# FLEXIBLE INFORMATION DISCOVERY WITH GUARANTEES IN DECENTRALIZED DISTRIBUTED SYSTEMS

**BY CRISTINA SIMONA SCHMIDT**

**A Dissertation submitted to the**

**Graduate School—New Brunswick**

**Rutgers, The State University of New Jersey**

**in partial fulfillment of the requirements**

**for the degree of**

**Doctor of Philosophy**

**Graduate Program in Electrical and Computer Engineering**

**Written under the direction of**

**Professor Manish Parashar**

**and approved by**

_____

_____

_____

_____

_____

**New Brunswick, New Jersey**

**October, 2005**

## ABSTRACT OF THE DISSERTATION

# Flexible Information Discovery with Guarantees in Decentralized Distributed Systems

**by CRISTINA SIMONA SCHMIDT**

**Dissertation Director: Professor Manish Parashar**

Recent years have seen increasing interest in Peer-to-Peer (P2P) information sharing environments. The P2P computing paradigm enables entities at the edges of the network to directly interact as equals (or peers) and share information, services and resources without centralized servers. Key characteristics of these systems include decentralization, self-organization, dynamism and fault-tolerance, which make them naturally scalable and attractive solutions for information sharing and discovery applications.

The ability to efficiently discover information using partial knowledge (for example queries containing keywords, wildcards and ranges) is an important and challenging issue in large, decentralized, distributed sharing environments such as P2P systems and Computational Grids. Existing P2P information discovery systems implement tradeoffs. Unstructured search systems are easy to maintain and allow complex queries but do not offer any guarantees. Lookup systems offer guarantees but at the cost of maintaining a complex and constrained structure and supporting only search based on exact identifiers. Finally, lookup systems enhanced with keyword searches offer guarantees, but the queries supported are still not expressive enough.

In this research we present an innovative approach to building a P2P information discovery system that provides the flexibility of keyword search systems while providing the guarantees and bounds of data lookup systems. The fundamental concept underlying our approach is the definition of multidimensional information (keyword) spaces

and the maintenance of locality in these spaces. The key innovation is a dimensionality reducing indexing scheme that effectively maps the multidimensional information space to physical peers, while preserving lexical locality. The presented system guarantees that all existing data elements matching a query are found with reasonable costs in terms of number of messages and nodes involved. Complex queries containing partial keywords, wildcards and ranges are supported. The design, analysis, implementation and an experimental evaluation of the system are presented.

# Acknowledgements

I would like to express my thanks to my thesis advisor, Dr. Manish Parashar, for his guidance and support during my research years in the TASSL Lab. He was an ever present driving force that offered advice and encouragement toward the completion of this thesis.

I am grateful to the members of my thesis committee, Dr. Peter Meer, Dr. Deborah Silver, Dr. Ramesh Subramabian and Dr. Yanyong Zhang, for taking time off their busy schedule to read and review my thesis.

I would like to thank all my colleagues in the TASSL Lab, especially Vincent Matossian with whom I discussed my research many times. I would also like to thank all my friends at Rutgers for their emotional support and camaraderie during my years as a PhD student.

I also would like to acknowledge the contributions made to this thesis by Vincent Matossian and Nanyan Zhang for the Meteor project, and Dr. David Foran and Wenjin Chen for the TMA Collaboratory. The collaboration with them was a great experience.

The research presented in this thesis was supported in part by the National Science Foundation via grants numbers ACI 9984357, EIA 0103674, EIA 0120934, ANI 0335244, CNS 0305495, CNS 0426354 and IIS 0430826.

# Dedication

To Riky, my husband and best friend.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Recent years have seen increasing interest in Peer-to-Peer (P2P) environments. The P2P computing paradigm enables participating entities to directly interact as equals (or peers) and share information, services and resources without centralized servers. Key characteristics of these systems include decentralization, self-organization, heterogeneity, large scales, dynamism and fault-tolerance. P2P systems have several advantages including lack of a central authority, no single point of failure and scalability. Several applications constructed around a centralized model have been re-modeled as P2P systems. As a result, the focus of this research is on large scale, decentralized sharing environments. Sample domains include:

- Grid computing is rapidly emerging as the dominant paradigm for wide area distributed computing [6]. Its overall goal is to realize a service infrastructure for enabling the sharing of autonomous and geographically distributed hardware, software, and information resources (e.g., computers, data, storage space, CPU, software, instruments, etc.)

- Scientific communities need to share large volumes of data. Data-sharing is typically across institutions with investigators and data spread over multiple campuses, cities and states and the sharing tends to be ad hoc and opportunistic rather than pre-orchestrated and centrally mediated.

- Web Services are enterprise applications that exchange data, share tasks and automate process over the Internet. They are designed to enable applications to communicate directly and exchange data, regardless of language, platform and location. The environment they operate in is large, dynamic, heterogeneous, and inherently decentralized. An important component of a Web Service architecture

is Web Service discovery. As the number of Web Services is in continuous growth, a scalable and decentralized discovery infrastructure is needed.

These systems present a common set of challenges: large scale, lack of global centralized authority, heterogeneity (resources, sharing policies) and dynamism. A fundamental issue in these large, decentralized, distributed sharing environments is the efficient discovery of information in the absence of global knowledge of content and/or naming conventions. The heterogeneous nature and large volume of data and resources, their dynamism and the dynamism of the sharing environment (with nodes joining and leaving) make information discovery a challenging problem. An ideal information discovery system has to be efficient, fault-tolerant, self-organizing, has to offer guarantees and support flexible searches (using keywords, wildcards, range queries). P2P systems, with inherent properties such as self-organization, fault-tolerance and scalability, offer attractive solutions.

## 1.2 Problem Description

As highlighted above, information discovery in decentralized environments presents significant challenges due to the large scale, dynamism and heterogeneity of these environments. P2P systems can be very large, scaling to thousands of nodes. The volume of information shared in these systems can be very large as well. Furthermore, these systems are very dynamic with nodes arbitrarily joining, leaving and failing. As a result, maintaining global knowledge about the current nodes and the content stored in the system is not feasible, and existing P2P information discovery systems implement design trade-offs. These systems are either easy to maintain and allow very complex queries, but do not offer any guarantees, or offer guarantees but at the cost of maintaining a complex and constrained structure and supporting less expressive queries.

Another factor is the heterogeneity of the system. Not all nodes have the same capabilities (e.g., storage, processing power, bandwidth), and they may choose to share only a fraction of their resources. These capabilities have to be taken into account while distributing content across the network.

Existing information discovery systems can be broadly classified as unstructured or structured, based on the underlying network of nodes. While unstructured systems are relatively easy to maintain, can support complex queries, and have been used for information discovery [29, 36], they do not guarantee that all matches to a query in a practical-sized system will be found. Hybrid P2P systems [80] provide search guarantees by using centralized directories, which in turn can limit their scalability and may not always be feasible. Structured data lookup systems [48, 63] also provide search guarantees and bound search costs, but at the cost of maintaining a rigid overlay structure. Furthermore, these systems only support lookup using unique identifiers (e.g., filenames). Structured data lookup systems can be extended to support keyword searches by replacing their hashing mechanism with a more flexible indexing scheme. One simple way is to construct an inverted index on top of a structured overlay, which enable keyword searches. A more sophisticated index, one that preserves data locality, can be used to support range queries.

## 1.3 Problem Statement

The overall goal of this research is to design, implement and evaluate a scalable and efficient P2P system that enables flexible information discovery with guarantees in large decentralized distributed environments. The specific objectives are as follows:

- Formulate an indexing mechanism that preserves locality and supports flexible queries containing keywords, partial keywords, wildcards and ranges.
- Develop a distributed query engine that takes advantage of the system structure and indexing mechanism to efficiently resolve queries.
- Analyze the behavior of the locality-preserving indexing mechanism and the performance of the query engine.
- Design cost-effective dynamic load balancing algorithms that can be used to evenly distribute load in the system.

- Address fault-tolerance by introducing redundancy into the system, at the overlay layer, and replicating the index.

- Validate the performance of the P2P system and demonstrate its scalability and efficiency.

- Develop prototype applications.

## 1.4 Overview

In this thesis we present Squid, a P2P information discovery system that supports complex queries containing partial keywords, wildcards and range queries. It guarantees that all existing data elements that match 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 overall architecture of Squid is a distributed hash table (DHT), similar to typical data lookup systems [48, 63]. The key difference is the way data elements[1] are mapped to the index space. In existing systems, this is done using consistent hashing, which uniformly maps data element identifiers to indices in the index space. 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 Squid all data elements are described using a sequence of keywords (e.g., common words in the case of P2P file sharing systems, or values of globally defined attributes - such as memory and CPU frequency - for resource discovery in computational grids). These keywords form a multidimensional keyword space where the keywords are the coordinates and the data elements are points in the space. A locality-preserving mapping, derived from a family of mappings called Space Filling Curves (SFC) [6, 8, 53] is

[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 a document, a file, an XML file describing a resource, an URI associated with a resource, etc.

used to map the multidimensional keyword space to a one-dimensional index space. In the current implementation, we use the Hilbert SFC [6, 8, 53] for the mapping, since it has been shown to have the best locality preserving property [41], and Chord [63] for the overlay network topology.

Squid takes advantage of the recursive nature of SFCs to design an efficient query engine. The query engine decentralizes query resolution, allowing the querying process to proceed in parallel at multiple nodes, and typically the nodes that store matching data elements for the query being resolved. The optimization is based on successive refinement and pruning of queries that significantly reduces the number of messages involved in the querying process.

Unlike the consistent hashing mechanisms, SFCs 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.

Squid addresses the problem of fault tolerance by building redundancy into the overlay. Specifically, it defines a two-tier overlay where nodes form small groups in addition to being part of the global overlay. The index is replicated in the groups of nodes, making the system more reliable.

### 1.4.1 Contributions

The primary contribution of this research is a P2P information discovery system based on a dimension reducing indexing scheme that enables queries containing partial keywords, wildcards, and range queries, while guaranteeing that all existing data elements matching a query will be found with reasonable bounded costs in terms of number of messages and queried nodes. Key components of this research are listed bellow:

- Design of a DHT-based P2P system, using a locality preserving indexing mechanism based on SFCs, that enable flexible queries (keywords, partial keywords,

wildcards and ranges) with guarantees.

- Design of a distributed query engine, that takes advantage of the SFCs properties to minimize the number of messages sent for a query.
- An analytic foundation for the behavior of the SFC-based index within Squid.
- Design of a two-tier overlay, constructed for fault-tolerance and reliability.
- Implementation of simulators for Squid, to demonstrate the scalability of the system.
- Prototype implementation of Squid on top of the JXTA peer-to-peer framework.
- Integration of Squid with several applications.

Squid has been used to index and locate content in P2P sharing systems (using keywords), as a complement for current resource discovery mechanisms in Computational Grids (to enhance them with range queries) and for Web Services discovery. Also, it has been used as a content-based routing infrastructure (middleware) for decentralized coordination spaces and for rendezvous-based systems built for content-based decoupled interactions in pervasive environments.

## 1.5 Impact of the presented Approach - Application Domain

Squid can be used for flexible information discovery and content-based routing in different application domains. A selection of potential applications is highlighted below.

### 1.5.1 P2P Information Sharing Systems

P2P information sharing systems enable sharing of information in a network of peers. These systems enable the end-users to interact directly, and share content (e.g., music, software, data). The existence of these systems has been made possible by recent advances in information technology (storage, computational power and bandwidth) and the associated drops in costs. All participants in the system are equal: they all share content stored locally, request content shared by others and participate in resolving

queries issued by other participants. A key component of a sharing system is the search engine, that must support searches based on complex queries (e.g., keywords, wildcards, ranges) and provide some guarantees as to the completeness of the search results.

A similar sharing environment is suited for sharing large collections of scientific data. In this case the participants are not end-users but research centers, universities, hospitals, etc. Data ownership issues are important, so only metadata is shared. The rest of the requirements remain the same.

Squid satisfies these requirements. Content described using keywords can be indexed and stored in the P2P network so that locality is maintained. It can then be queried using keywords, partial keywords and wildcards. The search is guaranteed to find all existing matches with bounded costs.

### 1.5.2 Resource Discovery in Computational Grids

Grid Computing is rapidly emerging as the dominant paradigm of wide area distributed computing [23]. Its overall goal is to realize a service-based infrastructure for enabling the sharing of autonomous and geographically distributed hardware, software, and information resources (e.g., computers, data, storage space, CPU, software, instruments, etc.) among user communities.

Grids present similar challenges to P2P systems: large scale, lack of global centralized authority, heterogeneity (resources, sharing policies) and dynamism [29]. A fundamental problem in these systems is the discovery of resources mentioned above. Squid can be used to complement current resource discovery mechanisms in Computational Grids, by enhancing them with range queries. Computational resources can be described in multiple ways (e.g., as URI's or XML files). Typically, they have multiple attributes such as memory and CPU frequency. The values of globally defined attributes of computational resources are considered keywords. These keywords are used to index the descriptions of the computational resources and the indices are stored at nodes in Squid.

### 1.5.3 Web Service Discovery

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 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 [82]) 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 approaches that have been proposed. These systems (e.g., [55]) build on P2P technologies and ontologies to publish and search for Web Service descriptions.

Squid is suited for building dynamic, scalable, decentralized registries with real-time and flexible search capabilities, to support Web Service discovery. Web Services can be described for example using WSDL, and can be characterized by a set of keywords. These keywords can then be used to index the Web Service description files, and store the index at peers in the P2P system.

### 1.5.4 Discovery of Newsgroups in Bulletin Board News Systems

Bulletin Board Systems (BBSs) are world-wide distributed discussion systems. A BBS consists of a set of *newsgroups*. Newsgroups are typically classified hierarchically by subject and contain collections of messages with a common theme. Users can discover and connect to the newsgroups (e.g., interest groups), subscribe to and publish messages on the newsgroups. Usenet is one of the earliest BBSs. Recently the number of BBSs has grown and they have become a popular means of communication.

Squid can be used to discover interest groups in a flexible manner. A newsgroup is indexed based on keywords, where the keywords are the subdomains in its hierarchical name.

### 1.5.5 Content Routing Infrastructure

Routing is traditionally done using globally known addresses that have nothing in common with the content of the message being sent. Since a node in a P2P overlay network resides at the application layer, and implements a routing mechanism, the P2P overlays can be used for routing based on content. Squid's publishing and querying mechanisms can be regarded as content-based routing, since the decision about where to forward the message is based on the set of keywords describing the data element to be stored, or the query.

Squid has been used to support content-based routing in Meteor [31] and Rudder [37]. Meteor [31] is a middleware that enables decoupled interactions in pervasive environments. Rudder [37] is a distributed coordination space.

## 1.6 Thesis Outline

The material presented in this thesis is structured in seven chapters as outlined below.

Chapter 2 presents an overview of existing information discovery P2P systems. A comparison between these system and Squid is also presented.

Chapter 3 describes the architecture and operation of Squid. This chapter describes the Hilbert SFC generation and how it is used to index the data, the overlay network, query mechanism with optimizations, and the load balancing techniques. Simulation results are also presented.

Chapter 4 gives a theoretical and experimental evaluation of the search characteristics of Space Filling Curve-based indexing within Squid, and the performance of the query engine.

Chapter 5 addresses the reliability and fault tolerance of Squid. The reliability and fault tolerance are addressed at two levels, the overlay and routing infrastructure, and

the stored information. Redundancy is built into the overlay, and the index is replicated at nodes. A two-tier overlay is designed, and evaluated.

Chapter 6 presents two applications based on Squid, a TMA Collaboratory [59] and Meteor [31].

Conclusions and future work directions are presented in Chapter 7.

# Chapter 2

# Background and Related Work

This chapter investigates the current landscape of Peer-to-Peer information discovery systems. An overview of P2P information discovery systems is presented, followed by a classification of existing systems based on their design and target applications. Specifically we present and compare data lookup and search systems. Finally we introduce Squid and highlight its contributions with respect to the existing systems.

