
World Wide Web: Internet and Web Information Systems, 7, 211–229, 2004

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

CRISTINA SCHMIDT and MANISH PARASHAR {cristins,parashar}@caip.rutgers.edu
The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers University, 94 Bret Road, Piscataway, NJ 08854, USA

Abstract

Web Services are emerging as a dominant paradigm for constructing and composing distributed business applications and enabling enterprise-wide interoperability. A critical factor to the overall utility of Web Services is a scalable, flexible and robust discovery 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 innovation 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.

Keywords: Web Service discovery, P2P, SFC-based indexing, flexible queries, DHT

1. Introduction

Web Services are emerging as a dominant paradigm for distributed computing in industry as well as academia. They are enterprise applications that exchange data, share tasks, and automate processes over the Internet. Web Services 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 ultimate utility of the Web Services and must support scalable, flexible and robust discovery mechanisms. Registries have traditionally had a centralized architecture (e.g., UDDI [14]) consisting of multiple 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

* 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 numbers PC295251 and 1052856.

approaches that have been proposed. These systems (e.g., [11,12]) build on Peer-to-Peer (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 search capabilities, to support Web Service discovery. Web Services can be described in different ways, according to existing standards (e.g., WSDL, ontologies), and can be characterized by a set of keywords.

For example consider an airline company that provides the following services: a service that provides the schedule of flights, a service that provides information about current flights, and a booking service. These services can be described by keywords as follows: the service that provides the schedule is described by the keywords *flights* and *schedule,* the service that provides current flights information is described by the keywords *flights* and *information,* and the booking service is described by the keywords *flights* and *booking.* To make these services known, the service provider (in this case the airline company) publishes them in a registry maintained by a service broker. Now the service requesters (e.g., users, applications) can discover them by searching the registry, using keywords.

In our system we use 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 [8,13]. The key difference is in the way we map data elements[1] to the index space. In existing systems, this is done using consistent 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 multidimensional 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) [1,10]. In the current implementation, we use the Hilbert SFC [1,10] for the mapping, and Chord [13] 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

Current approaches to Web Service discovery can be broadly classified as centralized or decentralized. The centralized approach include UDDI [14], 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 [11] that focusses on the discovery of Web Services in the Semantic Web, and (2) the Speed-R [12] system that makes use of ontologies to organize web service discovery registries and addresses scalability 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 [13] data lookup protocol. Chord [13] and other lookup protocols (CAN [8], Pastry [9] and Tapestry [16]) 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 [4] 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 [8,13], 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.

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 $f : \mathbb{N}^d \rightarrow \mathbb{N}$. 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.

The construction of SFCs is recursive. The d-dimensional cube is first partitioned into n^d equal subcubes. An approximation to a space-filling curve is obtained by joining the centers of these subcubes with line segments such that each cell is joined with two adjacent

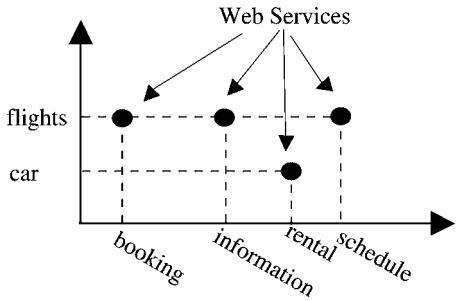

Figure 1. 2-dimensional keyword space. There are four web services described by keywords: (*flight*, *schedule*), (*flight*, *information*), (*flight*, *booking*) and (*car*, *rental*).

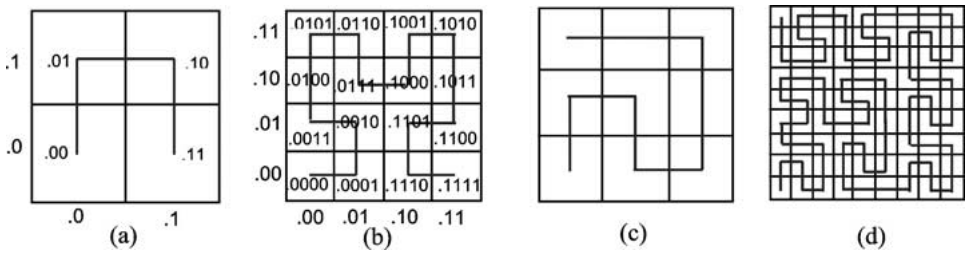

Figure 2. Space-filling curve approximations for $d = 2$. $n = 2$: (a) 1st order approximation, (b) 2nd order approximation; $n = 3$: (c) 1st order approximation, (d) 2nd order approximation.

cells. An example is presented in Figures 2(a) and (c). The same algorithm is used to fill each subcube. The curves traversing the subcubes 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 kth order approximation of the SFC. Figures 2(b) and (d) show the second order approximations for the curves in Figures 2(a) and (c), respectively.

