements. An example of keyword space is shown in Figure 1.

To efficiently support queries using partial keywords and wildcards, the index space should preserve locality and be recursive so that these queries can be optimized using successive refinement and pruning. Such an index space is constructed using the Hilbert SFC as described below.

3.2 *Hilbert Space-Filling Curve*

A SFC is a continuous mapping from a *d*-dimensional space to a 1-dimensional space f: $N^d \rightarrow N$. The *d*-dimensional space is viewed as a *d*-dimensional cube, which is mapped onto a line such that the line passes once through each point in the volume of the cube, entering and exiting the cube only once. Using this mapping, a point in the cube can be described by its spatial coordinates, or by the length along the line, measured from one of its ends.

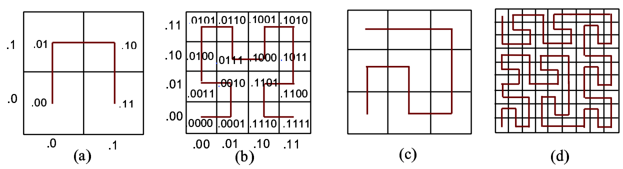

Figure 2: Space-filling curve approximations for d = 2: n = 2 (a) 1^{st} order approximation (b) 2^{nd} order approximation; n = 3 (c) 1^{st} order approximation, (d) 2^{nd} order approximation.

The construction of SFCs is recursive. The *d*-dimensional cube is first partitioned into n^d equal sub-cubes. An approximation to a space-filling curve is obtained by joining the centers of these sub-cubes with line segments such that each cell is joined with two adjacent cells. An example is presented in Figure 2 (a), (c). The same algorithm is used to fill each sub-cube. The curves traversing the sub-cubes are rotated and reflected such that they can be connected to form a single continuous curve that passes only once through each of the n^{2d} regions. The line that connects n^{kd} cells is called the k^{th} approximation of the SFC. Figures 2(b) and 2(d) show the second orders approximations for the curves in Figures 2(a) and 2(c) respectively.

An important property of SFCs is *digital causality*, which comes directly from its recursive nature. A unit length curve constructed at the k^{th} approximation has an equal portion of its total length contained in each sub-hypercube; it has n^{k*d} equal segments. If distances across the line are expressed as base-*n* numbers, then the numbers that refer to all the points that are in a sub-cube and belong to a line segment are identical in their first *k*d* digits. This property is illustrated in Figure 2 (a), (b).

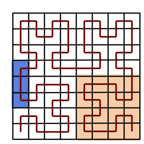

Figure 3: Clusters on a 3^{rd} order space-filling curve (d = 2, n = 2). The colored regions represent clusters: 3-cell cluster and 16-cell cluster.

Finally, SFCs are *locality preserving*. Points that are close together in the 1-dimensional space (the curve) are mapped from points that are close together in the *d*-dimensional space. For example, for k $\geq$ 1, d $\geq$ 2, the k^{th} order approximation of a *d*-dimensional Hilbert space filling curve maps the sub-cube $[0, 2^k - 1]^d$ to $[0, 2^{kd} - 1]$. The reverse property is false, not all adjacent sub-cubes in the *d*-dimensional space are adjacent or even close on the curve. A group of contiguous sub-cubes in *d*-dimensional space will typically be mapped to a collection of segments on the SFC. These segments called clusters are shown in Figure 3.

In our system, SFCs are used to generate the 1-d index space from the multi-dimensional keyword space. Applying the Hilbert mapping to this multi-dimensional space, each data element can be mapped to a point on the SFC. Any query composed of keywords, partial keywords, or wildcards, can be mapped to regions in the keyword space and the corresponding clusters in the SFC.

3.3 *Mapping Indices to Peers and the Overlay Network*

The next step consists of mapping the 1-dimensional index space onto an overlay network of peers. This step is similar to existing data-lookup systems. In our current implementation we use the Chord [11] overlay network topology. In Chord each node has a unique identifier ranging from 0 to 2^m-1. These identifiers are arranged as a circle modulo 2^m. Each node maintains information about (at most) *m* neighbors, called *fingers*, in a *finger table*. The i^{th} finger node is the first node that succeeds the current node by at least 2^{i-1}, where $1 \leq i \leq m$. The finger table is used for efficient routing.

