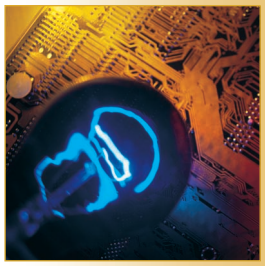

Enabling Flexible Queries with Guarantees in P2P Systems

The Squid peer-to-peer information discovery system supports flexible queries using partial keywords, wildcards, and ranges. It is built on a structured overlay and uses data lookup protocols to guarantee that all existing data elements that match a query are found efficiently. Its main innovation is a dimension-reducing indexing scheme that effectively maps multidimensional information space to physical peers.

Cristina Schmidt and Manish Parashar
Rutgers University

Scalable information discovery in the absence of global knowledge of names or naming conventions remains a fundamental problem in large, decentralized, distributed environments. The heterogeneous nature and large volume of data and resources, their dynamism, and the dynamism of the sharing environment (with various nodes joining and leaving) further compounds the issue. Thus, an ideal information-discovery system should be efficient, fault-tolerant, and self-organizing. Furthermore, it must offer guarantees and support flexible searches.

Recent years have seen increasing interest in peer-to-peer (P2P) information-sharing environments. In the P2P computing paradigm, entities at the network's edge can interact directly as equals and share information, services, and resources without centralized servers. Key characteristics of such systems are decentralization, self-organization, dynamism, and fault-tolerance, all of which make P2P solutions scalable and attractive for information storage and retrieval applications.

This article describes Squid, a P2P information-discovery system that supports complex queries containing partial keywords, wildcards, and ranges (see the "Related Work" sidebar). Furthermore, because it's built on a structured overlay and uses a lookup protocol, Squid guarantees that all existing data elements matching a query will be found with bounded costs in terms of the number of messages and nodes involved.

System Architecture and Design

Squid's architecture is based on a data-lookup system[1,2]; essentially, it imple-

 1089-7801/04/$20.00 Published by the IEEE Computer Society

Related Work

We can classify existing information storage and discovery systems broadly as unstructured, hybrid, or structured.

Unstructured systems, such as Gnutella (http://gnutella.wego.com) and Freenet,[1] support complex queries, including wildcards and ranges, but they don't offer any search guarantees. Rather, they use flooding techniques to process queries. A matching query might not find the information stored in these systems if it is not widely replicated. Hybrid systems, such as Napster (www.napster.com), use centralized directories to provide guarantees, which can limit their scalability.

We can further characterize structured systems as *data-lookup* and *structured keyword* systems. Data-lookup systems[2–5] guarantee that if information exists in the system, peers will find it within a bounded number of hops. These systems build on structured overlays and essentially implement Internet-scale distributed hash tables (DHTs). Information is located using unique and globally known data identifiers; complex queries are not supported. Structured keyword search systems extend data-lookup systems with search capabilities. The Squid system falls into this category. Other approaches that fall in this category include PeerSearch[6] and the work by Reynolds and Vahdat[7] and Andrzejak and Xu.[8]

PeerSearch is a P2P storage system that supports content- and semantics-based searches. It is built on top of CAN[4] and uses the Vector Space Model (VSM)[9] and Latent Semantic Indexing (LSI)[9] to index the documents.

Reynolds and Vahdat propose an indexing scheme for P2P storage systems such as Chord[3] and Pastry.[5] They build an inverted index, which is distributed across the nodes using consistent hashing, and use Bloom filters to reduce bandwidth consumption.

Squid differs from these approaches in that it uses a space-filling curve (SFC)-based indexing scheme to map data elements to peers using keywords. Consequently, when resolving a query, only those data elements that match all the keywords in the query are retrieved. It also supports flexible searching using partial keywords, wildcards, and range queries. Andrzejak and Xu propose a discovery system based on Hilbert SFCs. Unlike Squid, this system uses the inverse SFC mapping, from a 1D space to a *d*-dimensional space, to map a resource to peers based on a single attribute (such as memory). It uses CAN as its overlay topology, with the range of possible values for the resource attribute (1D) mapped onto CAN's *d*-dimensional Cartesian space. This system is designed for resource discovery in computational grids — more specifically, to enhance other resource discovery mechanisms with range queries. In contrast, Squid uses SFCs to encode the *d*-dimensional keyword space to a 1D index space. In this way, we can map and search a resource using multiple attributes.