An important property of SFCs is *digital causality*, which comes directly from its recursive nature. A unit length curve constructed at the kth approximation has an equal portion of its total length contained in each subhypercube; it has n^{kd} 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 subcube and belong to a line segment are identical in their first kd digits. This property is illustrated in Figures 2(a) and (b).

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 $k \geqslant 1$, $d \geqslant 2$, the kth order approximation of a d-dimensional Hilbert space filling curve maps the subcube $[0, 2^k - 1]^d$ to $[0, 2^{kd} - 1]$. The reverse property is not true, not all adjacent subcubes in the d-dimensional space are adjacent or even close on the curve. A group of contiguous subcubes 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.

A number of different SFCs with the properties described above have been proposed in [10]. A selection of these, the Morton curve (z-curve), Gray code, and Hilbert curve, are shown in Figure 4. The degree to which locality is preserved by a particular SFC is defined

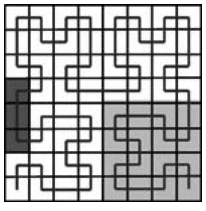

Figure 3. Clusters on a 3rd order space-filling curve ($d = 2$, $n = 2$). The colored regions represent clusters: 3-cell cluster and 16-cell cluster.

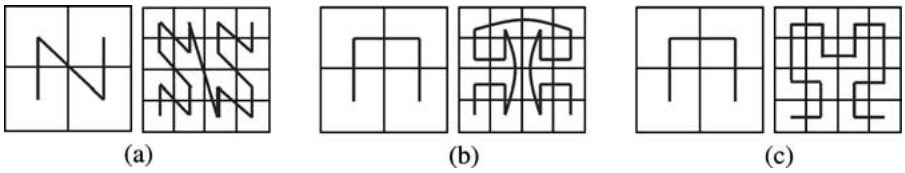

Figure 4. Examples of space-filling curves (first and second order): (a) Morton curve, (b) Gray code, (c) Hilbert curve.

by the number of clusters that it maps an arbitrary region in the d-dimensional space to. It is widely believed that the Hilbert space-filling curve achieves the best clustering [6] and hence is used in this research to map data elements to the index space.

In our system, SFCs are used to generate the 1-D index space from the multidimensional keyword space. Applying the Hilbert mapping to this multidimensional 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 generated using the Hilbert SFC onto an overlay network of peers (nodes). This step is similar to existing data-lookup systems (e.g., CAN [8], Chord [13]). In fact, in our current implementation, we use the Chord [13] overlay network topology, routing algorithms and management mechanism as described below.

In the Chord overlay network, each node has a unique identifier ranging from 0 to $2^m - 1$. These identifiers are arranged as a circle modulo 2^m, where each node maintains information about its successor and predecessor on the circle. Additionally, each node also maintains information about (at most) m other neighbors, called *fingers*, in a *finger table*. The ith finger node is the first node that succeeds the current node by at least 2^{i-1}, where $1 \leqslant i \leqslant m$. The finger table is used for efficient routing.

In our implementation, node identifiers are generated randomly. Each data element is mapped to nodes 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, shown in Figure 5. In this example, data elements with keys 6, 7, and 8 will map to node 8, which is the *successor* of these keys. The management of node joins, departures, and failures using this overlay network and mapping scheme are described below.