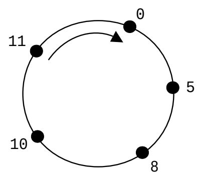

Figure 4: Example of the overlay network. Each node stores the keys that map to the segment of the curve between itself and the predecessor node.

In our implementation, node identifiers are generated randomly. Each data element is mapped, based on its SFC-based index or key, to the first node whose identifier is equal to or follows the key in the identifier space. The node is called the *successor* of the key. Consider the sample overlay network with 5 nodes and an identifier space from 0 to 16, as shown in Figure 4. In this example, data elements with keys 6, 7, and 8, will map to node 8, the successor of these keys. The management of node joins, departures, and failures is described below.

Node Joins: The joining node has to know about at least one node already in the network. It randomly chooses an identifier from the identifier space and sends a join message with this identifier to the known node. This message is routed across the overlay network to the successor of the new node (based on the new identifier). The joining node is inserted into the overlay network at this point and takes a part of the successor node's load. The cost for joining is O($\log_2{}^2$N) messages[2].

Node Departures: The finger tables of the nodes that have entries pointing to the departing node have to be updated. The cost is O($\log_2{}^2$N) messages.

Node Failures: When a node fails, the finger tables that have entries pointing to it will be incorrect. Each node periodically runs a stabilization algorithm where it chooses a random entry in its finger table, checks for its state, and updates it if required.

Data Lookup: Nodes are efficiently located based on their content. Data lookup takes O($\log_2$N) hops. In our system partial queries will typically require interrogating more than one node, as the desired information may be stored at multiple nodes in the system.

3.4 *The Query Engine*

The primary function of the query engine is to efficiently process user queries. As described above, data elements in the system are associated with a sequence of one or more keywords (up to *d* keywords, where *d* is the dimensionality of the keyword space). The queries can consist of a combination of keywords, partial keywords, or wildcards. The expected result of a query is the complete set of data elements that match the user's query. For example, (flight, schedule), (flight, info*), (flight, *) are all valid queries.

Query Processing: Processing a query consists of two steps: translating the keyword query to relevant clusters of the SFC-based index space, and querying the appropriate nodes in the overlay network for data-elements.

If the query consists of whole keywords (no wildcard) it will be mapped to at most one point in the index space, and the node containing the result is located using the network's lookup protocol. If the query contains partial keywords and/or wildcards, the query identifies a set of points (data elements) in the keyword space that correspond to a set of points (indices) in the index space. In Figure 5, the query (000, *) identifies 8 data elements, the squares in the vertical region. The index (curve) enters and exits the region three times, defining three segments of the curve or clusters (marked by different patterns). Similarly the query (1*, 0*) identifies 16 data elements, defining the square region in Figure 5. The SFC enters and exits this region once, defining one cluster.

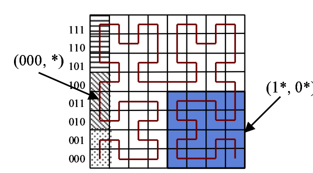

Figure 5: Regions in a 2-dimensional space defined by the queries (000, *) and (1*, 0*).

Each cluster may contain zero, one or more data elements that match the query. Depending on its size, an index space cluster may be mapped to one or more adjacent nodes in the overlay network. A node may also store more than one cluster. Once the clusters associated with a query are identified, straightforward query processing consists of sending a query message for each cluster. A query message for a cluster is routed to the appropriate node in the overlay network as follows. The overlay network provides us with a data lookup protocol: given an identifier for

[2]N is the number of nodes in the system

a data element, the node responsible for storing it is located. The same mechanism can be used to locate the node responsible for storing a cluster using a cluster identifier. The cluster identifier is constructed using SFC's digital causality property. The digital causality property guarantees that all the cells that form a cluster have the same first i digits. These i digits are called the cluster's *prefix* and form the first i digits of the m digit identifier. The rest of the identifier is padded with zero.

The node that initiated the query can not know if a cluster is stored in the network or not, or if multiple clusters are stored at the same node, to make optimizations. The number of clusters can be very high, and sending a message for each cluster is not a scalable solution. For example, consider the query (000, *) in Figure 5, but using base-26 digits and higher order approximation of the space-filling curve. The cost of sending a message for each cluster can be prohibitive.