References

1. I. Clarke et al., "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *Proc. ICSI Workshop Design Issues in Anonymity and Unobservability*, LNCS 2009, Springer-Verlag, 2001, pp. 311–320.
2. C. Plaxton, R. Rajaraman, and A.W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Proc. ACM SPAA*, ACM Press, 1997, pp. 311–320.
3. I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGComm*, ACM Press, 2001, pp. 149–160.
4. S. Ratnasamy et al., "A Scalable Content-Addressable Network," *Proc. ACM SIGComm*, ACM Press, 2001, pp. 161–172.
5. A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms*, LNCS 2218, Springer-Verlag, 2001, pp. 329–350.
6. C. Tang, Z. Xu, and M. Mahalingam, *PeerSearch: Efficient Information Retrieval in Peer-to-Peer Networks*, tech. report HPL-2002-198, HP Labs, 2002.
7. P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *Proc. ACM/IFIP/Usenix Int'l Middleware Conf.*, LNCS 2672, Springer-Verlag, 2003, pp. 21–40.
8. A. Andrzejak and Z. Xu, "Scalable, Efficient Range Queries for Grid Information Services," *Proc. 2nd IEEE Int'l Conf. Peer-to-Peer Computing* (P2P '02), IEEE Press, 2002, pp. 33–40.
9. M. Berry, Z. Drmac, and E. Jessup, "Matrices, Vector Spaces, and Information Retrieval," *SIAM Rev.*, vol. 41, no. 2, 1999, pp. 335–362.

ments an Internet-scale distributed hash table (DHT). The key difference from other systems is the way it maps data elements to the DHT space. (We use the term *data element* to represent a discoverable piece of indexed information – a document, a file, an XML file describing a resource, a Web service, and so on.) Existing systems perform this mapping by using a hashing function that uniformly distributes data elements to nodes. As a result, the data element can be retrieved only if its exact identifier is known. In contrast, Squid uses a dimension-reducing mapping called a *space-filling curve* (SFC).[3] The recursive, self-similar, and locality-preserving properties of SFCs described here enable Squid to support more complex queries using keywords, partial keywords, wildcards, and ranges.

Publishing Data

Publishing data in Squid consists of two steps: constructing the index space and mapping indexes to peers.

Constructing the index space. A data-lookup system deterministically maps data elements to an index space, which is the set of all possible values of an index, by using their unique identifiers. To support keyword searches, we associ-

ate data elements in Squid with a sequence of descriptive keywords and use a mapping function that preserves lexical keyword locality. For P2P storage systems, these keywords are common words in a language such as English; for resource discovery, they are values of globally defined resource attributes.

Keywords in Squid form a multidimensional keyword space in which data elements are points and keywords are coordinates. Think of these keywords as base-n numbers — n can be 10 for numeric terms or 26 for English words, for example. Two data elements are "local" if they're close together in this keyword space — if, for instance, their keywords are lexicographically close (such as *comput*er and *comput*ation) or if they have common keywords. Not all character combinations represent meaningful keywords, which results in a sparse keyword space with nonuniformly distributed data-element clusters. Figure 1 shows examples of keyword spaces.