## 2.1 Peer-to-Peer Systems

Recent years have seen an increasing interest in the Peer-to-Peer (P2P) computing paradigm where entities at the edges of the network can directly interact as equals (or peers) and share information, services and resources without centralized servers. While the P2P architecture is not new (the original Internet was conceived as a P2P system), the proliferation of the World Wide Web and advances in computation and communication technologies have enabled the development and deployment of a large number of P2P applications. These applications take advantage of resources such as storage, computing cycles and content available at user-machines at the edges of the Internet. Properties such as decentralization, self-organization, dynamism and fault-tolerance make them naturally scalable and attractive solutions for applications in all areas of distributed computing and communication. Furthermore, as communications are peer-to-peer and take place at the edges of the network, network congestion is reduced. Usenet and DNS represent an earlier generation of successful applications with P2P characteristics. Recent studies have shown that P2P applications have a significant and growing impact on Internet traffic [38]. More recent P2P applications include content sharing and distribution systems such as Napster [80] and Gnutella [73], communication and collaboration systems such as Jabber [76], Groove [74] and ICQ [75], systems for anonymity such as Freenet [12], censorship-resistant systems such as Publius

[66] and embarrassingly parallel computing such as SETI@home [81]. Furthermore, P2P systems have been recently proposed for resource discovery in Computational Grids [1, 22, 29, 36], and for Web Service Discovery [55].

## 2.2 Peer-to-Peer Information Discovery Systems

An important area where P2P technologies continue to be successfully applied is information discovery. The key strengths of the P2P architecture, such as the ability to harness peer resources, self-organization and adaptation, inherent redundancy and fault-tolerance and increased availability, make it an attractive solution for this class of applications.

P2P information discovery systems enable information sharing in a network of peers. The "network" of peers is typically an overlay network, defined at the application layer of the protocol stack (TCP/IP, ISO/OSI). A peer can have multiple roles. It can be a server (it stores data and responds to query requests), a client (it requests information from other peers), a router (it routes messages based on its routing table), and a cache (it caches data to increase the system's availability). Note that these characteristics describe pure P2P systems. In hybrid systems a peer may have fewer roles, some peers are servers, routers and caches, and others are only clients.

Existing P2P systems can be classified, based on the overlay network of peers employed and based on their capabilities, as systems for data lookup and systems for searching. Such a classification is summarized in Table 2.1. The classes are described below.

## 2.3 Data Lookup Systems

Data lookup systems guarantee that if information exists in the system, it will be found by the peers within a bounded number of hops. These systems build on structured overlays and essentially implement Internet-scale Distributed Hash Tables (DHT).

| P2P SYSTEM | | PROS & CONS | EXAMPLES | APPLICATIONS |
|---|---|---|---|---|
| **Structured Data Lookup**: Use structured overlay networks and consistent hashing to implement Internet-scale distributed hash tables. Data placement and overlay topology highly structured. Data lookup only using unique data identifiers (e.g. filenames). | | **Pro:** Provide efficient lookups with guarantees and bounded search costs. **Cons:** Complex queries (keywords, wildcards, ranges) not supported. High structured overlay maintenance costs. | Chord [63], CAN [48], Pastry [51], Tapestry [70] | Storage systems: Past [18], CFS [16], OceanStore [33]. Content distribution systems: Scribe [52], Coral [24]. |
| **Structured Keyword Search** | **Distributed Inverted Indexes**: Use structured overlay networks to implement Internet-scale distributed hash tables. Extend the data lookup protocol with a distributed inverted index to support keyword searches. | **Pro:** Support keyword searches with guarantees and bounded costs. **Cons:** Each data element is individually mapped and searched for each keyword - keyword searches can be inefficient. Do not support wildcards, partial keywords and range queries. High structured overlay maintenance costs. | pSearch [65], Reynolds and Vahdat [49] | Information retrieval systems: pSearch [65]. Search mechanisms for DHT-based file systems: Reynolds and Vahdat [49] |
| | **Locality-preserving Index**: Use structured overlay networks to implement Internet-scale distributed hash tables. Extend the data lookup protocol with a locality-preserving indexing scheme. Use explicit load balancing algorithms to balance information distribution among peers. | **Pro:** Can support complex queries (e.g. keywords, wildcards, ranges) with guarantees and bounded search costs. **Cons:** High structured overlay maintenance costs. | Squid [56], Andrzejak and Xu [1], Mercury [5], Place Lab [10], Ganesan et. al. [25] | Resource discovery in Grid: Squid [56], Andrzejak and Xu [1]. Web Service discovery: Schlosser et. al. [55]. Scientific data sharing: Schmidt et. al. [59]. Content routing: Meteor [31], Place Lab [10]. |
| **Unstructured**: Gnutella [73] like systems. Use flooding techniques to process queries by forwarding them to neighboring peers. Flooding algorithms used include "limited-scope flooding", "expanding ring search", "random walks" [38], "best neighbor", "learning based" [69] and "routing indices" [15]. Data replication is used to improve the number of results. | | **Pro:** Overlay networks are easy to maintain. Can support complex queries and different querying systems. **Cons:** No search or cost guarantees. All matching information for a query may not be found in a practical-sized system. | Gnutella [73], Freenet [12] Iamnitchi et. al. [29], Li et. al. [36] | Community information sharing applications: Gnutella [73], Freenet [12]. Resource discovery in Grid: Iamnitchi et. al. [29], Li et. al. [36]. |
| **Hybrid**: Unstructured systems using centralized directories to address search guarantee problem. | | **Pro:** Can support complex queries. **Cons:** Limited scalability. May not be feasible for very large systems. | Napster [80], Morpheus [79], Mesh [64] | Community information sharing applications: Napster [80]. Coorporate resource sharing: Mesh [64]. |

Table 2.1: A classification of existing P2P systems.

### 2.3.1 Structured Overlays of Peers

The nodes[1] in a typical P2P system self-organize into an overlay network at the application level. Search systems such as Gnutella use an unstructured overlay that is very easy to maintain. However, these systems do not scale well as they use a flooding-based search mechanism, where each node forwards the query request to all or some of its neighbors. Furthermore, systems built on unstructured overlays do not guarantee that all existing information that matches a query will be found. Structured overlay networks address these issues.

In structured overlays, the topology of the peer network is tightly controlled. The nodes work together to maintain the structure of the overlay in spite of the dynamism of the system (i.e., nodes dynamically join, leave and fail). Maintaining the overlay structure can lead to non-negligible overheads. However, data lookup in these systems is very efficient, and can be guaranteed. For example, the Chord [63] system organizes nodes in a ring. The cost of maintaining the overlay when a node joins or leaves is $O(\log^2 N)$ in this case, where N is the number of nodes in the system. In addition to this cost, each node periodically runs checks to verify the validity of its routing tables, and repairs them if necessary. Similarly, CAN [48] organizes nodes in a $d$-dimensional toroidal space. Each node in CAN is associated with a hypercube region (zone), and has as its neighbors the nodes that "own" the adjacent hypercubes. The cost of a node join is $O(d)$, and a background algorithm that reassigns zones to nodes makes sure that the structure of the overlay is maintained in case of node failures.

### 2.3.2 Distributed Hash Tables (DHTs) and DHT Routing

A hash table is a natural solution for a lookup protocol: given an identifier (a key), the corresponding data is fetched. Data stored in hash tables must be identified using unique numeric keys. A hashing function is used to convert the data identifier into a numeric key. In a P2P lookup system the hash table is distributed among the peers, each peer storing a part of it. Consistent hashing [32] is typically used to ensure a

[1] In this thesis we use "node" and "peer" interchangeably

uniform distribution of load among the peers.

As DHTs build on structured overlays, the placement of data and the topology of the overlay network are highly structured and tightly controlled, allowing the system to provide access guarantees and bounds. Since the mapping of data elements to indices in the index space is based on unique data identifiers (e.g., filenames), only these identifiers can be used to retrieve the data. Keyword searches and complex queries are not efficiently supported by these systems. Examples of data lookup systems include Chord [63], CAN [48], Pastry [51], and Tapestry [70]. These systems have been used to construct persistent storage systems, cooperative distributed file systems, distributed name systems, cooperative web caches, and multicast systems. For example, PAST [18] (a persistent distributed storage system) is built on Pastry and CFS [16] (a cooperative distributed file system) builds on Chord. However, the lack of support for keyword searches can significantly limit the applicability and scalability of these systems, as the data identifiers have to be unique and globally known.

A DHT essentially implements one operation, lookup(key), which routes the request to the peer responsible for storing the key. The routing algorithm depends on the overlay used, and its efficiency has a direct impact on the scalability of the system. Each node in a DHT-based P2P lookup system has a unique numeric identifier which is a string of bits of a specified length. The data keys are also strings of bits of the same length. Furthermore, each node in the system maintains a routing table containing pointers to a small number of other nodes. When a node receives a query for a key that is not stored locally, it routes the query to the node in its routing table that places the query "closest" to the destination. "Closeness" is measured differently from system to system, and it is typically a function of the identifier of the current node and the key. Selected P2P DHT systems are summarized below.

**Plaxton** *et. al.* **[46]**

Plaxton *et al.* [46] developed an algorithm for accessing replicated shared objects in a distributing system. In the Plaxton mesh, each node is assigned a unique integer label. Furthermore, each object (data) has a unique identifier and is replicated at multiple

nodes in the system. The node whose identifier shares the longest prefix with the object identifier is considered the "root" for that object. Object replicas form a virtual height-balanced tree, which is embedded one-to-one into the overlay. Each node of the tree keeps information associated with the copies of the object residing in its subtree.

Each node in the system maintains a routing table and a pointer list. The pointer list contains pointers to copies of some objects in the network. The pointer list is updated only when an object is inserted or deleted. An insert request for object $A$ originated at node $x$ will update the pointer lists at nodes whose labels share a prefix subsequence with $A$'s identifier.

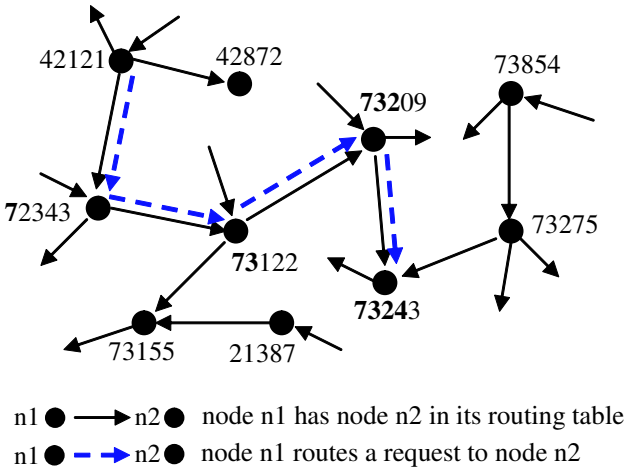



Figure 2.1: Example of routing from node 42121 to node 73243 in a Plaxton mesh.

The Plaxton routing algorithm is based on prefixes. In this algorithm, a single digit of the destination's identifier is resolved at a time. For example, if the node 72343 receives a request with a key **7**3241 (the key and the node identifier have the first digit in common), the node will route the message to a neighbor that shares the first two digits with the key (for example, **73**122). Figure 2.1 illustrates the routing of a message from node 42121 to node 73243. To enable this routing, each node has to maintain a routing table organized into routing levels, with each level pointing to nodes that match a prefix of its own identifier and each level increasing the length of the common prefix. For example the node 72343 will have the following nodes in its routing table: (level0: 0..., 1..., 2..., etc.), (level1: **7**0..., **7**1..., **7**2..., etc.), (level2: **72**0..., **72**4..., **72**5..., etc.), ..., and so forth. The size of the routing table at each node is O(log N), where N is the

number of nodes in the system, and a data lookup requires at most O(log N) hops.

### Tapestry

Plaxton's algorithm [46] assumes a relatively static node population, which makes it unsuitable for P2P systems. Tapestry [70] is a P2P overlay routing infrastructure that uses a variant of the Plaxton algorithm that is more appropriate for a dynamic node population. The basic routing mechanism in Tapestry follows the Plaxton algorithm described above. Each node maintains a neighbor table, organized into routing levels, and a backpointer list that points to nodes where the current node is a neighbor. The references to objects are stored in the same way as in the Plaxton system.

Additionally, Tapestry addresses fault tolerance using a soft-state approach: (1) heartbeat messages are exchanged to discover failed nodes, and a failed node is given a second chance: it is not removed from the routing tables of other nodes until after a period of time; (2) each object has multiple roots (the single root in the Plaxton scheme is a single point of failure); and (3) references to objects expire and have to be renewed periodically, so eventually there are no references that point to nonexistent objects.

The algorithm maintains the routing table size of the order of O(log N), and the routing cost O(log N), where N is the number of nodes in the system.

### Pastry

In Pastry [51] each peer node has a unique identifier from a circular index space, ranging from 0 to $2^{128}$-1. The unique node identifiers are randomly generated at node join and are uniformly distributed in the identifier index space. Data elements are mapped to nodes based on unique identifiers called keys. A data element is mapped to the node whose identifier is numerically closest to its key. For example, in Figure 2.2, data with key 65923 is mapped to node 65980. Similarly, data with key 65211 would be mapped to node 65017.

Each node in Pastry maintains a routing table, a neighborhood set, and a leaf set. The routing table is organized into levels (rows) similar to the Plaxton approach. The neighborhood set maintains information about nodes that are closest to the local nodes

according to a proximity metric (for example, number of IP hops). The leaf set is the set of nodes with identifiers that are numerically closest, half larger and half smaller than the identifier of the current node.

The routing algorithm is prefix based. The Pastry node routes a message to the node in its routing table with an identifier that is numerically closest to the destination key, for example, it has the longest shared prefix with the key. Figure 2.2 illustrates the routing process.

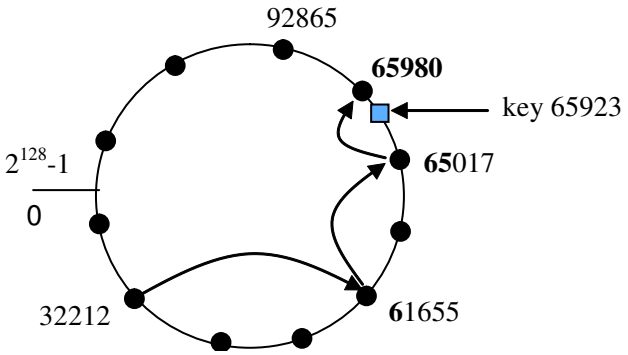



Figure 2.2: Example of routing in Pastry. The figure illustrates routing from node 32212 to the node whose identifier is closest to the key 65923.

The size of a routing table in Pastry is O(log N), where N is the number of active Pastry nodes. The expected number of routing hops is O(log N). Pastry attempts to minimize the distance traveled by the message by taking network locality into account.

### Chord

In the Chord overlay network [63] 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 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.

Data elements are mapped to nodes based on their keys (identifiers). A data element is mapped to the first node whose identifier is equal to or follows its key. This node is called the successor of the key. Consider a sample overlay network with 6 nodes and an

identifier space from 0 to $2^5$-1, as shown in Figure 2.3. In this example, data elements with keys 6, 7, 8, and 9 will map to node 9, which is the successor of these keys. A node forwards requests to the neighbor that is closest to the key.

The size of the routing table is $m$, a lookup requires O(log N) hops, where N is the number of nodes in the system.

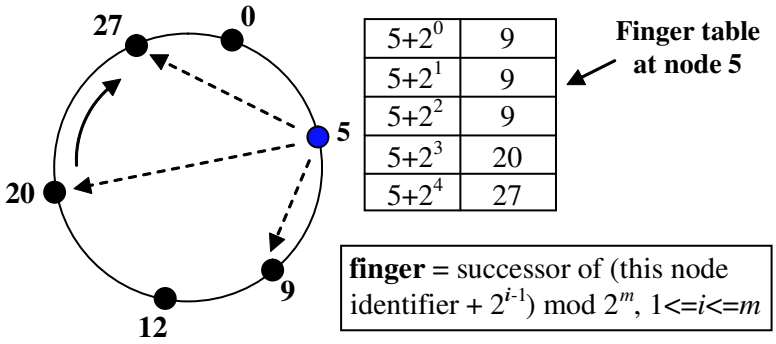



Figure 2.3: Example of Chord overlay network. Each node stores the keys that map to the segment of the curve between itself and its predecessor node.