Node joins. When a node wants to join the overlay network, it has to know about at least one node that is already a part of the network. The joining node randomly chooses an identifier from the identifier space (0 to $2^m - 1$) and sends a join message with this identifier to the known node. This message is routed across the overlay network until it reaches the node that is the successor of the new node based on the new identifier. The joining node

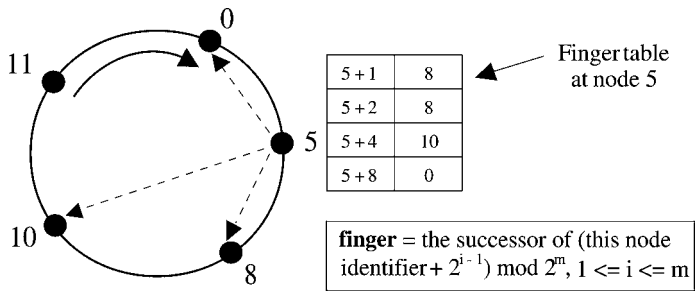

Figure 5. Example of the overlay network. Each node stores the keys that map to the segment of the curve between itself and the predecessor node.

is inserted into the overlay network at this point and takes a part of the successor node's load. The new node constructs its finger table and the direct neighbor entries (successor and predecessor). It also updates the finger tables at other nodes in the system that should point to this node. The cost for a node joining the network is $O(\log^2 N)$ messages, where N is the number of nodes in the system.

Node departures. When a node leaves the system, the finger tables of nodes that have entries pointing to the leaving node have to be updated. The cost for updating these tables is $O(\log^2 N)$ messages (N is the number of nodes in the system), which is the same as the cost of a join.

Node failures. When a node fails, the finger tables that have entries pointing to the failed node will be incorrect. In order to prevent this from failing the system, each node periodically runs a stabilization algorithm where it chooses a random entry in its finger table and updates it, finding the correct finger node.

Data lookup. The data lookup protocol based on Chord can very efficiently locate nodes based on their content. Data lookup takes $O(\log N)$ hops, where N is the total number of the nodes in the system. In our system, a partial query 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.

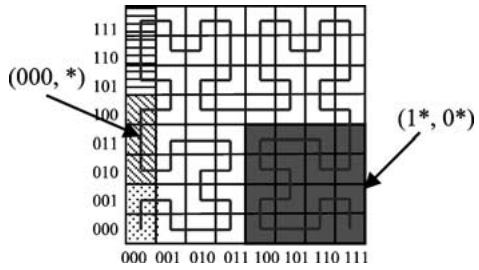

Figure 6. Regions in a 2-dimensional space defined by the queries (000, *) and (1*, 0*).

3.4.1. Straightforward solution. 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 6, 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 6. The SFC enters and exits this region once, defining one cluster.

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 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 cannot 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 6, 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.

3.4.2. Query optimization. Query processing can be made scalable using the observation that not all the clusters that correspond to a query represent valid keys or even if they

do, they may not be stored in the system. 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. So the index space is sparsely populated with keys, and there are clusters that are not stored in the system. Also, a node typically stores more than one cluster. As a result the number of nodes that have to be queried is by far smaller than the number of clusters. The number of messages sent and nodes queried can be significantly reduced by identifying 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 subqueries to the appropriate nodes in the system.

Consider the following example. We want to process the query (011, *) in a 2-dimensional space using base-2 digits for the coordinates. Figure 7 shows the successive refinement for the query and Figure 8 shows the corresponding tree. The leaves of the tree represent all possible matches for the query.

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 subtrees that are not stored in the system, or are stored at nodes already reached by the query.

Figure 9 illustrates the process, using the query in Figure 7 as an example. The leftmost path (solid arrows) and the rightmost path (dashed arrows) of the tree presented in Figure 8 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 solved. The query initiated at node 111000. The first cluster has prefix 0, so the cluster's identifier will be

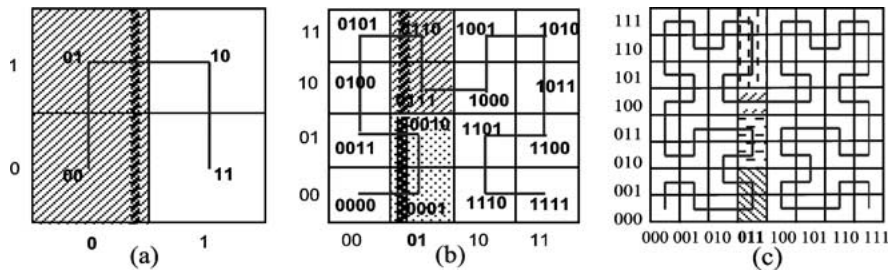

Figure 7. Recursive refinement of the query (011, *). (a) One cluster on the first order Hilbert curve, (b) two clusters on the second order Hilbert curve, (c) four clusters on the third order Hilbert curve.