To efficiently support range queries or queries using partial keywords and wildcards, the index space should preserve locality and be recursive so that these queries can be optimized with successive refinement and pruning. We can construct such an index space using the Hilbert SFC.[3]

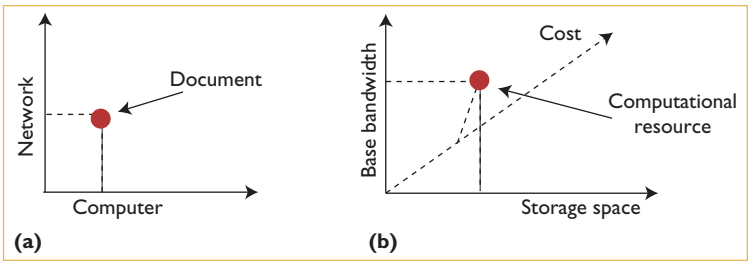

Figure 1. Examples of keyword spaces. (a) A 2D keyword space for a storage system in which the data element "document" is described by the keywords Computer: Network. (b) A 3D keyword space for storing computational resources, using the attributes' storage space, base bandwidth, and cost.

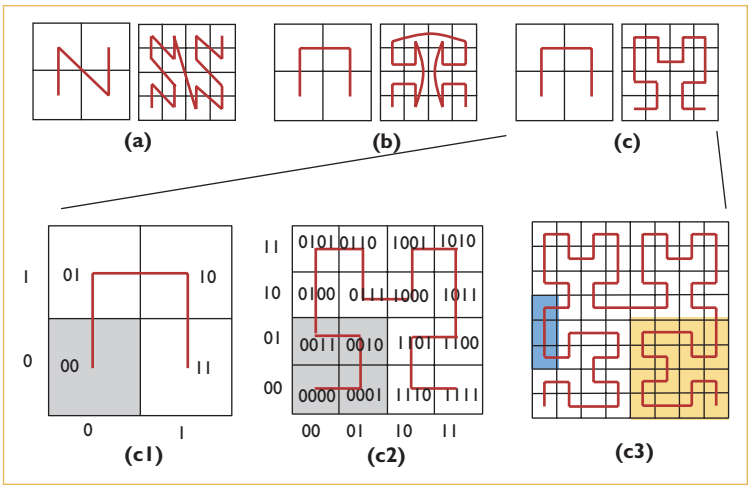

Figure 2. Examples of space-filling curves (SFCs). (a) Morton curve, (b) Gray code, (c) and Hilbert curve. The Hilbert curve approximations are for d = 2, n = 2: (c1) first approximation, (c2) second approximation, and (c3) third approximation. The colored regions in the last panel represent three- and 16-cell clusters.

Space-filling curves. An SFC[3] is a continuous mapping from a d-dimensional space to a 1D space. Think of the d-dimensional space as a d-dimensional cube with the SFC passing once through each point in the cube's volume, entering and exiting the cube only once. Using this mapping, we can describe a point in the cube by its spatial coordinates or by the length along the curve measured from one of its ends.

SFC construction is recursive. We first partition the d-dimensional cube into n^d equal subcubes and then get an approximation to the SFC by joining the centers of these subcubes with line segments such that each cell is joined with two adjacent cells. We similarly fill each subcube using the same algorithm. Next, we rotate and reflect the curves traversing the subcubes such that we can connect them to form a single continuous curve that passes only once through each of the n^{2d} regions. Each refinement of the curve is called an *approximation*. Figure 2 shows three examples of SFCs based on Morton, Gray code, and Hilbert mapping, respectively. Each of these curves imposes a different ordering of the subcubes.

An important property of SFCs is *digital causality*, which follows from the SFC's recursive nature. An SFC constructed at the k-th approximation has an equal portion of its total length contained in each subcube, which means it has n^{k*d} equal segments. If we express distances along the line as base-n numbers, then the numbers that refer to all the points in the subcube and belong to a line segment are identical in their first $(k-1)*d$ digits. In Figure 2c, the subcube (0,0) with SFC index 00 is refined, resulting in the four subcubes also shown in the figure, each with SFC indexes identical in their first two digits.