Query Optimization: Query processing can be made scalable using the observation that not all the clusters that correspond to a query represent valid keywords as the keyword space and the clusters are typically sparsely populated with data elements. This is because we are using base-n numbers as coordinates along each dimension of the space and not all the base-n numbers are valid keywords. The number of messages sent and nodes queried can be significantly reduced by filtering out the useful clusters early. However, useful clusters cannot be identified at the node where the query is initiated. The solution is to consider the recursive nature of the SFC and its digital causality property, and to distribute the process of cluster generation at multiple nodes in the system, the ones that might be responsible for storing them.

Since the SFC generation is recursive, and clusters are segments on the curve, these clusters can also be generated recursively. This recursive process can be viewed as constructing a tree. At each level of the tree the query defines a number of clusters, which are refined, resulting in more clusters for the next level. The tree can now be embedded into the overlay network: the root performs first query refinement, and each node further refines the query, sending the resulting sub-queries to the appropriate nodes in the system.

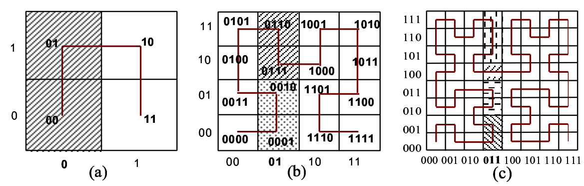

Figure 6: Recursive refinement of the query (011, *). (a) one cluster on the first order Hilbert curve, (b) two clusters on the second order Hilbert curve, (c) four clusters on the third order Hilbert curve.

Consider the following example. We want to process the query (011, *) in a 2-dimensional space using base-2 digits for the coordinates. Figure 6 shows the successive refinement for the query and Figure 7 shows the corresponding tree. The leaves of the tree represent all possible matches for the query.

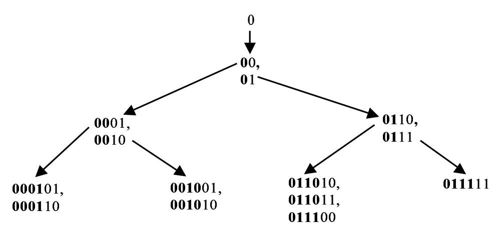

Figure 7: Recursive refinement of the query (011, *) viewed as a tree. Each node is a cluster, and the bold characters are cluster's prefixes.

The query optimization consists of pruning nodes from the tree during the construction phase. As a result of the load-balancing steps (see Section 3.5), the nodes tend to follow the distribution of the data in the index space - i.e., a larger number of nodes are assigned to denser portions of the index space, and no nodes for the empty portions. If we embed the query tree onto the ring topology of the overlay network, we can prune away many of the nodes that do not contain valid data elements, knowing that their children do not exist in the system.

Figure 8 illustrates the process, using the query in Figure 6 as an example. The leftmost path (solid arrows) and the rightmost path (dashed arrows) of the tree presented in Figure 7 are embedded into the ring network topology. The overlay network uses 6 digits for node identifiers. The arrows are labeled with the prefix of the cluster being queried. The query initiated at node 111000. The first cluster has prefix 0, so the cluster's identifier will be 000000. The cluster is sent to node 000000. At this node the cluster is further refined, generating two sub-clusters, with prefixes 00 and 01. The cluster with prefix 00 remains at the same node. After processing, the sub-cluster 0001 is sent to node 000100. The cluster with prefix 01 and identifier 010000 is sent to node 011110 (dashed line). This cluster will not be refined because the node's identifier is greater than the query's identifier, and all

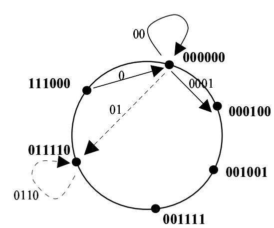

Figure 8: Embedding the leftmost tree path (solid arrows) and the rightmost path (dashed arrows) onto the overlay network topology.

matching data elements are stored at this node.

A second query optimization is used to reduce the number of messages involved. It is based on the observation that multiple sub-clusters of the same cluster can be mapped to the same node. To reduce the number of messages, we sort the sub-clusters in increasing order and send the first one in the network. The destination node of the sub-cluster replies with its identifier. The sending node then aggregates all the sub-clusters associated with this identifier and sends them as a single message routed to the destination.