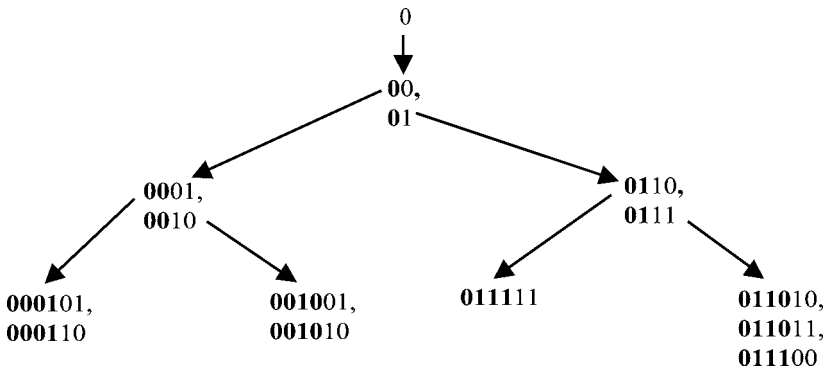

Figure 8. Recursive refinement of the query (011, *) viewed as a tree. Each node is a cluster, and the bold characters are cluster's prefixes.

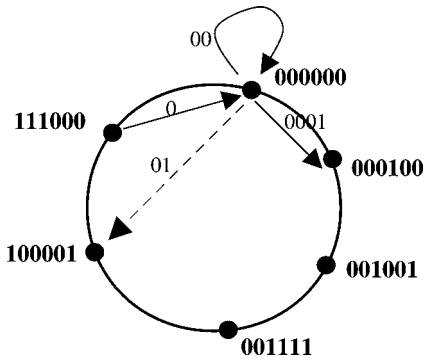

Figure 9. Embedding the leftmost tree path (solid arrows) and the rightmost path (dashed arrows) onto the overlay network topology.

000000. The cluster is sent to node 000000. At this node the cluster is further refined, generating two subclusters, with prefixes 00 and 01. The cluster with prefix 00 remains at the same node. After processing, the subcluster 0001 is sent to node 000100. The cluster with prefix 01 and identifier 010000 is sent to node 100001 (dashed line). This cluster will not be refined because the node's identifier is greater than the query's identifier, and all matching data elements for the subclusters (the whole subtree) should be 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 subclusters of the same cluster can be mapped to the same node. To reduce the number of messages, we sort the subclusters in increasing order and send the first one in the network. The destination node of the subcluster replies with its identifier. The sending node then aggregates all the subclusters 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., 5–10) 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 $O(n \log N)$, and the cost to update the tables remains the same, $O(\log^2 N)$. 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 $O(\log^2 N)$. 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 [13], 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 query engine is evaluated using three sets of experiments. The first set of experiments assumes that the number of data elements stored in the P2P system grows with the size

of the system. The results show that the system scales well under these conditions. The second set of experiments evaluates the case when the size of the system remains constant while the number of stored data-elements increases. The results show that the performance of the system does not decrease in this case. In the third set of experiments, the number of data elements is kept constant and the system size is increased. In this case, as the system size increases, clusters are distributed across a large number of nodes. The results show that the system continues to perform well. The overlay network used in the experiments consists of 1000–5400 nodes. Data elements are associated with up to two keywords for a 2-dimensional keyword space (2D), and with up to three keywords for a 3-dimensional keyword space (3D). 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. For each experiment, 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 subquery 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, tennis, *) for 3D.

4.2. Experiment 1 – increasing system size and increasing number of data elements

This experiment represents a typical P2P storage 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 $2 \cdot 10^5$ to 10^6.

A set of queries of each type were tested. The queries were chosen such that the number of matches represent the same fraction of the total data regardless of the size of the system (number of nodes) and the quantity of data. For each query we measured the number of nodes that process it (processing nodes) and the number of nodes that found matching data (data nodes). The results were averaged and normalized.