### CAN

CAN [48] uses a $d$-dimensional Cartesian coordinate space on a $d$-torus. This coordinate space is dynamically partitioned among all the nodes in the system. Each node is associated with a zone in this coordinate space, and has as neighbor nodes that "own" adjacent zones. Figure 2.4 illustrates a system with seven nodes using a two-dimensional coordinate space.

Each data element is deterministically mapped to a point in this coordinate space using a uniform hash function. The data element is stored at the node that owns the zone in which the point lies. For example, in Figure 2.4, the data element mapped to point P is stored at node n2.

Routing consists of forwarding the request to a neighbor that is closest to the key. Figure 2.4 illustrates routing from node n3 to node n2. In CAN, nodes have O($d$) neighbors and the data lookup cost is O($d\mathrm{N}^{\frac{1}{d}}$), where N is the number of nodes in the system.

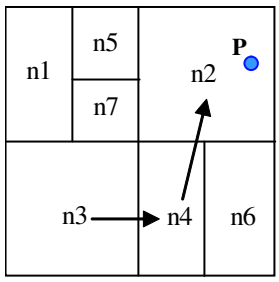



Figure 2.4: Example of CAN overlay network with 7 nodes. Data mapped to point P is stored at node n2. The figure illustrates routing from node n3 to node n2.

### Kademlia

Kademlia [39], like the systems presented above, uses identifiers for nodes and keys for data, both taken from a $b$-bit identifier space (in this case $b = 160$). The (key, data-value) pairs are stored at nodes with identifiers close to the key. The routing algorithm uses node identifiers to locate the node that stores a given key. A key innovation in Kademlia is the use of the XOR metric (a symmetric, non-Euclidean metric). Kademlia treats nodes as leaves in a binary tree, where each node's position is determined by the shortest unique prefix of its identifier. Any node in the system views the tree as a set of successively lower subtrees that do not contain the node. The node must know about some other node in these subtrees. Figure 2.5 illustrates the system from the node with prefix 1011's perspective.

The routing table at each node consists of $b$ $k$-buckets (i.e., 160 $k$-buckets). Each $k$-bucket keeps a list of $k$ nodes at a distance between $2^i$ and $2^{i+1}$ from the current node, where $0 \leq i \leq 160$. $k$ is a system-wide replication parameter chosen such that any given $k$ nodes are very unlikely to fail within an hour of each other. The entries in a bucket are ordered based on "last seen" time; the last recently seen nodes at the head and the most recently seen nodes at the tail. The buckets are updated when the node receives a message (request or reply) from another node. The last recently seen nodes have priority if they are still alive; they are not replaced by most recently seen nodes if the bucket is full. This policy is based on the fact that the longer a node has been up, the more likely it is to remain up for another hour [54]. A (key, data-value) pair is stored at $k$ nodes in the system that are closest to the key according to the XOR

metric. To find a data-value based on a key, a node initiates a recursive procedure to locate the $k$ closest nodes to the key. The initiator sends lookup messages to $\alpha$ nodes from its closest $k$-bucket. These nodes respond with at most $k$ known nodes that are closest to the destination. The initiator picks some of these nodes and sends them lookup requests. The process continues until the nodes with the data are located.

The (key, data-value) pairs need to be republished periodically to ensure their persistence in the system. Nodes that hold a copy can fail or leave the system, and/or other nodes with identifiers closer to the key can join the system. When a node joins the system, it must store all (key, data-value) pairs for which it is one of the $k$ closest nodes. These values are provided by the nodes that learn about the joining node.

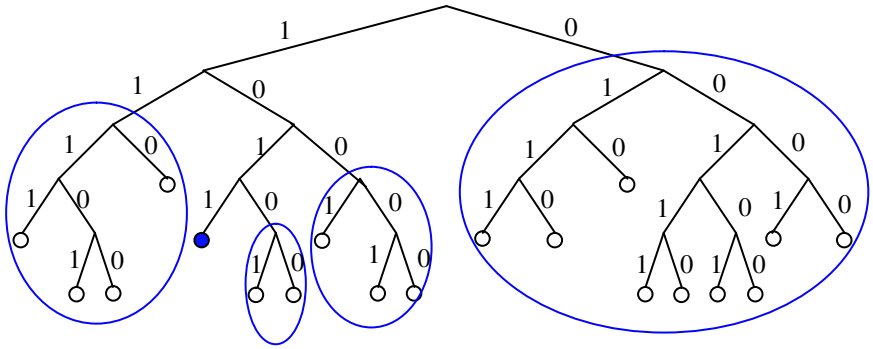


Figure 2.5: Kademlia binary tree. The leaves are nodes in the system; the black dot represents the node that has prefix 1011. The circles show the subtrees where the node with prefix 1011 must know some other node.

As for other DHT-based systems, data lookup takes O(log N), and the request is forwarded to a neighbor that is closest to the destination using the XOR metric to measure the closeness. While existing data is not guaranteed to be found, there is a high probability that any lookup is successful. Kademlia is the only DHT-based lookup system without strong guarantees. The other DHT-based systems guarantee that a piece of information existing in the system will be found.

## 2.4 Search Systems

Search systems allow complex keyword queries using partial keywords, wildcards and ranges. P2P systems that support data searches can be classified based on their overlay

topology as unstructured systems, which provide *loose guarantees*, structured systems, which provide *strong guarantees*, and hybrid server-based systems. Unstructured search systems are composed of Gnutella-type [73] systems. Structured search systems are relatively recent, and are built on top of data lookup systems. Selected existing search systems are presented bellow.

### 2.4.1 Hybrid Search Systems

Hybrid P2P systems, such as Napster [80], use centralized directories to index the information allowing them to support very complex queries. In these systems the search is centralized while the data exchange is distributed. Such systems are considered hybrid as they employ elements of both pure P2P systems and client/server systems. The key disadvantage of using centralized directories is that they limit the system's scalability. However, hybrid systems continue to be popular with recent systems such as KazaA [78] and Morpheus [79].

#### Napster

In Napster [80], a server node indexes files held by a set of users. Users search for files at a server, and when they find a file of interest they download it directly from the peer that stores the file. Users cannot search for files globally. They are restricted to searching on a single server that indexes only a fraction of the available files.

#### Morpheus

In Morpheus [79], nodes that have sufficient bandwidth and processing power are automatically elected as "SuperNodes" using proprietary algorithms and protocols called the FastTrack P2P Stack. "SuperNodes" index the content shared by local peers that connect to it, and proxy search requests on behalf of these peers. File downloads are peer-to-peer.

### 2.4.2 Unstructured Keyword Search Systems

Unstructured keyword search P2P systems, such as Gnutella [73], are ad-hoc and dynamic in structure and data location in these systems is random. The typical query mechanism consists of forwarding the query to as many peers as possible. Each node receiving a query processes it and returns results if any. A query may return only a subset (possibly empty) of available matches, and information present in the system may not be found. As a result, providing search guarantees requires that all peers in the system be queried, which is not feasible in most systems. Furthermore, the probability of finding a piece of data is directly proportional to its frequency. The mechanism used to route a query message in the network has a direct impact on the scalability of the system. The approach described above is called limited-scope flooding, and does not scale [73]. Other approaches such as *"Expanding Ring Search"*, *"Random Walks"* [38], and *"Routing Indices"* [15] improve search scalability. Furthermore, the number of results found can be improved using data replication. However, even with these optimizations, one cannot find all matches in a practical-size system. Unstructured keyword search systems are primarily used for community information sharing. Examples include Gnutella [73] and Freenet [12].

#### Gnutella and Optimizations

Gnutella uses a TTL[2]-based flooding search mechanism. This scheme introduces a large number of messages, especially if each node has a large number of neighbors. Furthermore, selecting the appropriate TTL so that requested data is found with minimum number of messages is not trivial. To address the TTL selection problem, successive floods with different TTL values are used. A node starts flooding with a small TTL, and if the search is not successful the TTL is increased and another flood is started. This process is called *Expanding Ring Search*. *Random Walks* [38] further improves the search efficiency. Instead of forwarding a query request to all the neighbors, a node forwards the query to $k$ randomly chosen neighbors in each step until the matching

[2]TTL (Time To Live), limits the living time of a message. Typically TTL is the number of hops a message can travel until it is discarded.

data is found. Another approach, *Probabilistic Search Protocol* [40], forwards the query to each neighbor with a certain probability $p$, called the *broadcast probability*.

The *Routing Indices* [15] approach allows nodes to forward the queries to neighbors that are more likely to have answers. If a node cannot answer a query it forwards the query to a subset of its neighbors based on its local *Routing Index*, rather than selecting neighbors at random. The *Routing Index* is a list of neighbors, ranked according to their "goodness" for the query. Multiple approaches for building *Routing Indices* have been proposed. As they are not relevant to this research they are not discussed here.

#### Freenet

Freenet [12] was designed to prevent the censorship of documents and to provide anonymity to users. Unlike Gnutella, which uses a breadth-first search (BFS) with depth TTL, Freenet uses a depth-first search (DFS) with a specified depth. Each node forwards the query to a single neighbor and waits for a response from the neighbor before forwarding the query to another neighbor (if the query was unsuccessful) or forwarding the results back to the query source (if the query was successful).

### 2.4.3 Structured Keyword Search Systems

Systems in this category build on top of a data look-up protocol. Their goal is to combine complex queries supported by unstructured and hybrid systems with the efficiency and guarantees of data lookup systems. Four systems in this class are summarized in this section: pSearch [65], the system proposed by Reynolds and Vahdat [49], Squid (the system presented in this thesis), and the system proposed by Andrzejak and Xu [1]. These systems were among the first to enrich lookup systems with search capabilities. More recently, a number of strategies and indexing schemes have been proposed, with a special interest in supporting range queries in P2P systems. A summary of these recent efforts is presented at the end of this section.

### pSearch

pSearch [65] is a P2P system that supports content and semantics based searches. It is built on top of CAN [48] and uses the Vector Space Model (VSM) [4] and Latent Semantic Indexing (LSI) [4] to index the documents. It uses LSI to store the documents in CAN using their semantic vectors as keys.

VSM represents documents and queries as vectors. Each component of the vector represents the importance of a word (term) in the document query. The weight of a component is usually computed using the TF $\times$ IDF (term frequency $\times$ inverse document frequency) scheme [4]. When a query is issued, the query vector is compared to all document vectors. The ones closest to the query vector are considered to be similar and are returned. One measure of similarity is the cosine between the vectors.

The VSM scheme computes the lexical similarity between the query and the documents. LSI is used to compute the semantic similarity. LSI uses singular value decomposition (SVD) to transform and truncate the matrix of *document vectors* computed from VSM to discover the semantic terms and documents. As a result a hi-dimensional document vector is transformed into a medium-dimensional *semantic vector* by projecting the former into a medium-dimensional semantic subspace. The similarities between the query and the documents are measured as in VSM, as the cosine of the angle between their vector representations. This scheme does not support queries containing partial keywords, wildcards, and ranges.

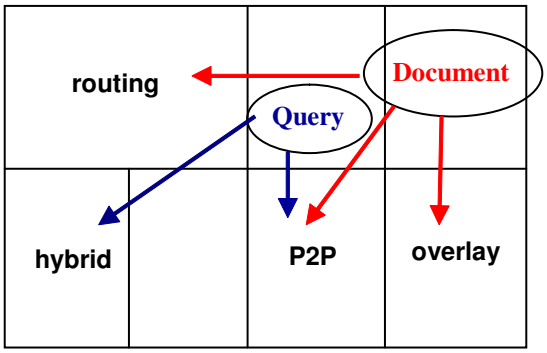



Figure 2.6: Storing a document described by the keywords "p2p", "overlay" and "routing" in the system, and querying the system for documents matching the keywords "p2p" and "hybrid".

Queries are resolved using the query semantic vectors as keys, by contacting the node that stores the key and flooding the query to nodes within a radius $r$ of the destination. Figure 2.6 summarizes the publish and query process.

### Reynolds and Vahdat [49]

Reynolds and Vahdat [49] propose an indexing scheme for structured P2P systems such as Chord [63] and Pastry [51]. They build an inverted index, which is distributed across the nodes using consistent hashing [32], and use Bloom filters to reduce bandwidth consumption during querying. An inverted index is a data structure that maps a keyword to the set of documents that contain the keyword. A Bloom filter [49] is a hash-based data structure that summarizes membership in a set. The membership test returns false positives with a tunable, predictable probability and never returns false negatives. The number of false positives falls exponentially as the size of the Bloom filter increases. In this system, data elements are described using a set of keywords. The data element, or a reference to it, is stored in the overlay at as many nodes as keywords. Each keyword is hashed to an identifier, which is used to locate the node where a reference to the data element will be stored. The local repository is an inverted index. A query consisting of multiple keywords is directed to multiple nodes. The results are merged at one of the nodes, and the final result is sent to the user. The system does not support partial keywords, wildcards, and range queries. Figure 2.7 illustrates how data is stored at nodes.

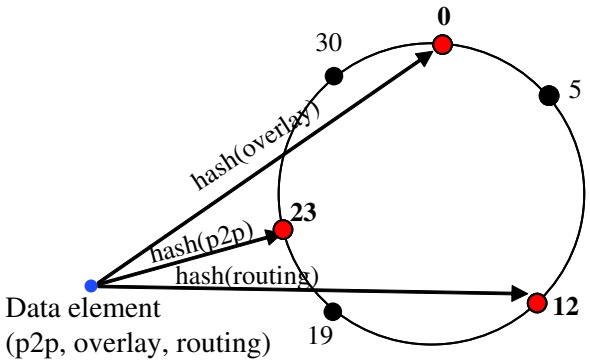



Figure 2.7: Storing the data element described by the keywords “p2p”, “overlay” and “routing” in the system.

### Squid

The approach presented in this thesis falls into the structured keyword search category, i.e., it provides the flexibility of keyword search systems while providing the guarantees and bounds of data lookup systems. Squid uses a structured overlay and an SFC-based indexing scheme. It guarantees that all existing data elements that match a query will be found with reasonable costs in terms of number of messages and number of nodes involved. It also supports searches with partial keywords, wildcards, and range queries.

### Andrzejak and Xu [1]

Andrzejak and Xu propose an information discovery system based on Hilbert SFC [1]. Unlike Squid, this system uses the inverse SFC mapping (I-SFC), from a one-dimensional space to a $d$-dimensional space, to map a resource to peers based on a single attribute (for example, memory). It uses CAN [48] as its overlay topology, and the range of possible values for the resource attribute (one-dimensional) is mapped onto CAN's $d$-dimensional Cartesian space. Figure 2.8 illustrates the publishing process. This system is designed for resource discovery in computational Grids, more specifically, to enhance resource discovery mechanisms with range queries. In contrast, Squid uses SFCs to encode the $d$-dimensional keyword space to a one-dimensional index space.

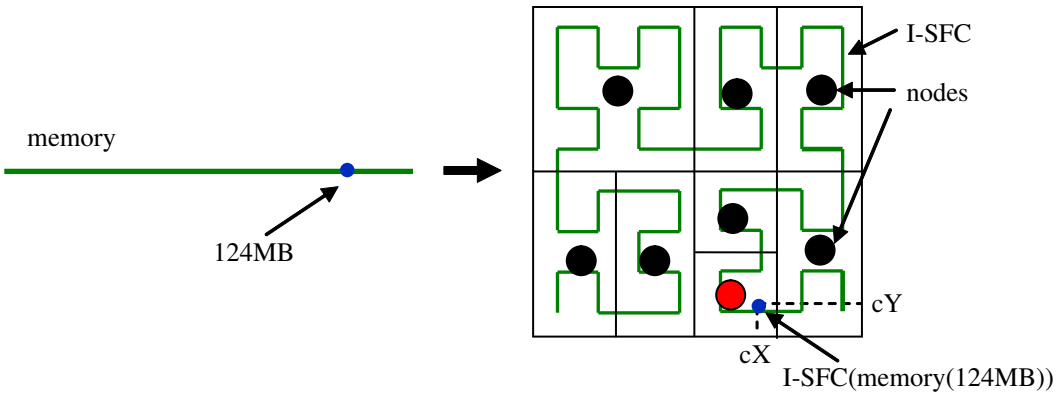



Figure 2.8: Storing the resource 124MB memory in CAN. Inverse SFC (I-SFC) is used to map the unidimensional resource into a 2-dimensional point, (cX, cY), which is stored at the corresponding node.