SFCs are also *locality preserving*. Close points in the 1D space (the curve) are mapped from close points in the d-dimensional space. The reverse property is not true, however: not all adjacent subcubes in the d-dimensional space are adjacent

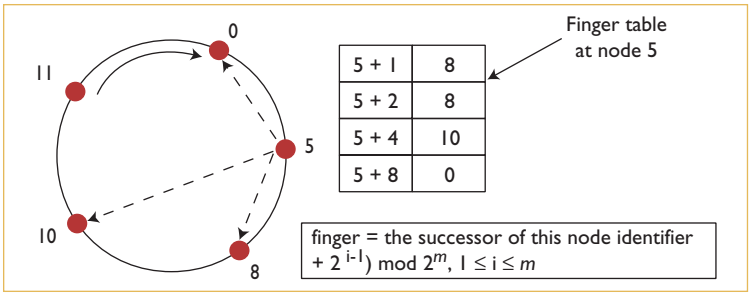

Figure 3. Example overlay network. Each node stores the keys that map to the segment of the curve between itself and the predecessor node.

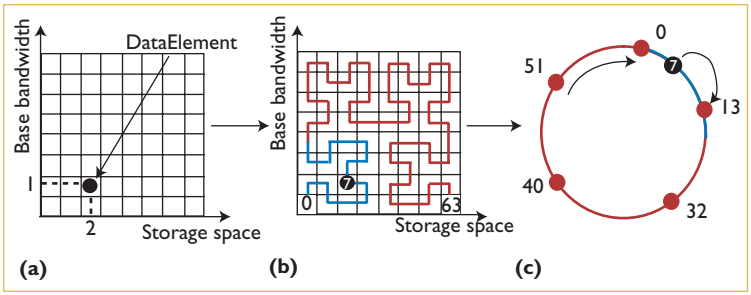

Figure 4. Publishing the data element (2, 1) representing a computational resource with 2 Mbytes of memory and 1 Mbps of bandwidth: (a) the data element (2, 1) is viewed as a point in a multidimensional space; (b) the data element is mapped to the index 7, using the Hilbert SFC; and (c) the data element is stored in the overlay (an overlay with five nodes and an identifier space from 0 to $2^6 - 1$) at node 13, the successor of index 7.

or even close on the curve. A group of contiguous subcubes in d-dimensional space typically will be mapped to a collection of segments on the SFC. These segments are called clusters. (In Figure 2c, the colored regions represent three- and 16-cell clusters.)

In Squid, we use SFCs to generate the 1D index space from the multidimensional keyword space. By applying the Hilbert mapping to this multidimensional space, each data element is mapped to a point on the SFC. Any range query or query composed of keywords, partial keywords, or wildcards can be mapped to regions in the keyword space and the corresponding clusters in the SFC.

Mapping indexes to peers. The next step consists of mapping the 1D index space onto an overlay network of peers. In our current implementation, we use the Chord[2] 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*. Chord uses the finger table for efficient routing and enables data lookup with $O(\log N)$ cost,[2] where N is the number of nodes in the system. Each node constructs its finger table when it joins the overlay and finger tables are updated any time a node joins or leaves the system. The cost of a node joining or leaving is $O(\log^2 N)$.

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. This node is called the key's *successor*. Figure 3 shows an example of an overlay network with five nodes and an identifier space from 0 to $2^4 - 1$.

Publishing a data element in Squid consists of the following steps: attach keywords that describe the data element's content, use the SFC mapping to construct the data element's index, and finally, using this index, store the element at the appropriate node in the overlay (see Figure 4).

Query Engine

The query engine's primary function is to process user queries efficiently. As described earlier, data elements in the system are associated with a sequence of up to d keywords, where d is the keyword space's dimensionality. Queries can consist of a combination of keywords, partial keywords, or wildcards. The expected result is the complete set of data elements that match the user's query — for example, (computer, network), (computer, net*), and (comp*, *) are all valid queries. Another query type is a range query, in which at least one dimension specifies a range. If the index encodes memory, CPU frequency, and base bandwidth resources, for example, the query (256 – 512 Mbytes, *, 10 Mbps – *) specifies a machine with memory between 256 and 512 Mbytes, any CPU frequency, and at least 10 Mbps base bandwidth.

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 complete keywords — no wildcards or ranges — it will be mapped to, at most, one point in the index space; we can locate the node containing the matching data element by using the overlay's lookup protocol. If the query contains partial keywords, wildcards, or ranges (a complex

query), the query identifies a region in the keyword space that corresponds to a set of points in the index space. In Figure 5 (next page), for example, the query (000, *) identifies eight data elements — essentially, 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.