Figures 10(a) and 11(a) plot the results for Q1 and Q2 queries, Figure 10 for the 2D keyword space, and Figure 11 for the 3D keyword space. The vertical axis represent the percentage of nodes that participate in the query solving process. The horizontal axis plots the size of the system (number of nodes). As seen in the figures, the number of processing and data nodes is a small fraction of the total nodes in the system, and increases at

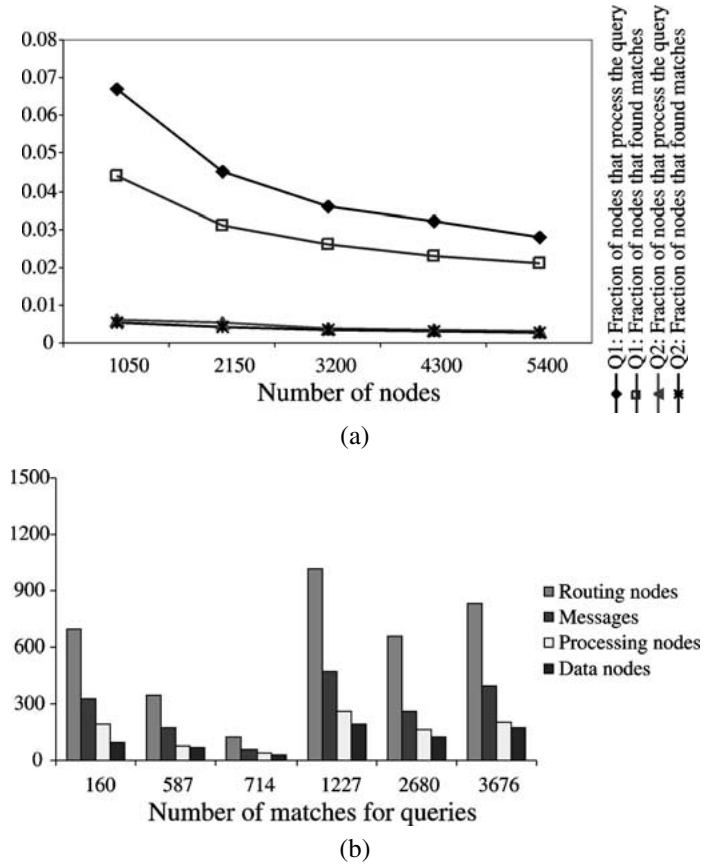

Figure 10. Experiment 1, 2D: (a) Results for query types Q1 and Q2; (b) results for query type Q1, in a system of 5400 nodes and 10^6 keys.

a slower rate than the system size. For a 2D keyword space (Figure 10(a)), the percentage of processing nodes is below 8%, and the percentage of data nodes is below 5%, and this percentage decreases as the system size increases (number of nodes and data), demonstrating the scalability of the system. 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. Also, Q2 queries are more efficient than Q1 queries. This is expected because query optimization and pruning are more effective when both keywords are at least partially known.

The 2D and 3D results follow a similar pattern, 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 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.

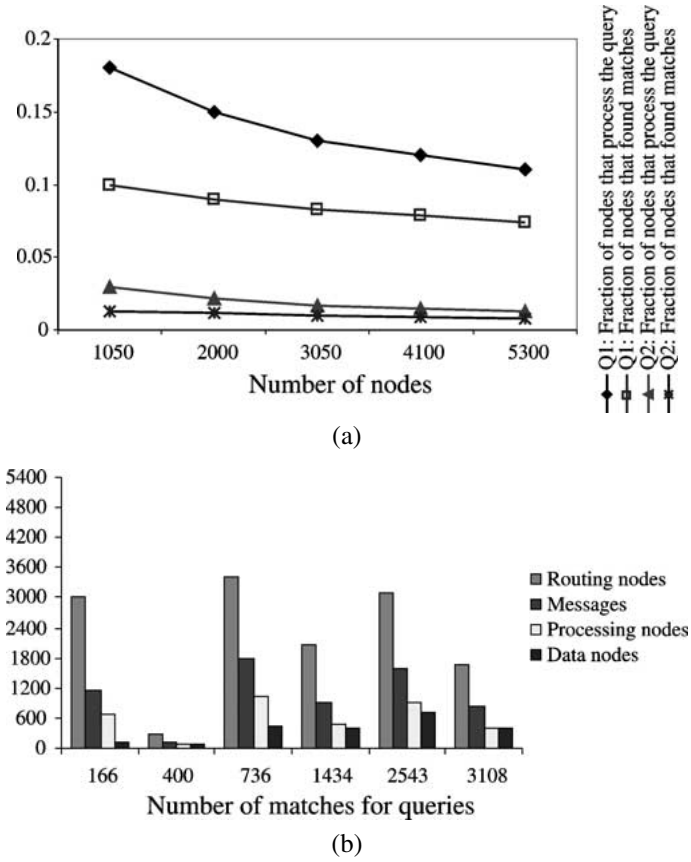

Figure 11. Experiment 1, 3D: (a) Results for query types Q1 and Q2; (b) results for query type Q1, the system size is 5300 nodes and 10^6 keys.