3.5 *Balancing Load*

As we mentioned earlier, the original *d*-dimensional keyword space is sparse and data elements form clusters in this space instead of being uniformly distributed in the space. As the Hilbert SFC-based mapping preserves keyword locality, the index space will have the same properties. Since the nodes are uniformly distributed in the node identifier space when the data elements are mapped to the nodes, the load will not be balanced. Additional load balancing has to be performed. We have defined two load-balancing steps as described below.

Load Balancing at Node Join: At join, the incoming node generates several identifiers (e.g., 5 to 10) and sends multiple join messages using these identifiers. Nodes that are logical successors of these identifiers respond reporting their load. The new node uses the identifier that will place it in the most loaded part of the network. This way, nodes will tend to follow the distribution of the data from the very beginning. With *n* identifiers being generated, the cost to find the best successor is $O(n*\log_2 N)$, and the cost to update the tables remains the same, $O(\log_2{}^2 N)$. However, this step is not sufficient by itself. The runtime load-balancing algorithms presented below improve load distribution.

Load Balancing at Runtime: The runtime load-balancing step consists of periodically running a local load-balancing algorithm between few neighboring nodes. We propose two load-balancing algorithms. In the first algorithm, neighboring nodes exchange information about their loads, and the most loaded nodes give a part of their load to their neighbors. The cost of load-balancing at each node using this algorithm is $O(\log_2{}^2 N)$. As this is expensive, this load-balancing algorithm cannot be run very often.

The second load-balancing algorithm uses virtual nodes. In this algorithm, each physical node houses multiple virtual nodes. The load at a physical node is the sum of the load of its virtual nodes. When the load on a virtual node goes above a threshold, the virtual node is split into two or more virtual nodes. If the physical node is overloaded, one or more of its virtual nodes can migrate to less loaded physical nodes (neighbors or fingers). An evaluation of the load balancing algorithms is presented in Section 4.

4 Experimental Evaluation

The performance of our P2P indexing system is evaluated using a simulator. The simulator implements the SFC-based mapping, the Chord-based overlay network, the load-balancing steps, and the query engine with the query optimizations described above. As the overlay network configuration and operations are based on Chord [11], its maintenance costs are of the same order as in Chord. An evaluation of the query engine and the load-balancing algorithms is presented below.

4.1 *Evaluating the Query Engine*

The overlay network used to evaluate the query engine consists of 1000 to 5400 nodes. 2-dimensional (2D), and a 3-dimensional (3D) keyword spaces are used in this evaluation. Finally, we use up to 10^6 keys (unique keyword combinations) in the system, each of which could be associated with one or more data elements. We measure the following:

Number of routing nodes: the nodes that route the query. Some of them also process the query.

Number of processing nodes: the nodes that actually process the query, refine it, and search for matches. The goal is to restrict processing only to those nodes that store matching data elements.

Number of data nodes: the nodes that have data elements matching the query.

Number of messages required to resolve a query. When using the query optimization each message is a sub-query that searches for a fraction of the clusters associated with the original query.

The types of queries used in the experiments are:

Q1: Queries with one keyword or partial keyword, e.g. (library, *) for 2D, (sport, *, *) for 3D.

Q2: Queries with two to three keywords or partial keywords (at least one partial keyword), e.g. (math*, add*) for 2D, (sport, tenis, *) for 3D.

4.1.1 Evaluating a 2-dimensional keyword space

This experiment represents a P2P indexing system where the number of keys and data elements in the system increases as the number of nodes increases. The system size increases from 1000 nodes to 5400 nodes, and the number of stored keys increases from $2*10^5$ to 10^6.

The results for experiments using six different type Q1 queries (query1 - query6) are plotted in Figure 9. Each query resulted in a different number of matches.

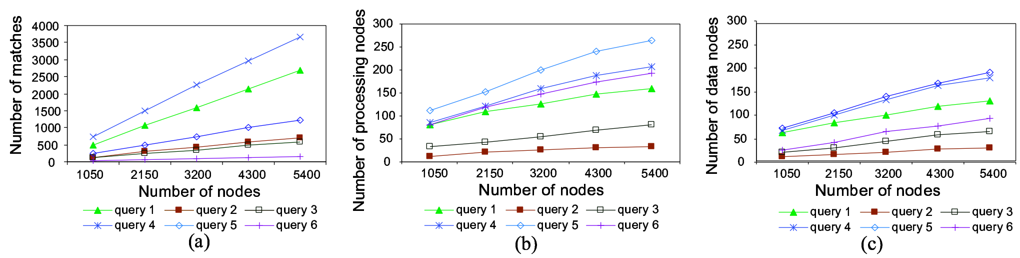

Figure 9: Results for query type Q1, 2D: (a) the number of matches for the queries, (b) the number of nodes that process the query, (c) the number of nodes that found matches for the query.