Each cluster can contain zero, one, or more data elements that match the query. Depending on its size, an index space cluster might be mapped to one or more adjacent nodes in the overlay network. A node can also store more than one cluster. Once the query engine at the requesting node identifies the clusters associated with a query, it sends a query message for each cluster. Squid routes a query message for a cluster to the appropriate node in the overlay network as follows. First, the overlay network provides us with a data-lookup protocol: given an identifier for a data element, the lookup mechanism locates the node responsible for storing it. This mechanism can be used to locate the node responsible for storing a cluster by using a cluster identifier, which is constructed by using the SFC's digital-causality property. This guarantees that all the cells forming 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 zeroes. In Figure 6b, the cluster at the top of the tree has prefix 0, and the clusters at the next level have prefixes 00 and 01.

Query optimization. Because the number of clusters can be very high for complex queries, sending a message for each cluster is not a scalable solution. We can make query processing more scalable by capitalizing on the observation that not all clusters corresponding to a query represent valid keywords; the keyword space and clusters are typically sparsely populated with data elements. Filtering out the useful clusters early on can significantly reduce the number of messages sent and nodes queried, but useful clusters cannot be identified at the node where the query is initiated. To solve this, we use the SFC's recursive nature to distribute the process of cluster generation at multiple nodes — for example, to the ones that might be responsible for storing the data matching a query.

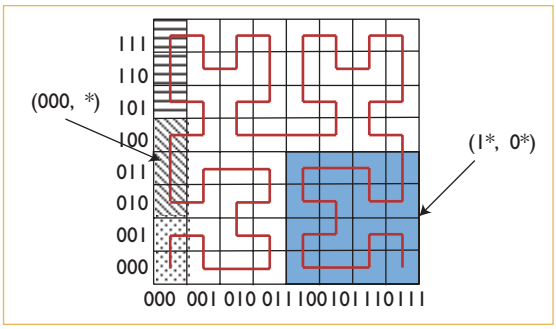

*Figure 5. Regions in a 2D space defined by the queries (000, *) and (1*, 0*). The query (000, *) defines three clusters (the vertical region on left marked by different patterns), and the query (1*, 0*) defines one cluster (the square region on the right).*

Because SFC generation is recursive, and clusters are segments on such curves, these clusters also can be generated recursively. Think of this process as constructing a tree. At each level of the tree, the query defines a number of clusters, which, when refined, result in more clusters at the next level. We now can embed this tree into the overlay network such that the root performs the first query refinement; each node refines the query further, sending the resulting subqueries to appropriate nodes in the system. Query optimization consists of pruning nodes from the tree during the construction phase.

Figure 6 (next page) illustrates this optimization process. Figure 6a shows the successive refinement for the query (011, *) in a 2D space, with base-2 digits as coordinates. Figure 6b shows the corresponding tree, with the tree's leaves representing all possible matches to the query. The tree's leftmost path (solid arrows) and rightmost path (dashed arrows) are embedded in the ring network topology (Figure 6c). The overlay network uses six digits for node identifiers, and the arrows are labeled with the prefix of the cluster being solved.

In Figure 6, we initiated the query at node 111000. The first cluster has prefix 0, so the cluster's identifier will be 000000. Node 111000 sends the cluster to node 000000, which further refines the cluster and generates two subclusters with prefixes 00 and 01. The cluster with prefix 00 remains at the same node, but after processing, node 000000 sends the subcluster 0001 to node 000100. The cluster with prefix 01 and identifier 010000 goes to node 100001 (dashed line), but this cluster will not be refined at node 100001 because the node's identifier is greater than the query's, and all matching data elements for the cluster and its subclusters should be stored at this node.