Figures 10(b) and 11(b) illustrate the fact that the number of processing nodes does not necessarily depend on the number of matches. We used 6 distinct Q1 queries (they are differentiated by the number of matches), in a system of 5400 nodes and 10^6 keys. For each query we measured the number of routing nodes, messages, processing nodes and data nodes. Each group of bars represents the results for a particular query. It can be seen that solving a query with 160 matches (the first group of bars, Figure 10(b)) can be more costly than solving a query with 714 matches (the third group of bars, Figure 10(b)). 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 in the query that 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 [4] 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 [13], one would have to know all the matches a priori and look them up individually.

4.3. Experiment 2 – constant system size and increasing number of data elements

In the second experiment, the size of the system remains constant as the number of keys and data elements increase. In this experiment, the system size is fixed at 2500 nodes while the number of stored keys increases from $2 \cdot 10^5$ to 10^6. A set of queries of type Q1 and Q2 were tested. For each query we measured the number of processing nodes and the number of data nodes. The results were averaged and normalized.

Figures 12 and 13 plot the results for the two types of queries, for the 2D and 3D keyword spaces. The vertical axis represents the percentage of nodes that participate in the query solving process. The horizontal axis plots the number of keys in the system. The results are as expected, the percentage of data nodes increases slowly as the number of keys increases. The percentage of processing nodes remains almost constant. Also, the number of data nodes is close to the number of processing nodes, demonstrating the efficiency of the query engine.

As illustrated in Figures 12 and 13, the results for query type Q2 have the same patterns as those corresponding to query type Q1, but they are significantly better. This is because query optimization and pruning are effective when both keywords are at least partially known.

The 2D and 3D results follow a similar pattern, the only difference is the magnitude of the results. We explained the reason for this behavior in Section 4.2.

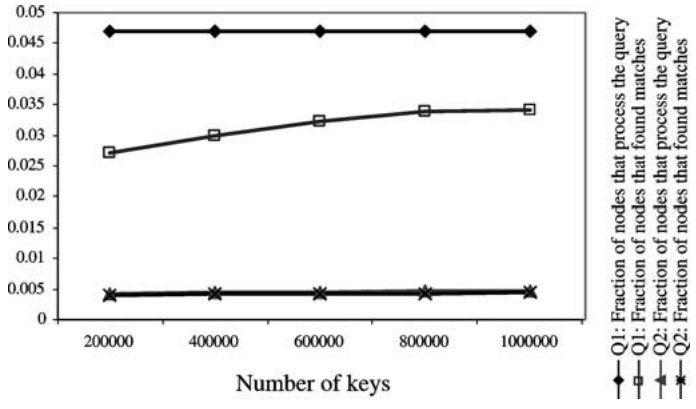

Figure 12. Experiment 2, 2D: Results for query types Q1 and Q2.

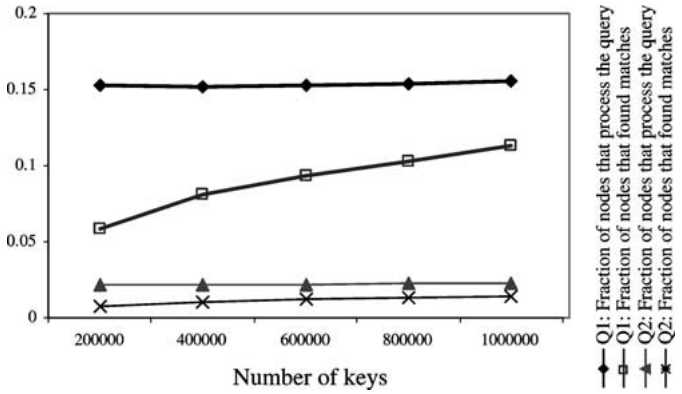

Figure 13. Experiment 2, 3D: Results for query types Q1 and Q2.