As seen in Figures 9 (b) and (c), the number of processing and data nodes is a fraction of the total nodes and increase at a slower rate than the system size. The number of processing nodes does not necessarily depend on the number of matches. For example, to solve a query with 160 matches (query6) can be more costly than solving a query with 2600 matches (query1). This is due to the recursive processing of queries and the distribution of keys in the index space. In order to optimize the query, we prune parts of the query tree based on the data stored in the system. The earlier the tree is pruned, the fewer processing nodes will be required and the better the performance will be. For example, if the query being processed is (computer, *) and the system contains a large number of data elements with keys that start with "com" (e.g., company, commerce, etc.) but do not match the query, the pruning will be less efficient and will result in a larger number of processing nodes.

Note that even under these conditions, the results are good. A keyword search system like Gnutella [14] would have to query the entire network using some form of flooding to guarantee that all the matches to a query are returned, while in the case of a data lookup system such as Chord [11], one would have to know all the matches a priori and look them up individually.

Figure 9 (c) plots the number of data nodes which are a subset of the processing nodes. The number of data nodes is close to the number of processing nodes, indicating that the query optimizations effectively reduce the number of nodes involved. Furthermore, comparing the number of matches to the number of data nodes demonstrates the clustering property of the Hilbert SFC index space, which can be defined as a ratio of the number of matches for a query to the number of data nodes that store these matches.

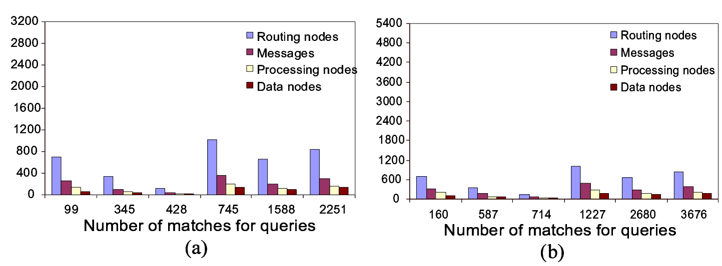

Figure 10: Results for all the metrics, 2D: (a) for a 3200 node system and $6*10^5$ keys, (b) for a 5400 node system and 10^6 keys.

Figure 10 plots all the measurements for two system sizes and demonstrates the scalability of the presented system. Each group of bars represents the results for a particular query. The maximum value, plotted on the vertical axis, is the size of the network. As seen in the graph, the processing nodes are a small fraction of the routing nodes, and a very small fraction of the entire system. The processing node population does not increase as fast as the system. The number of data nodes is close to the number of processing nodes. The number of messages used is almost twice the number of processing nodes, which is as expected.

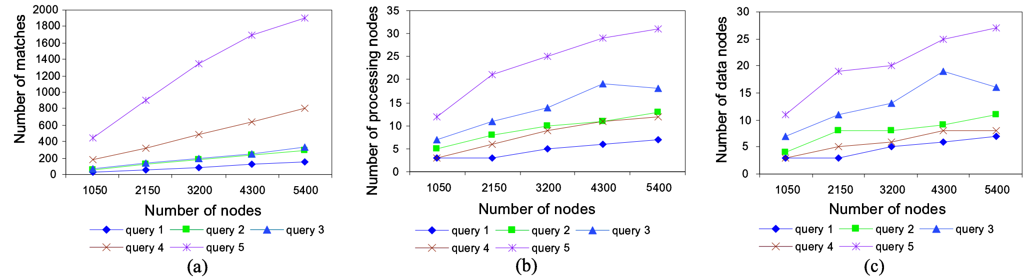

Figure 11: Results for query type Q2, 2D: (a) the number of matches for the queries, (2) the number of data nodes.