**Supporting Range Queries in P2P Systems**

Supporting range queries in P2P systems is an important and challenging problem and has been addressed extensively. A variety of systems have been proposed and evaluated recently. Some of these systems allow multi-attribute range queries, while others only support a single attribute.

To the best of our knowledge, Squid [56] is the first system to support multi-attribute range queries. Other systems that support multi-attribute range queries include: Mercury [5], PlaceLab [10] and Ganesan et. al. [25]. Systems supporting single attribute range queries include Andrzejak and Xu [1], P-Ring [14] and Skip Graphs [2].

The efficiency of these systems depends on the way the data is mapped onto the nodes. Maintaining data locality is very important, since it is desired that minimum number of nodes are contacted for each range query. It is known that hashing destroys data locality and can make range queries very expensive [60]. Several indexing schemes that preserve locality to different degrees have been proposed. The simplest approach is to use the exact values of the attributes. However, this approach limits the search to single attribute ranges [14], or imposes a more complex overlay network such as Mercury [5] where multiple simple overlays, one for each attribute, are mapped onto the same set of nodes. Other more complex indexing schemes, such as SFCs [10, 25] and kd-trees [25] have also been proposed.

Locality-preserving indexing schemes result in non-uniform data distribution at nodes. As a result, explicit load balancing mechanisms are employed, the join-leave type [5, 14] being the most popular one (e.g., a node has to move in a more crowded part of the overlay, leaving and re-joining the system). Place Lab [10] is an exception since it uses a two-layered indexing scheme - a locality preserving one on top of a consistent hashing mapping. As a result, a range query is resolved by performing a lookup for every value in the range, which makes the performance of the system suboptimal.

## 2.5 Summary

Table 2.2 summarizes the search characteristics of existing P2P systems, including our approach. To efficiently support complex queries, lexicographic locality must be preserved in the index space. Since DHTs and their hashing mechanisms do not preserve data locality, we use SFCs to build the index. Squid supports queries containing keywords, partial keywords, wildcards and ranges.

| P2P SYSTEM | | SEARCH CHARACTERISTICS | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Search Guarantees | Bounded Search Costs | Exact Search (Lookup) | Keyword Search | Partial Keyword Search | Wildcard Search | Range Query |
| Unstructured (Gnutella) | | No | No | Yes | Yes | Yes | Yes | N/A |
| Structured Data Lookup (Chord, CAN) | | Yes | Yes | Yes | No | No | No | No |
| Structured Keyword Search | Distributed Inverted Index (pSearch) | Yes | Yes | Yes | Yes | No | No | No |
| | Locality-preserving Index (Squid, Mercury, Place Lab) | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

Table 2.2: A comparison of P2P information discovery systems.

Squid presents unique characteristics that are not found in other structured keyword search systems. The systems based on a distributed inverted index do not support ranges, partial keywords and wildcards. Furthermore, to our knowledge, Squid is the first P2P system that supports multi-attribute range queries - existing solutions at that time supported only single-attribute queries. Squid was also the first P2P system to use SFCs to index the data. Andrzejak and Xu [1] used the inverse mapping (I-SFCs), supporting queries based on only one attribute. SFCs were used recently in several systems, including Place Lab [10] and Ganesan et. al. [25]. These systems use a similar approach to index data and distribute the index at nodes. However, the query engine is different. Place Lab performs a lookup for each value in the specified range, which makes its performance suboptimal. The approach used by Ganesan et. al. increases false-positives (e.g., non-relevant nodes may receive the query) in order to reduce the number of sub-queries.

# Chapter 3

# Squid - Architecture

The Squid architecture is similar to data-lookup systems presented in Chapter 2, and essentially implements an Internet-scale distributed hash table. The architecture consists of the following components: (1) a dimension-reducing, locality preserving indexing scheme, (2) a structured overlay network, (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 queries consisting of keywords, partial keywords, wildcards and ranges, using successive refinements and pruning. Figure 3.1 shows the overall architecture of Squid.

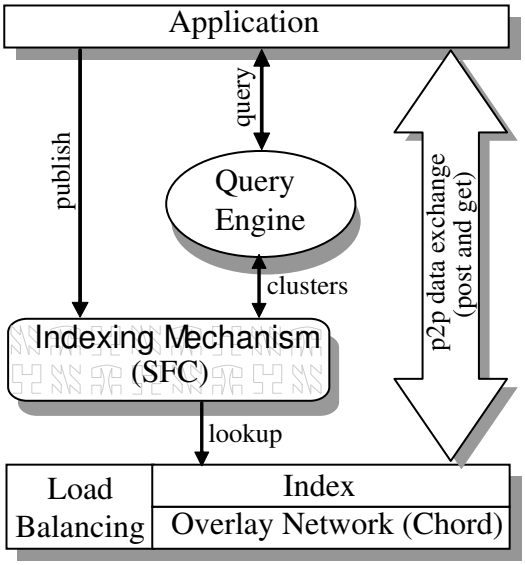



Figure 3.1: Squid Architecture - Overview

Squid provides the application with two primitives: *publish* and *query*. *publish* is used to store data elements (or references to them) within Squid, and *query* is used to query Squid for data elements. The data itself is directly transferred between nodes as in any P2P system. When publishing a data element, the lookup mechanism is used to obtain the address of the node where the data element is to be stored. When querying data elements, the query engine identifies the set of nodes where data elements

that match the query exist. The application is responsible for directly communicating with these nodes to retrieve the data elements. The rest of this chapter describes the components of Squid.

## 3.1 Constructing an Index Space: Locality Preserving Mapping

A key task of designing 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. The keywords may be common words in the case of P2P storage systems, or values of globally defined attributes of resources in the case of resource discovery in computational grids (examples in Section 3.3). Data elements can be searched using these keywords.

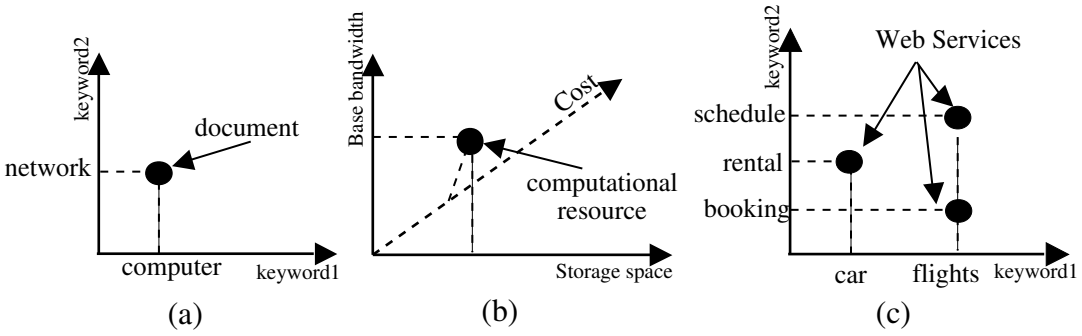



Figure 3.2: Keyword spaces: (a) a 2-dimensional keyword space for a file sharing system. The data element "document" is described by keywords *computer* and *network*; (b) a 3-dimensional keyword space for storing computational resources, using the attributes: storage space, base bandwidth and cost; (c) a 2-dimensional keyword space. There are three web services described by keywords: (flights, booking), (flights, schedule), (car, rental).

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 numerical values are used or 26 if the keywords are words in the English 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 data-elements. Examples of keyword spaces are shown in Figure 3.2.

To efficiently support range queries and queries using partial keywords and wild-cards, the index space should preserve locality and be recursive so that these queries can be optimized using successive refinement and pruning. In Squid, such an index space is constructed using Space Filling Curves.

### 3.1.1 Space-Filling Curves

SFCs were discovered by Peano [45] in 1890. Hilbert [28] later discovered a way to geometrically generate these curves in a recursive manner for a discrete space. SFC's have been used in various application domains including in databases to index multi-dimensional data [50], in image compression [35], for bandwidth reduction [6] and for dynamic partitioning in parallel systems [44].

A Space-Filling Curve (SFC) is a continuous mapping from a $d$-dimensional space to a 1-dimensional space, i.e., 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.

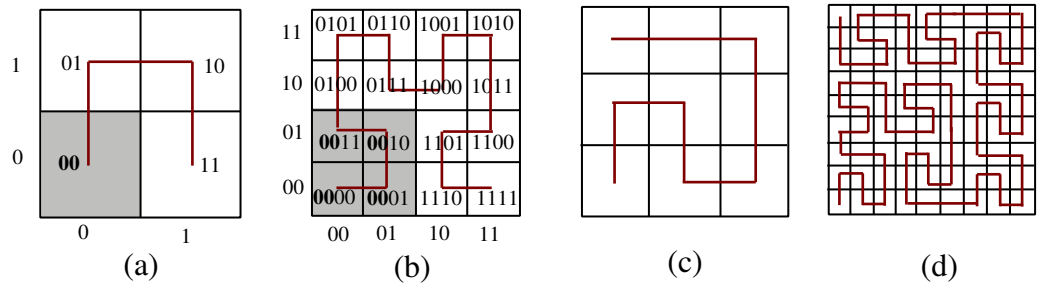



Figure 3.3: Space-filling curve approximations for $d = 2$: (a) $n = 2$, $1^{st}$ order approximation; (b) $n$ =2, $2^{nd}$ order approximation; (c) $n = 3$, $1^{st}$ order approximation; (d) $n = 3$, $2^{nd}$ order approximation.

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 Figures 3.3 (a) and (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 3.3 (b) and (d) show the $2^{nd}$ order approximation for the curves in Figures 3.3 (a) and (c) respectively. Note that the curve in each sub-cube is identical in structure to the original first approximation curve, possibly with a different orientation. This property of the space-filling curve is called *self-similarity*.

An important property of SFCs is *digital causality*, which comes directly from its self-similar nature. A unit length curve constructed at the $k^{th}$ approximation has an equal part of its total length contained in each sub-hypercube, i.e., 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 sub-cube and belong to a line segment are identical in their first *(k-1)d* digits. This property is illustrated in Figures 3.3 (a) and (b). The the subcube (0, 0) in (a), with SFC index 00 is refined resulting in four subcubes in (b), each with the same first two digits as the original subcube.

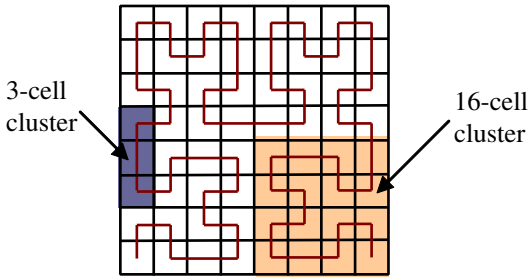



Figure 3.4: Clusters on a $3^{rd}$ order space-filling curve ($d = 2$, $n = 2$). The colored regions represent clusters: 3-cell cluster and 16-cell cluster.

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 $k \geq 1$ and $d \geq 2$, the $k^{th}$ order approximation of a $d$-dimensional Hilbert space filling curve maps the sub-cube $[0, 2^k - 1]^d$ to the segment

[0, $2^{kd}$ - 1] of the curve. The reverse property is not true, 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 are called clusters, and are shown in Figure 3.4. In this figure the colored regions represent 3-cell and 16-cell clusters.

There are different types of SFCs and each of them exhibit different locality preserving properties (also called clustering properties). A number of different SFCs have been proposed in [53]. A selection of these, the Morton curve (z-curve), Gray code, and Hilbert curve, are shown in Figure 3.5. The degree to which locality is preserved by a particular SFC is defined by the number of clusters that an arbitrary region in the $d$-dimensional space is mapped to.

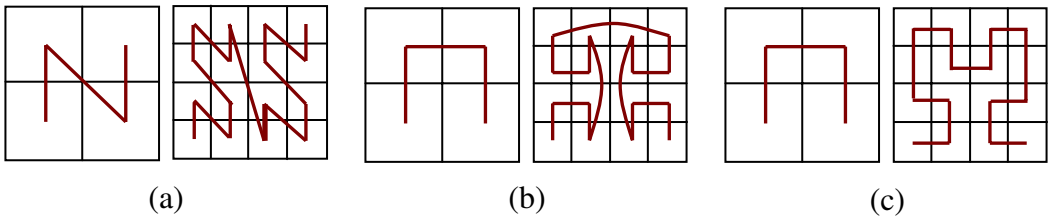



Figure 3.5: Examples of space-filling curves (first and second order): (a) Morton curve, (b) Gray code, (c) Hilbert curve.

In this research we use the Hilbert SFC to map data elements to the index space as it is widely believed that this curve achieves the best clustering [41]. In our system, SFCs are used to generate the one-dimensional index space from the multi-dimensional keyword space. Applying the Hilbert mapping to this multi-dimensional space, each data element is mapped to a point on the SFC. Any range query or query composed of keywords, partial keywords, or wildcards, are mapped to regions in the keyword space and corresponding clusters in the SFC.

Note that, if the type of queries (e.g., shape) is known a-priori, the curve can be chosen such that the number of clusters is minimized for those specific types of queries. Squid is a generic information discovery system and does not make any assumptions about the queries. However, any recursive SFC can be used in Squid, by only changing the indexing algorithm.

### 3.1.2 Mapping the Keyword Space to the Index Space

This section describes the process of mapping the keyword space to the index space using the Hilbert SFC. The example chosen to illustrate the process uses English words as keywords, which are treated as base-26 numbers.

The Hilbert SFC is base-2, i.e., each level of approximation is obtained from the previous one by refining each cell into $2^d$ new cells. At level $k$ of approximation, each axis has $2^k$ possible coordinates, a coordinate being expressed as a $k$-bit binary number. As shown in Figures 3.3 (a) and (b), the first level of approximation has 1-bit coordinates, and the second level of approximation has 2-bit coordinates.

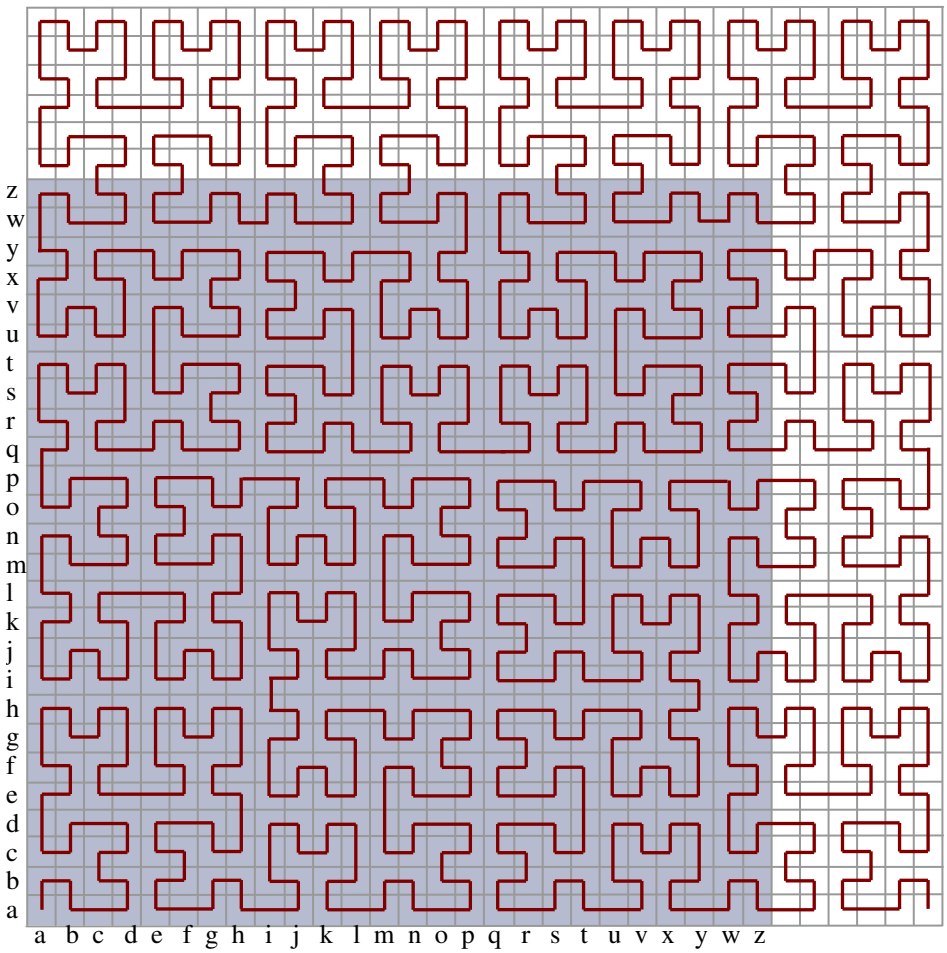



Figure 3.6: Hilbert SFC, the $5^{th}$ approximation.