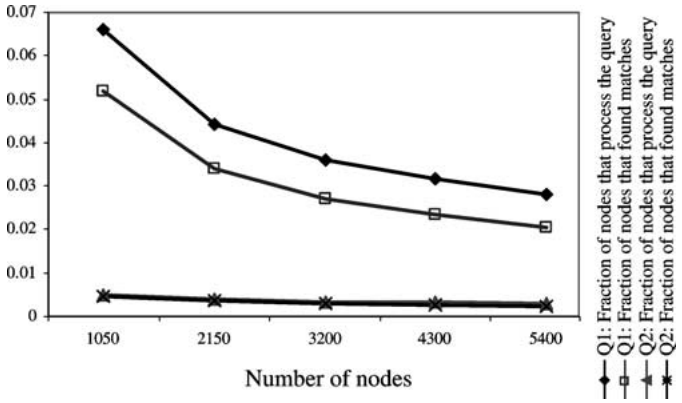

Figure 14. Experiment 3, 2D: Results for query types Q1 and Q2.

4.4. Experiment 3 – increasing system size and constant number of data elements

In the third experiment, the size of the system increases while the numbers of keys and data elements in the system remain constant. In this experiment, the system increases from about 1000–5400 nodes and the number of stored keys is fixed at 10^6. The results for the different query types are presented below. Note that in this case, the number of matches for a query remains the same for different system sizes.

As in the case of experiments 1 and 2, we used a set of queries of type Q1 and Q2 and we measured for each query the number of processing and data nodes. The results were averaged and normalized. Figure 14 presents results for the 2D case, and Figure 15 for the 3D case.

The results are very similar to those presented in Section 4.2 for Experiment 1. With the number of data elements in the system being constant, as the size of the system grows, the clusters associated will be scattered across a larger number of nodes. As a result,

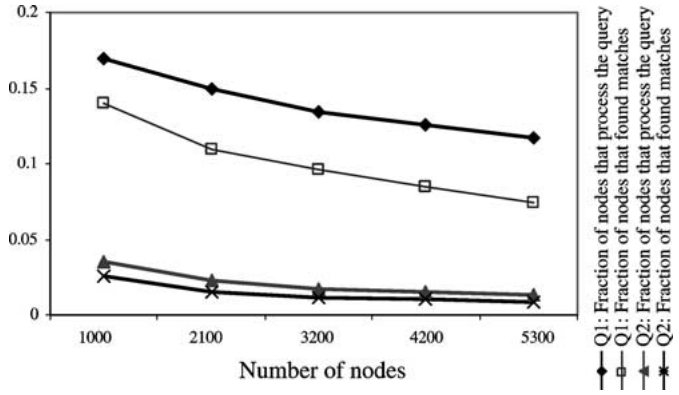

Figure 15. Experiment 3, 3D: Results for query types Q1 and Q2.

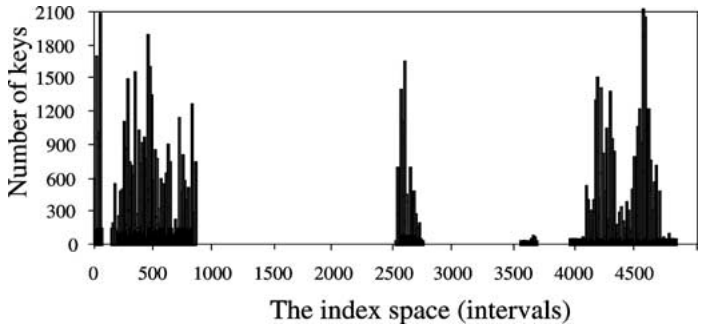

Figure 16. The distribution of the keys in the index space. The index space was partitioned into 5000 intervals. The Y-axis represents the number of keys per interval.

the number of processing nodes and data nodes increases (e.g., in Figure 14, 7% of 1050 nodes is less than 3% of 5400 nodes). However, as Figures 14 and 15 show, the percentage of nodes involved in solving a query decreases as the size of the system increases. This demonstrates that the rate of increase of processing and data nodes is slower than the rate of increase of the system size, indicating that the system is scalable. This also shows that the load balancing algorithms are successful in distributing load.