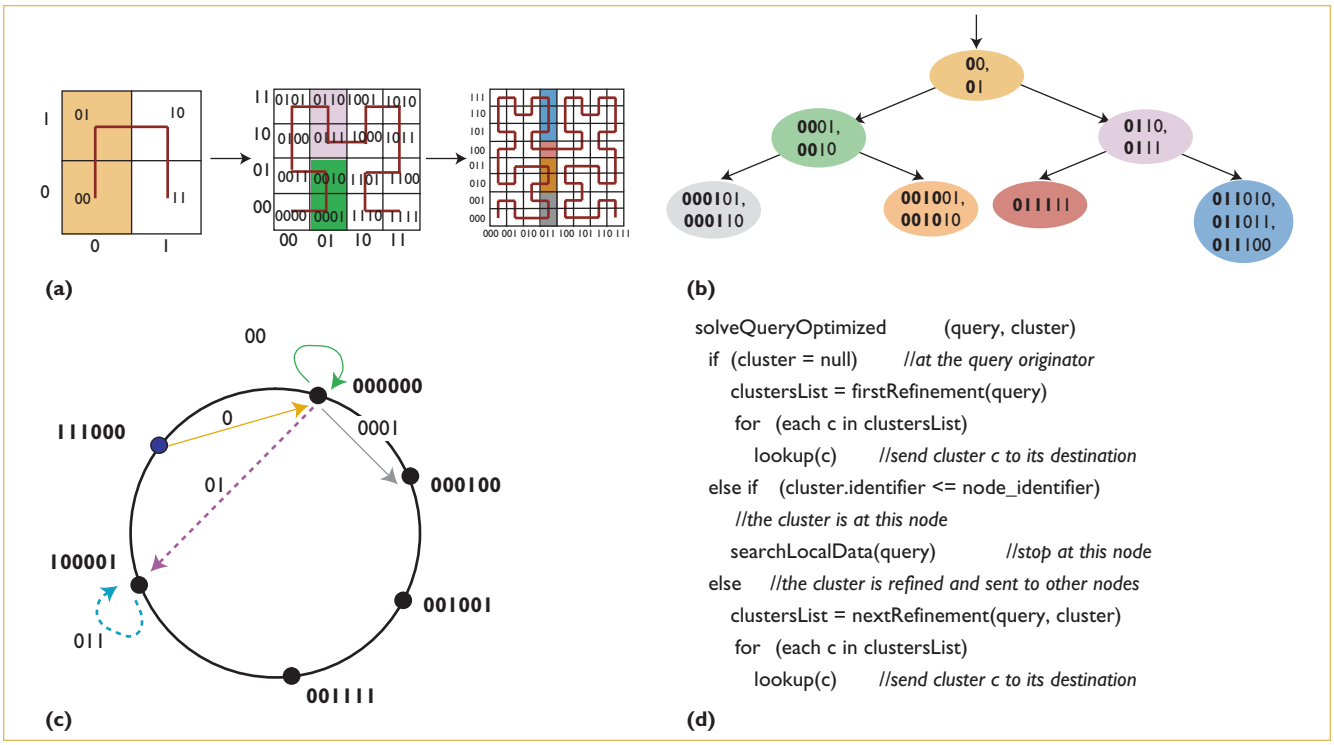

*Figure 6. An example of query processing with optimizations. (a) Recursive refinement of the query (011, *): one cluster on the first-order Hilbert SFC curve, two clusters on the second-order Hilbert curve, and four clusters on the third-order Hilbert curve. (b) Recursive refinement of the query (011, *) viewed as a tree. Each node is a cluster, and the bold characters are clusters' prefixes. (c) Embedding the leftmost tree path (solid arrows) and the rightmost path (dashed arrows) onto the overlay network topology. (d) Pseudocode of query processing at a node.*

Load Balancing

As we mentioned earlier, the original *d*-dimensional keyword space is sparse; data elements form clusters in this space instead of being uniformly distributed in it. The Hilbert SFC-based mapping preserves keyword locality, so the index space will also have this property. However, because the nodes are uniformly distributed in the node identifier space, the load will not be balanced when the data elements are mapped to the nodes. Additional load balancing must be performed; various load-balancing schemes at node join and runtime appear elsewhere.[4]

Experimental Evaluation

We evaluated Squid's performance by using a simulator that implements the SFC-based mapping, the Chord-based overlay network, and the query engine with the query optimizations described earlier. Because the overlay network configuration and operations are based on Chord,[2] maintenance costs are of the same order as in Chord.