English words can be viewed in the context of a base-26 coordinate space. Each character, in binary format, is 5-bit long. Since we need 5 bits to represent each

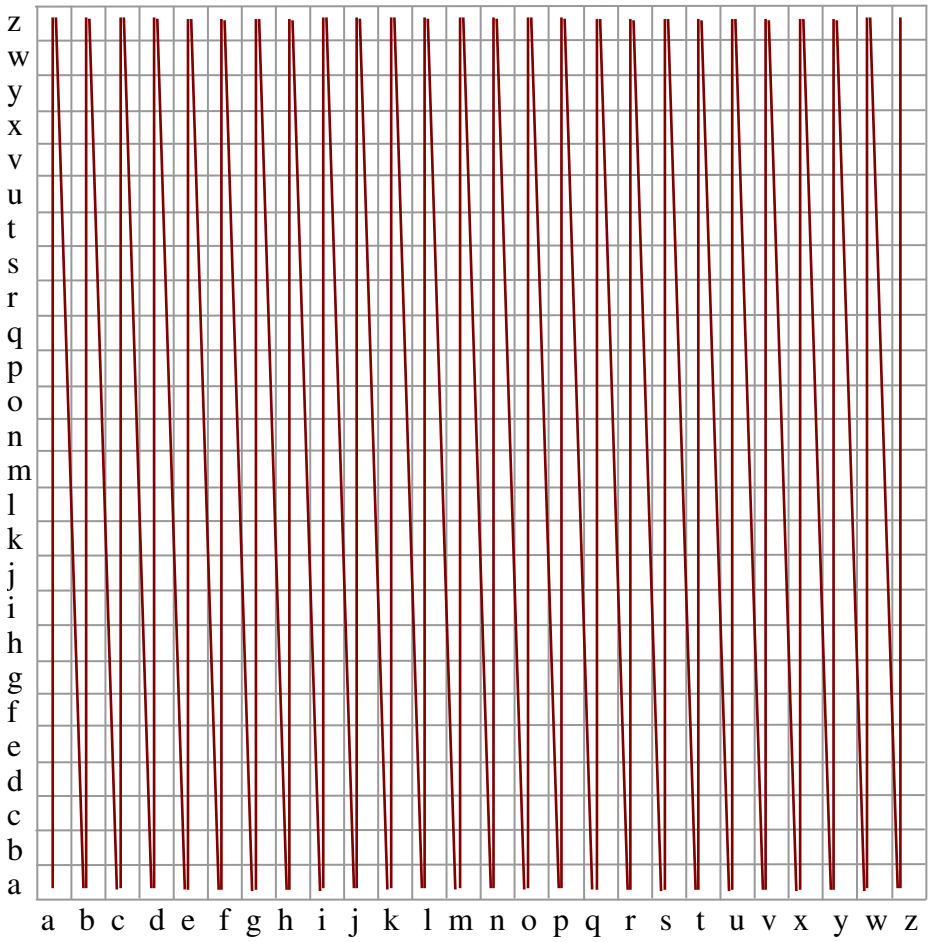



Figure 3.7: Morton SFC (z-curve), base 26, the first approximation.

character, the Hilbert SFC has to be refined 5 times. This is illustrated in Figure 3.6, which shows the $5^{th}$ approximation of Hilbert SFC and the mapping of the set of alphabets such that 00000 corresponds to 'a', 00001 to 'b', 00010 to 'c', etc. The colored region in Figure 3.6 shows the portion of the space actually used, with alphabetic keywords only. As the figure shows, there are 6 additional positions on each axis. They can be used for other characters (e.g., numeric, punctuation, etc.), or left unused. If they are not used, portions of the SFC will never be populated with data.

In general, if to represent the characters of the language used by the keywords requires $b$-bits, the base recursion step will be the $b^{th}$ approximation of the Hilbert SFC. For English alphabets, the base recursion step is the $5^{th}$ approximation of the curve, as shown in Figure 3.6. If the keyword is $c$ characters long, the curve will be

refined *bc* times.

As Figure 3.6 shows, when using base-26 numbers as coordinates, a portion of the space is never used. The ideal solution would be to use a 26-base SFC. Figure 3.7 shows an example of a modified Morton curve (z-curve), in base 26. The problem with this curve is that it does a poor job preserving locality, for example, for horizontal queries.

## 3.2 Overlay Network

Since Squid uses a one-dimensional index space, a one-dimensional overlay is the overlay of choice. The current implementation of Squid uses the Chord [63] overlay network topology. However, any one-dimensional overlay network can be used within Squid (e.g., Pastry). We choose Chord because of its simplicity of design, resilience and performance. The Chord overlay network topology, routing algorithms and management mechanisms are described below.

### 3.2.1 Chord Overview

In the Chord overlay network, each node has a $m$-bit 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 $i^{th}$ 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.

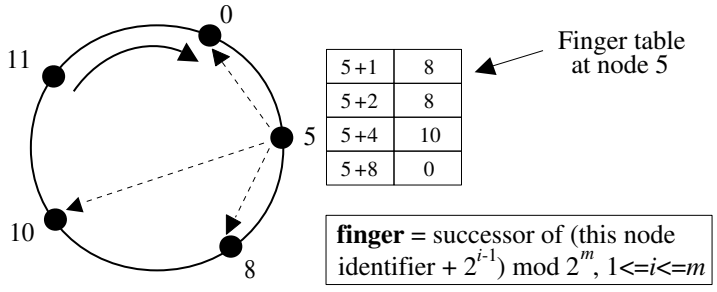



Figure 3.8: Example of the overlay network. Each node stores the keys that map to the segment of the curve between itself and the predecessor node.

In our implementation, node identifiers are generated randomly. Each data element

is mapped, based on its SFC-based index, called its key, to the first node whose identifier is equal to or follows the key in the identifier space. This node is called the *successor* of the key. For example, consider the sample overlay network with 5 nodes and an identifier space from 0 to $2^4$-1, shown in Figure 3.8. 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:** The joining node has to know about at least one node that is already in the system. The joining node randomly chooses an identifier from the identifier space [0, $2^m$-1) and sends a join message with this identifier to the node that it knows of. This message is routed across the overlay network until it reaches the successor of the identifier. The joining node is inserted into the overlay network before the successor node, and takes a part of the successor node's load. If a collision occurs, i.e., the new node's chosen identifier is already used by a node already in the system, the joining process will identify this node as the successor of the new identifier. The new node will recognize that it has chosen the same identifier as the successor and will select a new identifier that is in the interval between the successor's predecessor and the successor identifiers, and will complete the join. The predecessor will help the new node constructs its finger table and update its successor and predecessor entries. The new node will also update the finger tables at nodes that should point to it. 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 where 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 invalid, resulting in failed queries. To repair the finger tables each node periodically runs a stabilization algorithm where it chooses a random entry in its finger table and updates it by finding the correct finger node.

**Data Lookup:** The Chord data lookup protocol efficiently locates nodes based on their identifiers. Data lookup takes O(log N) hops, where N is the total number of the nodes in the system. Note that in our system, a complex query will typically require interrogating more than one node as the desired information may be stored at multiple nodes in the system. This requires additional optimizations which will be presented below.

### 3.2.2 Mapping Indices to Peers

Publishing data elements in Squid requires three steps: (1) describe the data elements using keywords, (2) construct the index using the Hilbert SFC, and (3) store the data (or a reference to the data) in the system using this index. Figure 3.9 illustrates the publishing process.

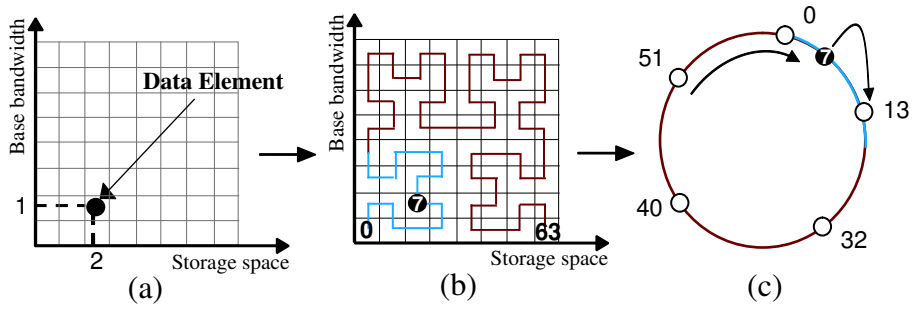



Figure 3.9: The process of publishing the data element which represents a computational resource (a machine with 2MB memory and 1Mbps bandwidth) into the system: (a) the data element (2, 1) is viewed as a point in a multidimensional space; (b) the data element is mapped to the index 7, using Hilbert SFC; (c) the data element is stored in the overlay (an overlay with 5 nodes and an identifier space from 0 to $2^6$-1) at node 13, the successor of index 7.

As Figures 3.9 (b) and (c) show, the curve is "flattened" and "wrapped" around the circle. Each node is responsible for storing data elements associated with the segment between its predecessor and itself.

## 3.3 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 up to

$d$ keywords, where $d$ is the dimensionality of the keyword space. 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 query. For example, (computer, network), (computer, net*) and (comp*, *) are all valid queries. Another type of query is a range query where at least one dimension specifies a range. For example if the index encodes memory, CPU frequency and base bandwidth resources, the following query (256 - 512MB, *, 10Mbps - *) specifies a machine with memory between 256 and 512 MB, any CPU frequency and at least 10Mbps base bandwidth.

### 3.3.1 Query Engine Design 1: Straightforward Approach

Processing a query consists of two steps: (1) translating the keyword query to relevant clusters of the SFC-based index space, and (2) querying the appropriate nodes in the overlay network.

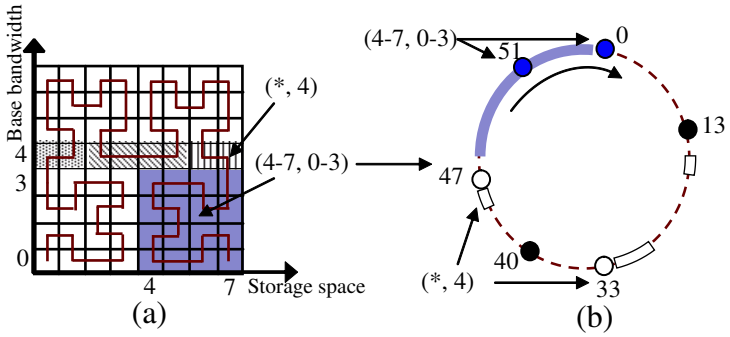



Figure 3.10: Searching the system: (a) regions in a 2-dimensional space defined by the queries (*, 4) and (4-7, 0-3); (b) the clusters defined by query (*, 4) are stored at nodes 33 and 47, and the cluster defined by the range query (4-7, 0-3) is stored at nodes 51 and 0.

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 overlay network's lookup protocol. If the query contains partial keywords and/or wildcards or is a range query, 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 3.10 (a), the query (*, 4) identifies 8 data elements (the horizontal region). The index (curve) enters and exits the region three times, defining three segments of the curve or

clusters, which are marked by different patterns in the figure. Similarly, the query (4-7, 0-3) identifies 16 data elements, defining the square region in Figure 3.10 (a). 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, a cluster may be mapped to one or more adjacent nodes in the overlay network. A node may also store more than one cluster. For example, in Figure 3.10 (b), node 33 stores 2 clusters, and the 16-cell cluster is stored at nodes 51 and 0. Once the clusters associated with a query are identified, straightforward query processing would consist 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 a data lookup protocol so that, given an identifier for a data element, the node responsible for storing it can be 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 the SFC 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 prefix* and form the first $i$ digits of the required $m$ digit identifier. The rest of the identifier is padded with zeros.

Note that the node that initiates a query can not know if multiple clusters are stored at the same node and make optimizations. The number of clusters can be very large, and sending a message for each cluster is not a scalable solution. For example, consider the query (000, *) in Figure 3.10, 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.

### 3.3.2 Query Engine Design 2: Optimized Query Engine

The scalability of query processing can be optimized using the observation that a large number of clusters will typically be stored at the same node. The goal is to identify the set of clusters so that each node containing matching data receives one message (or very few messages) per query. However, the exact number of clusters per node cannot be known at the node where the query is initiated. The solution is to use

the recursive nature of the SFC and its digital causality property to decentralize and distribute the process of cluster generation across multiple nodes in the system, which might be responsible for storing the clusters.

Since SFC generation is recursive and clusters are segments on the curve, these clusters can also be generated recursively. The digital causality property of the curve allows us to generate the cluster identifier hierarchically, where at each level of refinement a longer cluster prefix is created. Figure 3.11 illustrates the recursive cluster generation process.

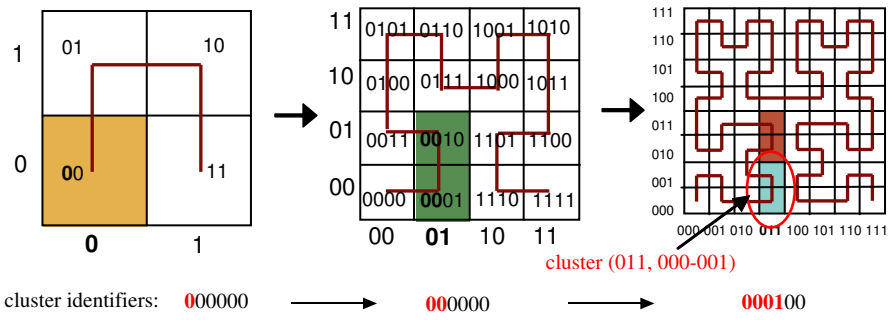
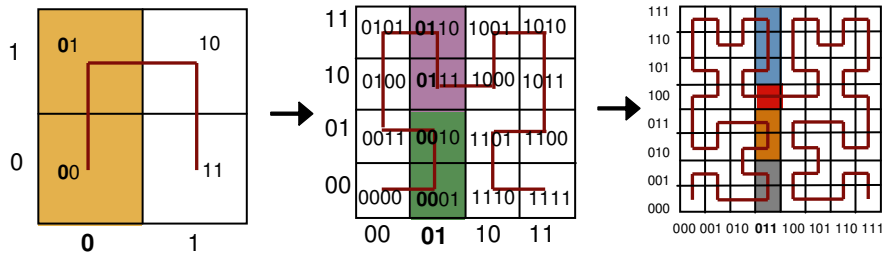



Figure 3.11: Recursive generation of cluster (000, 000-001). Each refinement results in a bigger cluster prefix (red digits). The cluster identifier is obtained by padding with zeros the cluster prefix, until the specified length is reached (in this case 6).