4.5. 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 16. 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 17. While the resulting load distribution is better than the original distribution in Figure 16, this step by itself does not guarantee a very good load balance. However, when it is used in conjunction

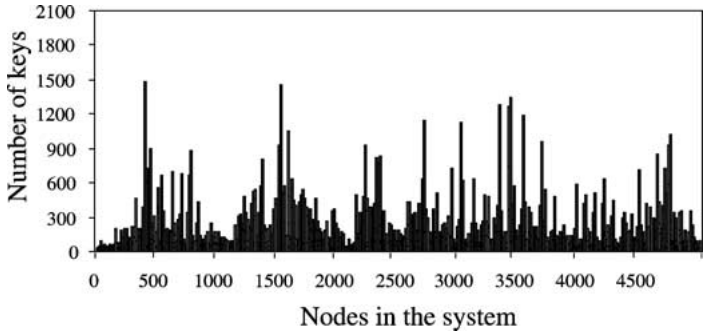

Figure 17. The distribution of the keys at nodes when using only the load balancing at node join technique.

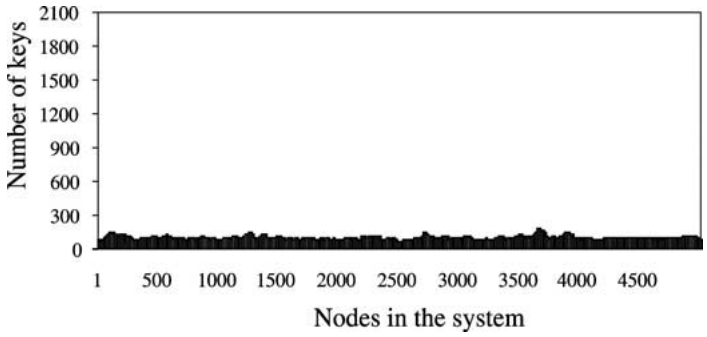

Figure 18. The distribution of the keys at nodes when using both the load balancing at node join technique, and the local load balancing.

with the runtime load-balancing step, the resulting load balance improves significantly as seen in Figure 18. The load is almost evenly distributed in this case.

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 from the data element keyword space to the index space that is used to assign the data elements to peers. This mapping is based on recursive, self-similar SFCs. The recursive structure of the index space is used to optimize query processing and to reduce the number of nodes queried. As the mapping preserves keyword locality, data-elements may not be uniformly distributed in the index space. Dynamic load balancing schemes (applicable at node join and at runtime) were presented.

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. We are extending this research to evaluate other network topologies and mappings, and to address issues such as hot-spots, fault-tolerance, security and resistance to attacks, and maintenance of geographical locality in the overlay network.

Note

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.

References

[1] T. Bially, "A class of dimension changing mapping and its application to bandwidth compression," Ph.D. thesis, Polytechnic Institute of Brooklyn, 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, CA, 2000, pp. 311–320.
[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, 2002, pp. 23–34.
[4] Gnutella webpage, `http://gnutella.wego.com/`
[5] 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*, New York, NY, 2002, pp. 84–95.
[6] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of Hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering* 13(1), 2001, 124–141.
[7] Open Grid Services Architecture (OGSA), `http://www.globus.org/ogsa/`
[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, San Diego, CA, 2001, pp. 161–172.
[9] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems," in *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 2001, pp. 329–350.
[10] H. Sagan, *Space-Filling Curves*, Springer-Verlag, New York, NY, 1994.
[11] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl, "A scalable and ontology-based P2P infrastructure for semantic web services," in *Proceedings of the Second International Conference on Peer-to-Peer Computing (P2P'02)*, Linkoping, Sweden, 2002, pp. 104–111.
[12] K. Sivashanmugam, K. Verma, R. Mulye, Z. Zhong, and A. Sheth, "Speed-R: Semantic P2P environment for diverse Web Service registries," http://webster.cs.uga.edu/∼mulye/SemEnt/Speed-R.html.
[13] 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*, San Diego, CA, 2001, pp. 149–160.
[14] "Universal description, discovery and integration," UDDI Technical White Paper, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000.
[15] B. Yang and H. Garcia-Molina, "Comparing hybrid peer-to-peer systems," in *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, Roma, Italy, 2001, pp. 561–570.
[16] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant widearea location and routing," Technical Report UCB/CSD-01-1141, Computer Science Division, University of California at Berkeley, 2001.