Our experiment represents a typical P2P storage system in which the number of keys and data elements in the system increases as the number of nodes increases. In this experiment, system size increases from 1,000 nodes to 5,400 nodes, and the number of stored keys (unique keyword combinations) increases from $2 * 10^5$ to 10^6. Each key can be associated with one or more data elements. We evaluated 2D and 3D keyword spaces with the following types of queries:

Q1: Queries with one keyword or partial keyword: (computer, *), (comp*, *, *)
Q2: Queries with two to three keywords or partial keywords (at least one partial keyword): (comp*, net*), (computer, network, *)
Q3: Range queries
Q3_1: (keyword, range, *)
Q3_2: (range, range, range)

We tested a set of queries of each type; we chose the queries such that the number of matches represented the same fraction of the total data regardless of the system's size (number of nodes) and data quantity. For each query, we measured the number of nodes that process it (refine it and

search for matches) and the number of nodes that found matching data (data nodes). We then averaged and normalized the results.

As Figure 7 shows, the number of processing and data nodes is a small fraction of the total nodes and it increases at a slower rate than the system's size. For a 2D keyword space, the average number of processing nodes is below 8 percent; the number of data nodes is below 5 percent. These percentages decrease as system size increases, demonstrating the system's scalability. The number of data nodes is close to the number of processing nodes, indicating that query optimizations effectively reduce the number of nodes involved. Also, we found that Q2 queries are more efficient than Q1 queries, which we 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 we described earlier, 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 — three 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 with a larger magnitude.

Even under these conditions, the results are quite good. A keyword search system like Gnutella (http://gnutella.wego.com) would have to query the entire network using some form of flooding to guarantee that all the matches to a query were returned. With data-lookup systems such as Chord,[2] we would have to know all the matches a priori and look them up individually.

Conclusions

Squid enables scalable and flexible information discovery in large, decentralized, distributed environments; it also provides search guarantees. We're currently deploying it to support information discovery and sharing among science and engineering research communities. It's also being used for resource discovery, Web service discovery, and semantic messaging. The next step is to research additional query-engine optimizations, topology-aware overlay networks, fault tolerance, and security.

Acknowledgments

The work presented in this article was supported in part by the US National Science Foundation via grant numbers ACI