Figure 11 shows the corresponding results for experiments with 5 different type Q2 queries. As expected, the results are significantly better than those for type Q1 queries. This is because query optimization and pruning are effective when both keywords are at least partially known.

4.1.2 Evaluating a 3-dimensional keyword space

The experiment conducted for the 2D keyword space was repeated for a 3D keyword space. Overall, the results are similar to the 2D case. The growth of the system, either in the number of nodes or in the quantity of data stored, or both, affects the behavior of the queries in the same way. The only difference is the magnitude of the results. As described in Section 1, documents that share a specific keyword will typically be mapped to disjoint fragments on the curve (clusters). In the 3D case the number of such fragments is larger than in the 2D case - 3 keywords used instead of 2 and result in a "longer" curve. Consequently, the results obtained for the 3D case for all the metrics have the same pattern as the 2D case but a larger magnitude.

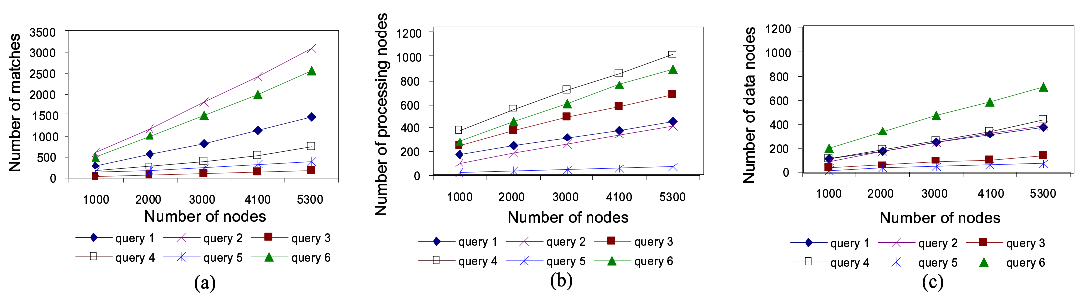

Figure 12: Results for query type Q1, 3D: (a) the number of matches for the queries, (b) the number of nodes that process the query, (c) the number of nodes that found matches for the queries.

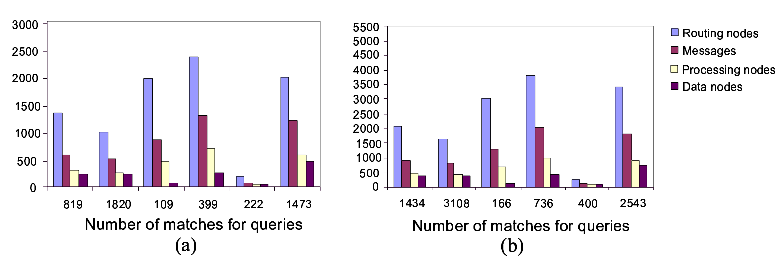

Figure 13: Results for all metrics, 3D: (a) for 3000 node system and $6*10^5$ keys, (b) for 5300 node system and 10^6 keys.

The results for experiments in the 3D case follow a similar pattern to the 2D cases presented in Section 4.1.1. The main difference from the 2D cases is the magnitude of the graphs; results for the 3D case may be larger by two to three times. This is expected and is due to the fact that for the same types of queries there are more clusters in the 3D case than in the 2D case.

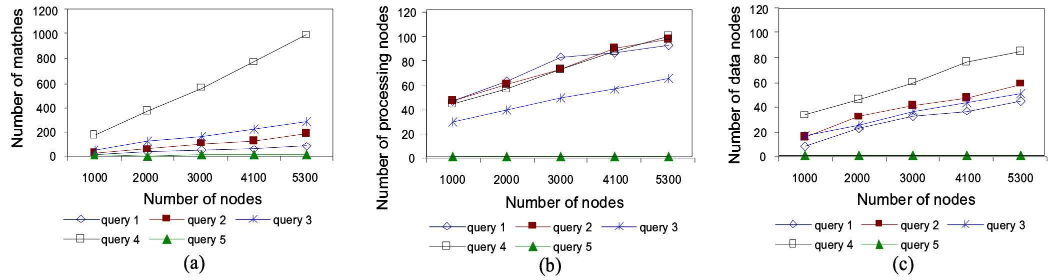

Figure 14: Results for query type Q2, 3D: (a) the number of matches for queries, (b) the number of processing nodes, (c) the number of data nodes.