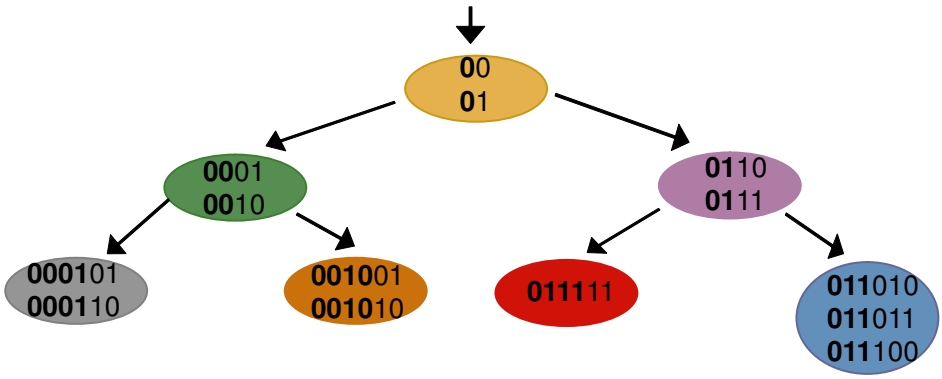
Recursive query processing and cluster generation can be viewed as constructing a tree. At each level of the tree the query defines a number of clusters, which are refined to generate more clusters at the next level. The tree can now be embedded onto the overlay network: the root of the tree performs the first refinement of the query, and each subsequent node further refines the query and forwards the resulting sub-queries down the tree to 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 3.12 (a) shows the successive refinement for the query and Figure 3.12 (b) shows the corresponding tree. The leaves of the tree contain all possible matches for the query.

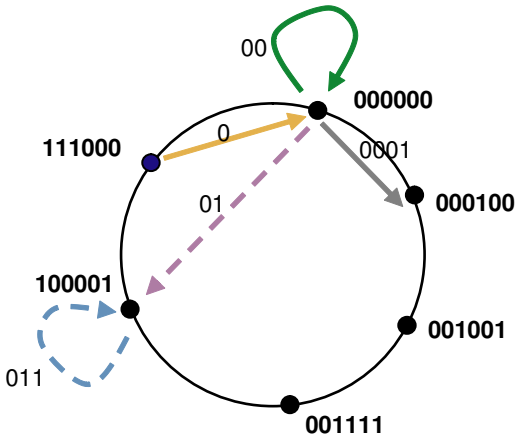
Query optimization consists of pruning nodes from the tree during the construction phase. As a result of the load-balancing steps described in Section 3.5, the nodes tend to follow the distribution of the data in the index space, i.e., a larger number of nodes are




(a)




(b)




(c)

Figure 3.12: Query optimization: (a) recursive refinement of the query (011, *): one cluster on the first order Hilbert curve, two clusters on the second order Hilbert curve, 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 cluster's prefixes; (c) embedding the leftmost tree path (solid arrows) and the rightmost path (dashed arrows) onto the overlay network topology.

assigned to denser portions of the index space, and no nodes are assigned to the empty portions. If we embed the query tree onto the ring topology of the overlay network, we can prune away many of the sub-trees that are mapped to the same overlay node, using the prefix of the root of these sub-trees. Figure 3.12 (c) illustrates the process, using the query in Figure 3.12 (a) as an example. The leftmost path (solid arrows) and the rightmost path (dashed arrows) of the tree shown in Figure 3.12 (b) are embedded onto the ring overlay. The overlay network uses 6 digits for node identifiers. The arrows are labeled with the prefix of the cluster being queried. The query is initiated at node 111000. The first cluster has prefix 0, so the cluster 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 100001 (dashed line). This cluster will not be refined because the prefix of the node identifier is greater than the prefix of the cluster identifier, and all matching data elements are stored at this node.

```
solveQueryOptimized(query, cluster)
  if (cluster = null)  //at the query originator
     clustersList = firstRefinement(query)
     sendClusters(clusterList)
  else if (cluster.identifier <= node_identifier)
     //the cluster is at this node
     searchLocalData(query)  //stop at this node
  else   //the cluster is refined and sent to other nodes
     clustersList = nextRefinement(query, cluster)
     sendClusters(clusterList)

//sends the clusters to the appropriate nodes in the overlay
sendClusters(clusterList)
  sortedClusterList = sortInAscendingOrder(clusterList)
  while (sortedClusterList.hasMoreElements())
     firstCluster = sortedClusterList.nextElement()
     nodeID = lookup(firstCluster)
     clustersToSend.add(firstCluster)
     for (each cluster in sortedClusterList with cluster.id <= nodeID)
        clustersToSend.add(cluster)
     send(clustersToSend, nodeID)
     clusterToSend.removeAll()
```

Figure 3.13: Pseudo-code of query processing at a node.

Note that the example presented above is a simplified one, chosen for the purpose of illustration. In practice, each query refinement corresponds to more than one SFC approximation, resulting in clusters with much longer prefixes.

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 may be mapped to the same node. To reduce the number of messages, each node sorts the sub-clusters in increasing order and performs a lookup for the subcluster with the lowest identifier. Once the identifier of the node storing the cluster is known, it can be deduced that all the subclusters with identifiers less than or equal with the node's identifier will also be stored at the node. A lookup for each sub-cluster is not required. All the subclusters can be aggregated and sent as a single message. Figure 3.13 presents the pseudo-code for processing a query at a node.

## 3.4 Application-Specific Issues

### 3.4.1 Interchangeable Axes

Some applications may have a keyword space where the axes are interchangeable. For example, this would be the case for a collection of scientific articles where the keywords describe the content of the articles, rather than representing specific attributes, such as author, title, publication year, etc. In this case any keyword can belong to any axis, an article described by the keywords "computer" and "network" and stored in a 2-dimensional keyword space, can be stored as (computer, network) or (network, computer). In this example, when searching for (computer, *), we would have to search for both (computer, *) and (*, computer), to get all the matches in the system. This is illustrated in Figure 3.14. In this figure, a data element described by keywords "computer" and "network" is stored only as (network, computer). As a result the query (computer, *) which should find this data element, does not.

There are two ways to address this issue. The first one is to store each data element multiple times, once for each permutation of the keywords, i.e., the data element in Figure 3.14 should be stored as (computer, network) and (network, computer). If the

keyword space has $d$ axis, each data element has to be stored $d!$ times. Figure 3.15 illustrates this process for the data element (0, 1, 3), indexing it 3! times. In reality however, we believe that a data element will be stored fewer than $d!$ times, since more than one identifier will typically be mapped to the same node, as demonstrated by the experiment below.

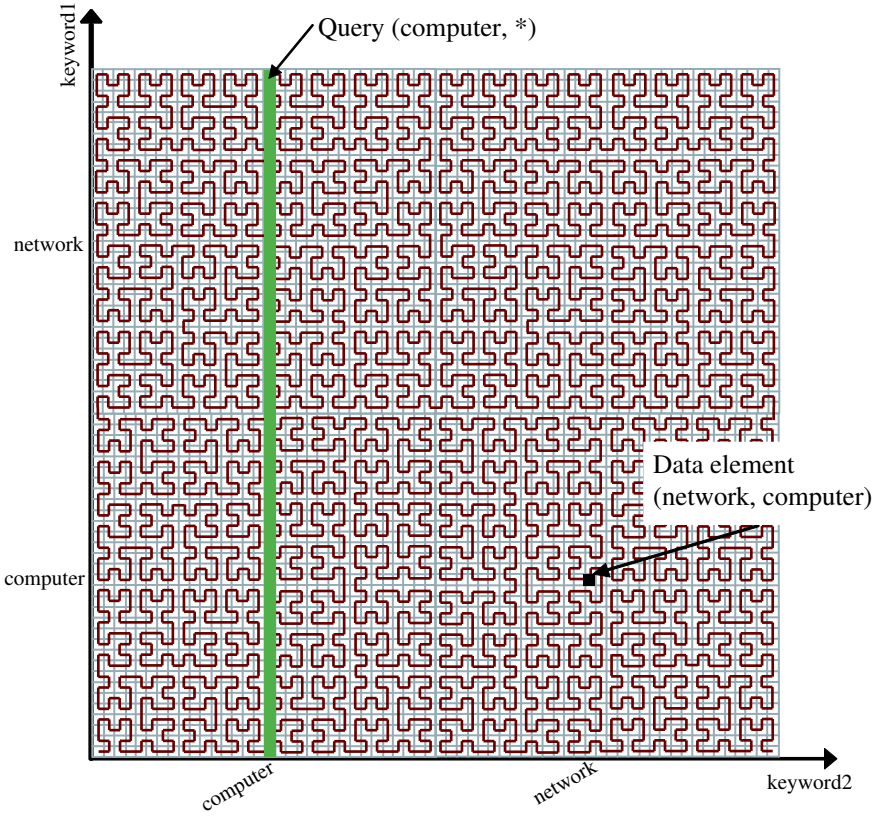



Figure 3.14: Storing a data element described by the keywords "computer" and "network" using the keyword ordering (network, computer). The query (computer, *) defines a region that does not include the data element.

The number of nodes that store a data element indexed $d!$ times was experimentally measured as follows. We used $10^4$ data elements, which were CiteSeer [72] HTML files describing scientific articles. The data elements were described using 8-character long keywords. 3-dimensional, 5-dimensional and 7-dimensional keyword spaces were used, and the system size was varied from $10^3$ to $10^5$ nodes. Each data element was indexed $d!$ times, and a lookup was performed for each resulting index. We counted the number of distinct nodes identified by the lookup mechanism for each data element. The results

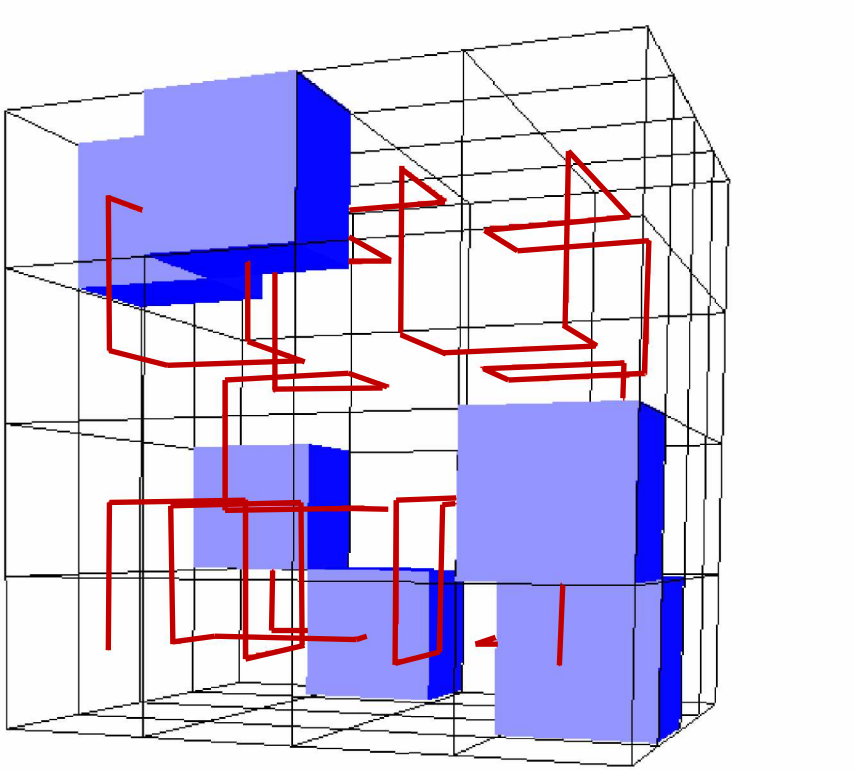


Figure 3.15: Mapping the data element (0, 1, 3), using a 3D space indexed by the Hilbert SFC. The data element is indexed 6 times, corresponding to all permutations. Each permutation is represented as a cube.

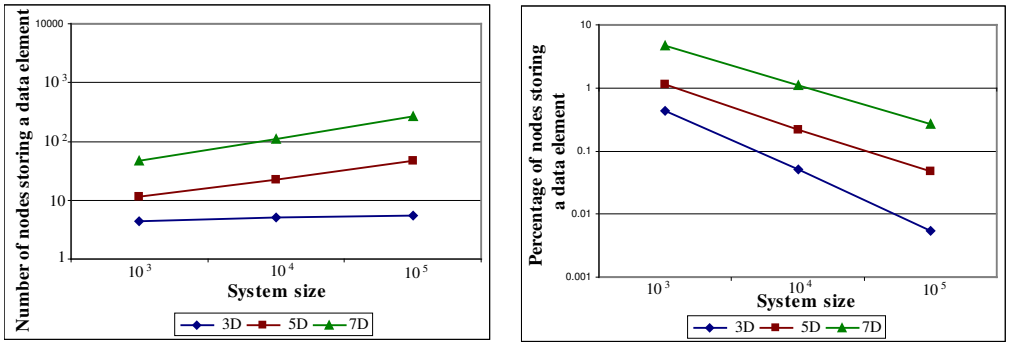
were averaged, and are presented in Table 3.1 and Figure 3.16.

Table 3.1 and Figure 3.16 (b) show that the percentage of nodes storing a data element is small, and it decreases as the size of the system increases. Further, the factorials seem to disappear, especially for high dimensional keyword spaces. For example, when using 7 keywords, a data element should be indexed 7! = 5040 times. However, as seen in Table 3.1, if the system has $10^3$ nodes, only 47 (4.7%) nodes on average store the data element. For systems with $10^4$ and $10^5$ nodes, 111 (1.11%) nodes respectively 270 (0.27%) nodes store the data element, on average. The number of nodes storing a data element indexed $d!$ times is plotted in Figure 3.16.

The second approach is to store the data element only once and to consider the $d!$ permutations while querying. Figure 3.17 (a) shows the region that has to be searched for the query (1, *). Both (1, *) and (*, 1) queries need to be resolved.

| Dimensions | Nodes | | |
|---|---|---|---|
| | $10^3$ | $10^4$ | $10^5$ |
| 3D | 0.44% | 0.051% | 0.0054% |
| 5D | 1.13% | 0.22% | 0.047% |
| 7D | 4.7% | 1.11% | 0.27% |

Table 3.1: Percentage of nodes storing data elements, each data element being indexed $d!$ times.




(a) (b)

Figure 3.16: (a) Number of nodes storing a data element indexed $d!$ times, as function of keyword space dimensionality, and system size. (b) Percentage of nodes storing a data element indexed $d!$ times.

In this case the region to be searched can be reduced if the keywords are sorted in lexicographical order. Sorting the keywords in lexicographical order and assigning them to axis in that order leads to a keyword space almost half empty. The data elements will be stored on the diagonal, and above it. Figure 3.17 (b) illustrates this situation. The queries are modified accordingly in this case, since they need to cover a smaller region. For example, as Figure 3.17 (c) shows, the query (1, *) will be solved using two range queries: (0-1, 1) and (1, 1-3).

The first solution is preferred, since publishing is less expensive than querying. Note that this is not an issue in applications where each axis has a distinct meaning.

### 3.4.2 Keyword Space Dimensionality

The dimensionality of the keyword space dictates the maximum number of keywords allowed. Analysis has shown that the locality-preserving property of SFC begins to

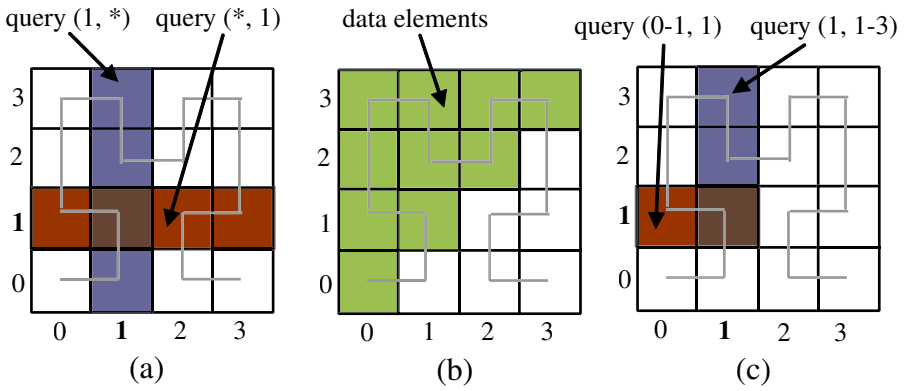



Figure 3.17: (a) Regions in a 2-dimensional keyword space defined by queries (1, *) and (*, 1). (b) The region in a 2-dimensional keyword space used by data elements when their keywords are ordered lexigrophically. (c) Regions in a 2-dimensional keyword space defined by queries (0-1, 1) and (1, 1-3).

deteriorate for information spaces of dimensionality greater than five [41]. As the keyword space dimensionality grows, the queries define more clusters, which are scattered further and further apart on the curve, requiring more nodes to be queried.