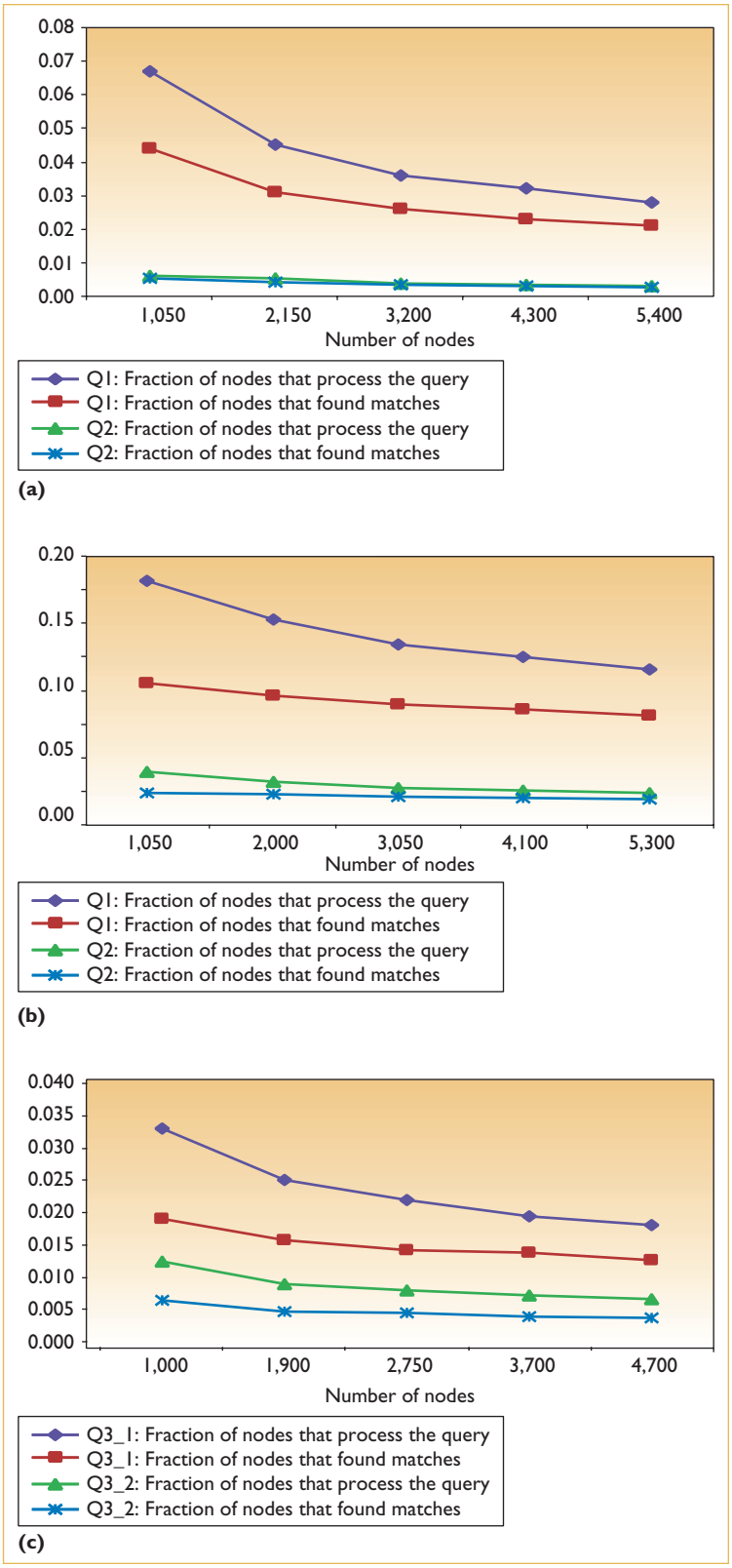

Figure 7. Experimental results. (a) Results for a 2D keyword space for query types Q1 and Q2; (b) results for 3D keyword space for query types Q1 and Q2; and (c) results for 3D keyword space for range queries.

9984357 (CAREERS), EIA 0103674, (NGS) and EIA-0120934 (ITR), and by DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

References

1. S. Ratnasamy et al., "A Scalable Content-Addressable Network," *Proc. ACM SIGComm*, ACM Press, 2001, pp. 161–172.
2. I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGComm*, ACM Press, 2001, pp. 149–160.
3. T. Bially, *A Class of Dimension Changing Mapping and Its Application to Bandwidth Compression*, PhD dissertation, Dept. of Electrical Eng., Polytechnic Inst. of Brooklyn, June 1967.
4. C. Schmidt and M. Parashar, "Flexible Information Discovery in Decentralized Distributed Systems," *Proc. 12th High-Performance Distributed Computing* (HPDC '03), IEEE Press, 2003, pp. 226–235.

Cristina Schmidt is a PhD student in the Department of Electrical and Computer Engineering at Rutgers University. Her research interests include distributed systems and networking — in particular, peer-to-peer systems and self-organizing overlays. She received a BS and an MS in computer science from Babe¸s-Bolyai University, Romania. Contact her at cristins@caip.rutgers.edu.

Manish Parashar is an associate professor in the Department of Electrical and Computer Engineering at Rutgers University, where he also is director of the Applied Software Systems Laboratory. His research interests include parallel and distributed computing, networking, scientific computing, and software engineering. Parashar received a BE in electronics and telecommunications from Bombay University, India and an MS and PhD in computer engineering from Syracuse University. Contact him at parashar@caip.rutgers.edu.