4.2 *Evaluating the Load Balancing Algorithms*

The quality of the load balance achieved by the two load balancing operations is evaluated for the initial distribution shown in Figure 15 (a). As expected, the original distribution is not uniform. The load-balance step at node join helps to match the distribution of nodes with the distribution of data. The resulting load balance is plotted in Figure 15 (b). While the resulting load distribution is better than the original distribution in Figure 15 (a), this step by itself does not guarantee a very good load balance. However, when it is used in conjunction with the runtime load-balancing step, the resulting load balance improves significantly as seen in Figure 15 (b). The load is almost evenly distributed in this case.

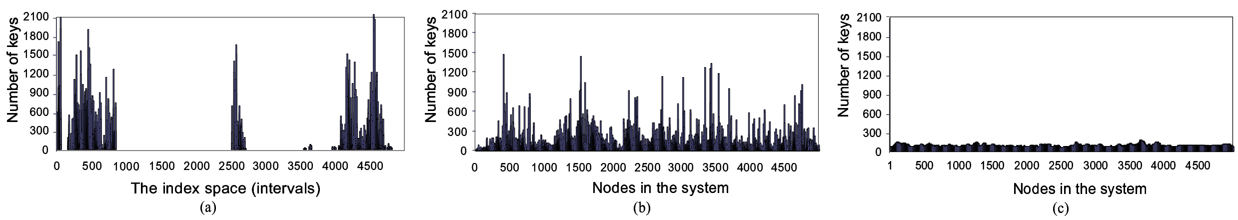

Figure 15: (a) The distribution of the keys in the index space. The index space was partitioned into 5000 intervals. The Y-axis represents the number of keys per interval; (b)The distribution of the keys at nodes when using only the load balancing at node join technique; (c) The distribution of the keys at nodes when using both the load balancing at node join technique, and the local load balancing

5 Conclusions

In this paper, we presented the design and evaluation of a P2P indexing system that can be used for Web Service discovery. The presented system implements an Internet-scale DHT that supports complex searches using keywords, partial keywords and wildcards, while guaranteeing that all data elements in the system that match the query are found with bounded costs in terms of the number of messages and nodes involved in the query. A key component is a locality-preserving mapping (Hilbert's SFC) from the data element keyword space to the index space.

An experimental evaluation of the system (using a simulator) demonstrated the scalability of the system, the ability of the mapping to preserve keyword locality and the effectiveness of the load balancing algorithms.

References

[1] T. Bially. A class of dimension changing mapping and its application to bandwidth compression. Ph. D Thesis, Polytechnic Institute of Brooklyn, June 1967.

[2] I. Clarke, O. Sandberg, B. Wiley and T.W. Hong. Freenet: A distributed Anonymous Information Storage and Retrieval System. In Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability, Berkeley, California, June 2000.

[3] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, July 2002.

[4] Q. Lv, P. Cao, E. Cohen., K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In Proceedings of the 16th international conference on Supercomputing, June 2002.
[5] B. Moon, H.V. Jagadish, C. Faloutsos and J.H. Saltz. Analysis of the Clustering Properties of Hilbert Space-filling curve. Submitted to IEEE Transactions on Knowledge and Data Engineering, March1996.
[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A Scalable Content-Addressable Network. In Proceedings of ACM SIGCOMM, 2001.
[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
[8] H. Sagan. Space-Filling Curves. Springer-Verlag, 1994.
[9] M. Schlosser, M. Sintek, S. Decker and W. Nejdl. A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services. Second International Conference on Peer-to-Peer Computing (P2P'02) September 05-07, Linkoping, Sweden 2002.
[10] K. Sivashanmugam, K. Verma, R. Mulye, Z. Zhong, A. Sheth. Speed-R: Semantic P2P Environment for diverse Web Service Registries.
[11] I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In Proceedings of ACM SIGCOMM, 2001
[12] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Systems. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, July 2002.
[13] B. Y. Zhao, J. Kubiatowicz and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Widearea Location and Routing. Technical Report No. UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, April 2001.
[14] Gnutella webpage: http://gnutella.wego.com/.
[15] Open Grid Services Architecture (OGSA): http://www.globus.org/ogsa/.
[16] Universal Description, Discovery and Integration: UDDI Technical White paper, September 2000.