There are several ways to address this problem. A simple solution is to use the most relevant subset of keywords to index the data element using SFC, and the rest of the keywords to create a local index at the node storing the data elements. When querying the system, the query will be resolved at two levels: keywords for searching the SFC index, and keywords for searching the local index. For example, if the data elements are text files, and the keywords are assigned relevancies (e.g., frequency, weight), the keywords used by Squid will be the most relevant ones. If the data elements are scientific data for example, and the keywords are values of globally known attributes, the most relevant attributes will be used in Squid (the ones that will narrow the search area).

The second approach is application-specific, and can be applied to keyword spaces where the keywords are values of globally-known attributes and each axis of the keyword space corresponds to an attribute. In this case, the keywords can be grouped into possibly overlapping sets such that each set can be queried independent of others. An index space is constructed for each set and all indices are mapped onto the same overlay of peer nodes. Local indices can also be created for additional (less significant) attributes.

The third approach is to deterministically reduce the dimensionality of the space. For example, if the keywords are values of globally known attributes, and each axis of the keyword space corresponds to an attribute, the keyword space dimensionality can be reduced by combining axis as follows. Two or more attributes are mapped to an axis and the values of these attributes are interlaced to create a single value. For example, if the attributes *memory* and *bandwidth* use the same axis, and their values are 148 and 256, the value used as coordinate on that axis will be 124586. The queries have to be modified accordingly. For example, if the query specifies only the memory, and the bandwidth is substituted by a wildcard, the query component on that axis will be: 1?4?8?, which is easy to resolve using the optimized query engine. Other ways to reduce the dimensionality of the keyword space will be investigated, as part of the future work. This work uses the first two approaches.

## 3.5 Balancing Load

As mentioned earlier, the original $d$-dimensional keyword space is sparse, and data elements typically form clusters in this space rather than being uniformly distributed in the space. As the Hilbert SFC-based mapping preserves keyword locality, the index space will also preserve this locality. Since the nodes are uniformly distributed in the node identifier space, when the data elements are mapped to the nodes, load will not be balanced, i.e., some nodes will get more data elements than others. Additional load balancing is required. Note that even in existing P2P data lookup systems, where a uniform hash function is used for mapping, load balancing may be required as the range of possible values for node identifiers is very large and the uniform distribution is achieved only if the number of nodes in the system is equally large. Two load-balancing strategies are described below.

### 3.5.1 Load Balancing at Node Join

When a node joins the system, 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. In 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 of updating the tables remains O($\log^2$N). However, this step is not sufficient by itself. The runtime load-balancing algorithms presented below further improve load distribution.

### 3.5.2 Load Balancing at Runtime

The runtime load-balancing step consists of periodically running a local load-balancing algorithm between a few neighboring nodes. Two load-balancing algorithms are proposed. 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 used 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). The cost of this load balancing algorithm is also O($\log^2$N). A discussion of load balancing techniques for DHT systems can be found in [9]. An evaluation of the load balancing algorithms described here is presented in Section 3.6.1.

## 3.6 Experimental Evaluation

The performance of Squid is evaluated first using a simulator and then using a prototype implementation. The system implementation uses JXTA [77], a general-purpose P2P framework. These evaluations are presented in the following sections.

### 3.6.1 Evaluation Using the Squid Simulator

The Squid 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 [63], its maintenance costs are of the same order as in Chord. In a system with N nodes, the cost of a node join or departure is $O(\log^2 N)$, and data lookup requires O(log N) messages. An evaluation of the query engine and the load-balancing algorithms is presented below.

#### Evaluating the Query Engine

The query engine is evaluated using three sets of experiments. In each case the performance of the query engine is measured in terms of the number of nodes that participate in a query, the number of messages required to process a query, and the number of nodes where matches are found. The first set of experiments assumes that the number of data elements stored in the P2P system grows with the size of the system. This is typical of P2P sharing systems. The results show that the system scales well under these conditions. The second set of experiments evaluates the case where 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 larger number of nodes. The results show that the system continues to perform well. The overlay network used in the experiments consisted of between 1000 and 5400 nodes. Data elements were associated with up to two keys for a 2-dimensional keyword space (2D), and with up to three keys for a 3-dimensional keyword space (3D). Finally, up to $10^6$ keys (unique keyword combinations) were used, each of which could be associated with one or more data elements. In each experiment, we measured the following:

- **Number of processing nodes:** Processing nodes are the nodes that actually process the query, refine it, and search for matches. Ideally, these nodes should

be exactly the nodes that store matching data elements.

- **Number of data nodes:** Data nodes are nodes that have data elements that match the query. Data nodes are a subset of the processing nodes. The goal is to maintain the number of processing nodes as close as possible to the number of data nodes.

- **Number of messages:** This is the number of messages required to resolve a query. When using the query optimization, each message is a sub-query that searches for a subset of the clusters associated with the original query.

Three types of queries were used in the experiments:

- **Q1:** Queries with one keyword or partial keyword, e.g., (computer, *) for 2D, (comp*, *, *) for 3D.

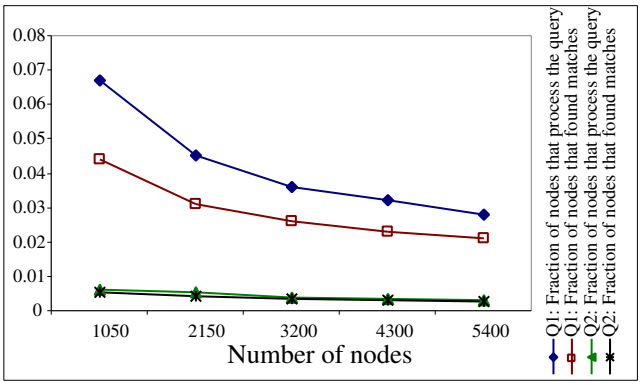
- **Q2:** Queries with two to three keywords or partial keywords (at least one partial keyword), e.g., (comp*, net*) for 2D, (computer, network, *) for 3D.

- **Q3:** Range queries of type (keyword, range, *) and (range, range, range). For range queries we performed experiments using a 3-dimensional space only.

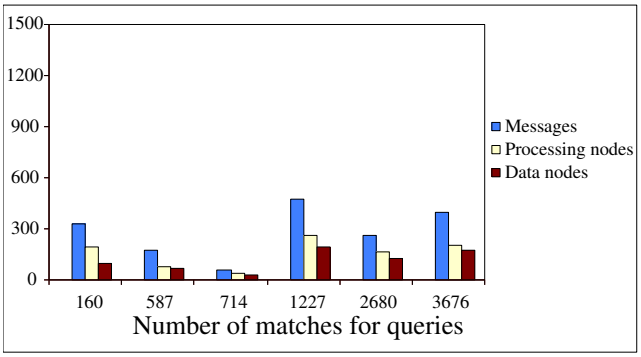
Note that queries of the form (keyword, keyword, keyword) are similar to the queries supported by data lookup systems such as Chord and are consistently resolved using O(log N) messages. Also, note that keys may have a different number of data elements associated with them depending on their popularity.

**Experiment 1 - Increasing System Size and Increasing Number of Data Elements**

This experiment represents a typical P2P sharing system where the number of keys and data elements in the system increases as the number of nodes increases. The system size in the experiment increased from 1000 nodes to 5400 nodes, and the number of stored keys increased from $2\times10^5$ to $10^6$.




(a)

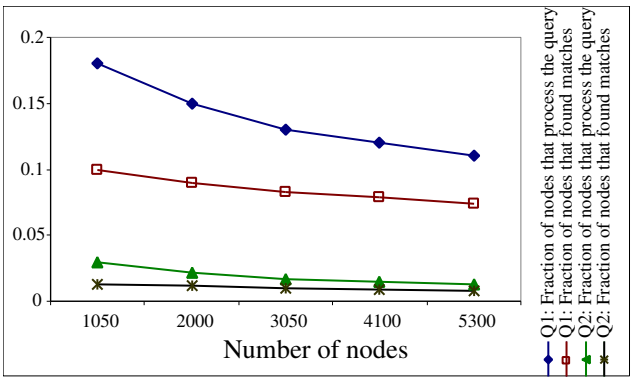



(b)

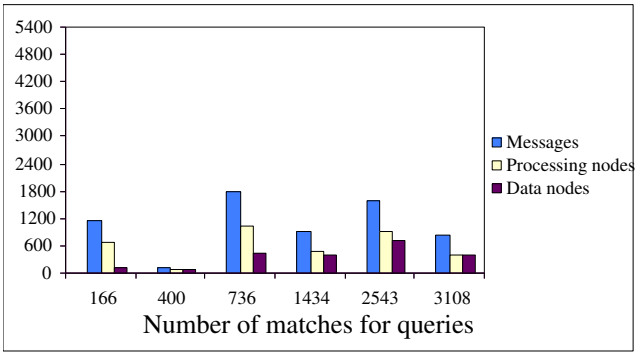
Figure 3.18: Experiment 1, 2D: (a) results for query types Q1 and Q2; (b) results for query type Q1, in a system of 5400 nodes and $10^6$ keys.

Sets of queries of each type were tested. The queries were chosen such that the number of matches represented the same fraction of the total data regardless of the size of the system (number of nodes) and the number of data elements. 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). The results were averaged and normalized.

As seen in Figures 3.18 (a), 3.19 (a), 3.20 (a), the number of processing and data nodes is a small fraction of the total nodes and increases at a slower rate than the system size. For a 2D keyword space, the average number of processing nodes is below 8% and the number of data nodes is below 5%, and this percentage decreases as the system size and number of data elements increases, which demonstrates the scalability of the system. The number of data nodes is close to the number of processing nodes, indicating that the query optimization sends clusters to the nodes that have matching




(a)




(b)

Figure 3.19: 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.

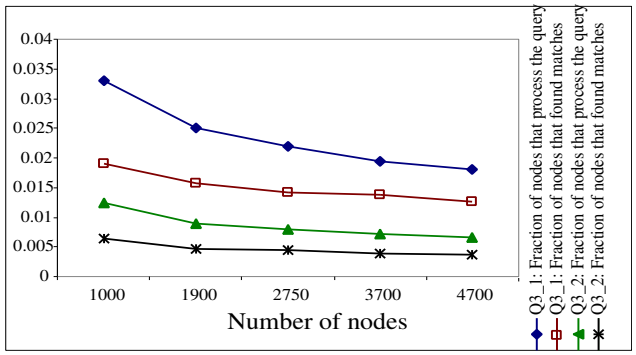
data most of the time. Also, Q2 queries are more efficient than Q1 queries, which is expected. This is because query optimization and pruning are more effective when both keywords are at least partially known.

The 2D and 3D results have a similar pattern, the only difference is in the magnitude of the results. As described in Section 1, documents that share a specific keyword will typically be mapped to disjoint segments of the curve (clusters). In the 3D case the number of such segments is larger than in the 2D case as 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.

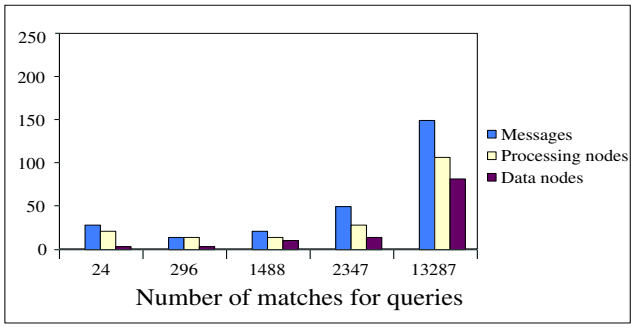
Figures 3.18 (b), 3.19 (b), 3.20 (b) illustrate the fact that the number of processing nodes does not necessarily depend on the number of matches. For example, resolving a query with 160 matches (query6, Figure 3.18 (b)) can be more costly than resolving a query with 2600 matches (query1, Figure 3.18 (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 the tree is pruned during query processing, 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 [73] 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 [63], one would have to know all the matches a priori and look them up individually.




(a)




(b)

Figure 3.20: Experiment 1, 3D: (a) results for range queries; (b) results for range queries, the system size is 4700 nodes and $10^6$ keys.

**Experiment 2 - Constant System Size and Increasing Number of Data Elements**

In the second experiment, the size of the system remained constant as the number of keys and data elements in the system increased. In this experiment, the system size was fixed at 2500 nodes while the number of stored keys increased from $2\times10^5$ to $10^6$. The results for the different query types are presented below.

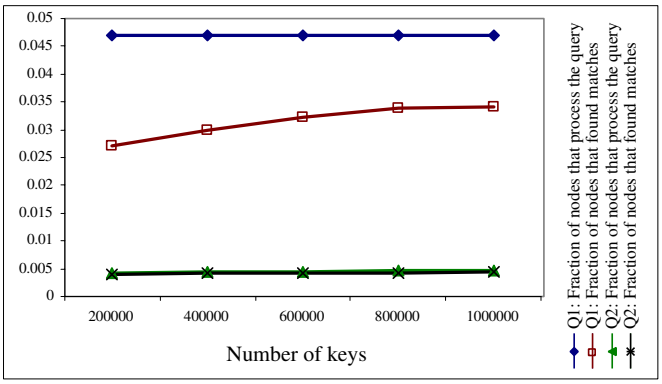



Figure 3.21: Experiment 2, 2D: results for query types Q1 and Q2.

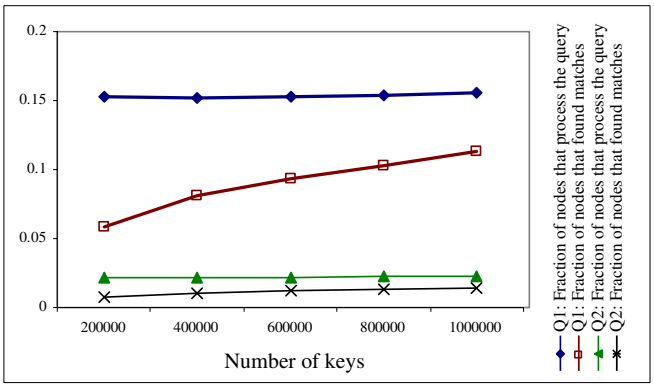



Figure 3.22: Experiment 2, 3D: results for query types Q1 and Q2.

A set of queries of type Q1 and Q2 were tested. For each query the number of processing nodes and the number of data nodes were measured. The results were averaged and normalized. Figures 3.21 and 3.22 plot the results for the two types of queries, for 2D and 3D keyword spaces. The vertical axis represents the percentage of nodes that participated in the query resolving process. The horizontal axis plots the number of keys in the system. As expected, the percentage of data nodes increased

slowly as the number of keys increased. The percentage of processing nodes remained almost constant. Also, the number of data nodes was close to the number of processing nodes, demonstrating the efficiency of the query engine.

As illustrated in Figures 3.21 and 3.22, the results for query type Q2 have the same patterns as those for query type Q1, but are significantly better in their values. This is, once again, because query optimization and pruning are most effective when both keywords are at least partially known

The 3D results have a similar pattern as the 2D results, the only difference is in the magnitude of the results. The reason for this behavior was explained in Section 3.6.1.

**Experiment 3 - Increasing System Size and Constant Number of Data Elements**

In the third experiment, the size of the system increased while the numbers of keys and data elements in the system remained constant. In this experiment, the system increased from about 1000 to 5400 nodes and the number of stored keys was 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 remained the same for different system sizes. Also, note that the keys/node ratio is inversely proportional to the number of nodes.

As in the case of experiments 1 and 2, a set of queries of type Q1 and Q2 were used and the number of processing and data nodes were measured for each query. The results were averaged and normalized. Figure 3.23 presents results for the 2D case, and Figure 3.24 for the 3D case.

The results are very similar to those presented in Section 3.6.1 for Experiment 1. With the number of data elements in the system being constant, as the size of the system grows, the clusters associated with a query will be scattered across a larger number of nodes. As a result, the number of processing nodes and data nodes increases (e.g., in Figure 3.23, 7% out of 1050 nodes is less than 3% out of 5400 nodes). However, as Figures 3.23 and 3.24 show, the percentage of nodes involved in solving a query decreases as the size of the system increases. This shows that the number of processing and data nodes grow slower than the size of the system, and it can be concluded that the

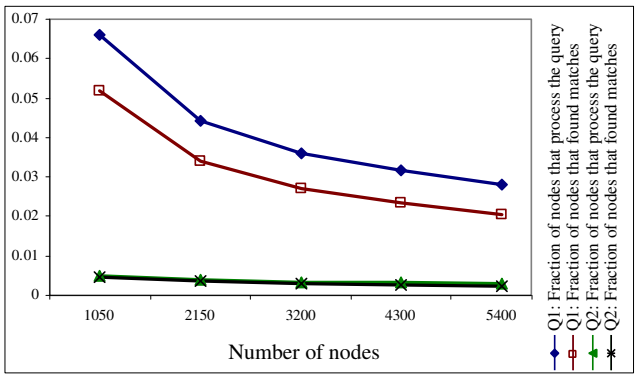



Figure 3.23: Experiment 3, 2D: results for query types Q1 and Q2.

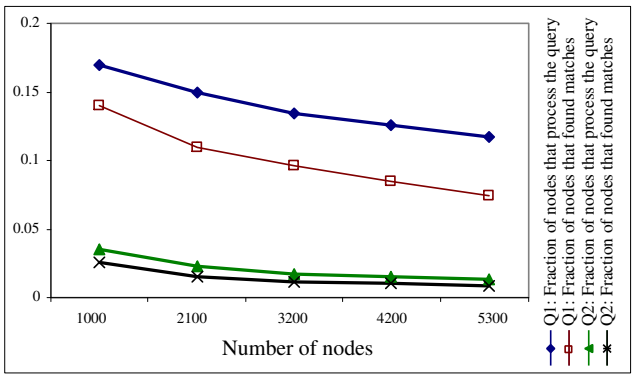



Figure 3.24: Experiment 3, 3D: results for query types Q1 and Q2.

system is scalable. This also shows that the load balancing algorithms were successful in distributing load. Load balancing algorithms are separately evaluated in the next section.

### Evaluating the Load Balancing Algorithms

#### Cost of Load Balancing at Node Join

The cost of load balancing at node join is a small part of the total cost of the join operation. With load balancing, the join operation has to identify a loaded part of the system, and place the node there. The cost for a join with load balancing is the cost to identify the best successor, O($n$ log N), where $n$ is the number of join messages sent, in addition to the cost of updating the finger tables, O($\log^2$N). Since $n$ is bounded by a constant, the overall cost to join with load balancing is O($\log^2$N), which is the same as the cost of a simple join. N is the number of nodes in the system.

### Cost of Load Balancing at Runtime

Squid uses two runtime load balancing algorithms. The first algorithm is run between neighboring nodes and may require the node identifier to be changed. As a result the routing tables at several other nodes in the system have to be updated. A node changing its identifier has to leave the system using its old identifier, and to join it again using the new identifier. The cost for each of these operations is $O(\log^2 N)$, and the cost of this load balancing algorithm is $O(\log^2 N)$.

The second runtime load balancing uses virtual nodes. A virtual node that migrates to a different physical node changes its address (e.g., IP address) while keeping the same identifier. The finger tables at several other nodes in the system have to be updated at a cost of $O(\log^2 N)$. The cost of this load balancing algorithm is the cost of locating the physical node with a light load, which is $0(m)$, where $m$ is the length of the finger list and is a constant, plus the cost of updating the finger tables of other nodes, $O(\log^2 N)$. The overall cost of the operation remains $O(\log^2 N)$.

Note that we have experimentally evaluated the cost of the two load-balancing steps and they match the analytical results presented above.

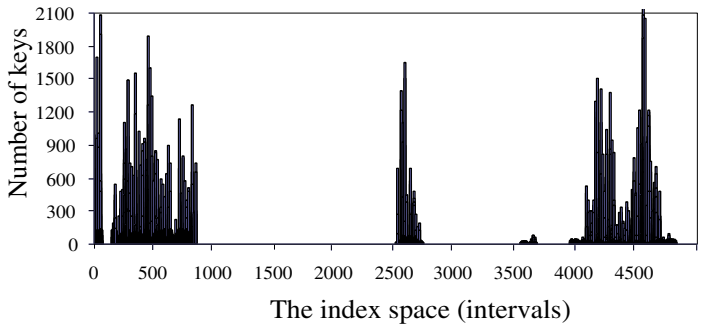



Figure 3.25: The distribution of the keys in the index space. The index space was partitioned into 5000 intervals. The Y-axis represents the number of keys per interval.

### Quality of the Load Balance Mechanisms

The quality of the load balance achieved by the load balancing operations is evaluated for the initial distribution of data elements shown in Figure 3.25. As expected, the original distribution is not uniform. The load balancing step at node join helps to

match the distribution of nodes to the distribution of data. The resulting load balance is plotted in Figure 3.26. While the resulting load distribution is better than the original distribution in Figure 3.25, this step by itself does not guarantee good load balance. However, when it is used in conjunction with the runtime load-balancing steps, the resulting load balance improves significantly as seen in Figure 3.27. The load is almost evenly distributed in this case.

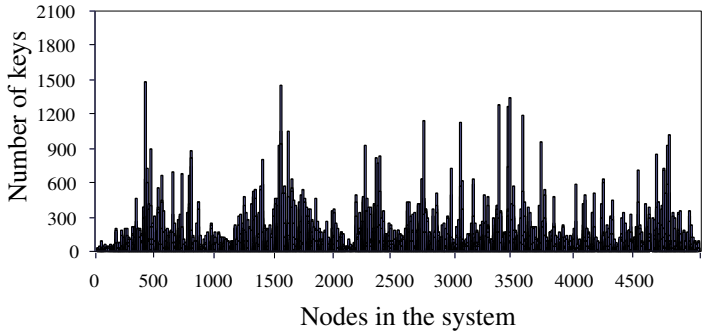



Figure 3.26: The distribution of the keys at nodes when using only the load balancing at node join technique.

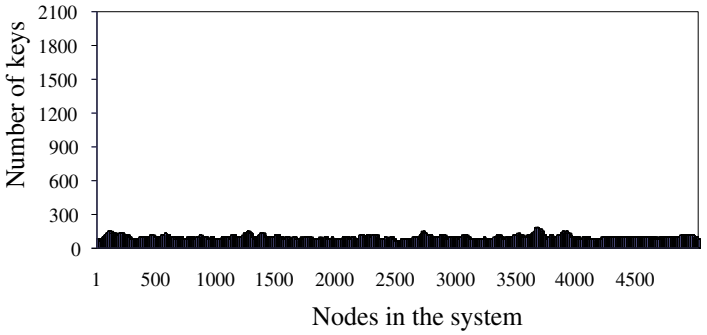



Figure 3.27: The distribution of the keys at nodes when using both the load balancing at node join technique, and the local load balancing.

### 3.6.2 Evaluation Using the Squid Prototype

The current implementation of Squid builds on Project JXTA [77], a general-purpose peer-to-peer framework.

JXTA defines concepts, protocols, and a network architecture. JXTA concepts include peers, peergroups, advertisements, modules, pipes, rendezvous and security. JXTA defines protocols for: (1) discovering peers (Peer Discovery Protocol, PDP), (2)

binding virtual end-to-end communication channels between peers (Pipe Binding Protocol, PBP), (3) resolving queries (Peer Resolver Protocol, PRP), (4) obtaining information on a particular peer, such as its available memory or CPU load (Peer Information Protocol, PIP) (5) propagating messages in a peergroup (Rendezvous protocol, RVP) and (6) determining and routing from a source to a destination using available transmission protocols (Endpoint Routing Protocol, ERP). The JXTA architecture builds on three layers, a core layer, for essential common functionalities, a service layer, for additional pluggable/unpluggable behaviors, and an application layer for end-to-end high level control.

The Chord overlay network and the indexing and query engine are implemented as event-driven JXTA services. Each service registers itself as a listener for specific messages, and gets notified when a corresponding event is raised. Figure 3.28 presents the Squid stack.

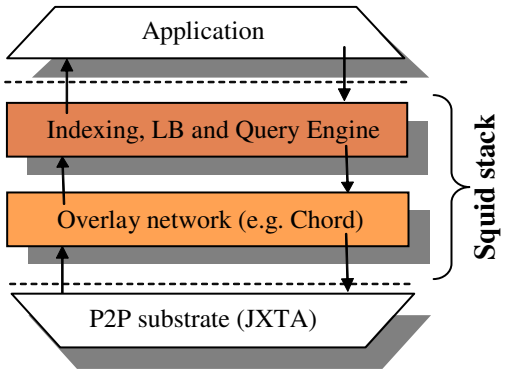



Figure 3.28: Squid Stack - Overview

Squid is deployed incrementally. A joining node uses the Chord overlay protocol and becomes responsible for an interval in the identifier space. The overall operation of the Squid overlay consists of two phases: bootstrap and running. During the bootstrap phase (or join phase) messages are exchanged between a joining node and the rest of the group. During this phase, the joining node attempts to discover an already existing node in the system and construct its routing table. The joining node sends a JXTA discovery message to the group. If the message remains unanswered after a set duration (in the order of seconds), the node assumes that it is the first in the system. If a node

responds to the message, the joining node follows the Chord protocol for node joins, sending a Chord join message and identifying its successor on the ring. The successor helps the new node update its routing tables.

The running phase consists of a stabilization and a publish-query mode. The purpose of the stabilization mode is to ensure routing tables are up to date, and to verify that other nodes in the system have not failed or left the system. In publish-query mode, a node initiates queries received from the application layer and responds to queries issued by other nodes in the system. Data publishing is also done at this level.

Squid was experimentally evaluated on a Linux cluster consisting of 64 1.6 GHz Pentium IV machines and an 100 Mbps Ethernet interconnection. Each of the 64 nodes acted as a peer. The overheads at each layer of the stack were measured. The experiments are presented below.

### Overlay Network Layer

This experiment measured the latency for peer lookup in the overlay network as a function of the size of the system. To get an accurate measurement of the latencies a single peer was run on each node of the cluster and each peer sent messages to a randomly selected destination peer. Each message required an overlay lookup operation. Average times are plotted in Figure 3.29. The graph shows that the average elapsed time is not affected much by the linear growth of the size of the system, validating the scalability of the overlay network lookup operation and the Chord routing algorithm.

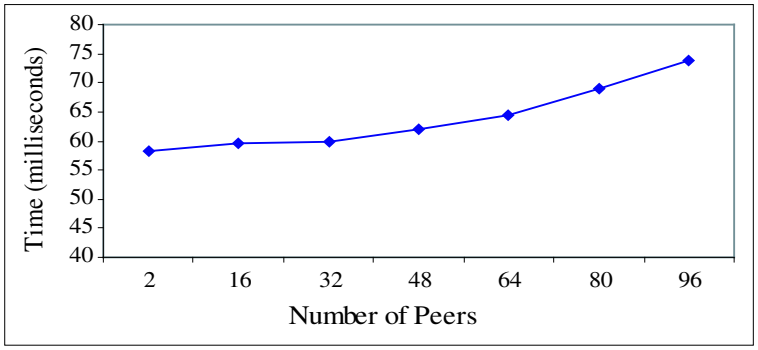



Figure 3.29: Overlay network lookup overhead (Chord).

### Query Engine

This experiment measured the overhead of the query engine using complex keyword tuples. Three sets of queries were used; the first containing wildcards, the second containing ranges, and the third containing both wildcards and ranges. The routing overheads at the query engine were measured at each peer and averaged. The results are plotted in Figure 3.30. The measured overhead includes times for cluster refinements and subcluster lookup. As Figure 3.30 shows, the overhead grows slowly and at a much smaller rate than the system size. This demonstrates that Squid can effectively scale to large numbers of nodes while maintaining acceptable routing times. As expected, the routing times are high for queries with wildcards as they involve a larger number of clusters and correspondingly larger number of nodes.

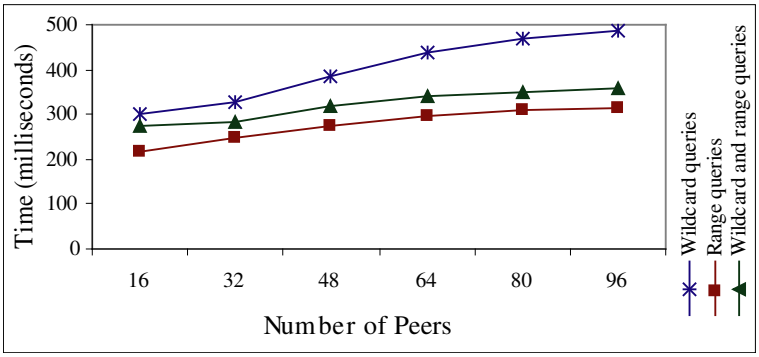



Figure 3.30: Query engine overhead.

# Chapter 4

# Analyzing the Search Characteristics and Performance of Squid

Squid, a structured keyword search system, supports complex queries and provides search guarantees. The search characteristics and performance of structured search systems largely depends on two factors: (1) the indexing scheme used and its ability to preserve information locality, and (2) the efficiency of the distributed query engine and its ability to minimize the number of additional nodes and messages involved in a query.

Data lookup systems typically use indexing mechanisms that distribute the data uniformly across the nodes to achieve load balance. Such an indexing scheme destroys data locality, and as a result, make resolving complex queries very expensive. To support complex queries containing wildcards and ranges the index should preserve information locality. In Squid, data elements are described using multiple keywords. Two data elements are local if their keywords are lexically close. The index preserves locality if these data elements are also close in the index space. Squid uses the Hilbert Space Filling Curve (SFC) indexing scheme, which preserves locality for multiple keywords.

The efficiency of the distributed query engine is also very important. In an ideal case a query should only involve those nodes that store data elements relevant to the query. However, in P2P systems query routing involves additional nodes. The query engine should minimize the number of these additional nodes as well as the messages involved. Further, the query resolution, rather than being done at a single node, should be distributed.

In this chapter we analyze the search characteristics and performance of Squid. Our analysis shows that in large systems, for a generic query matching $p\%$ of the data, the number of nodes with matching data approaches $p\%$ of all nodes in the system. We also show that by optimizing the search process using recursive query refinement and

pruning, the number of nodes involved in the querying process is close to the number of nodes that store data matching the query.

## 4.1 Related Work

Squid uses SFCs to map a multidimensional keyword space to a one-dimensional index space, since they have been shown to preserve locality [41]. The locality-preserving or clustering property of SFCs has been studied in many papers. A key goal of these studies was to find the "best" curve with the "best" locality for an application domain. Jagadish [30] studied the clustering properties of several SFCs, considering a 2-dimensional space, and 2x2 range queries, concluding that the Hilbert SFC was the best to preserve locality. Bugnion et al. [7] analyzed the distribution of inter-cluster intervals, and the number of clusters, for 2-dimensional range queries. Faloutsos and Roseman [21] conclude that the Hilbert SFC achieves better clustering than other methods, for every situation tried. Moon et al. [41] give an asymptotic formula for the clustering property of the Hilbert SFC, for queries in a $d$-dimensional space.

## 4.2 Analysis of the Search Characteristics of SFC-based Indexing within Squid

This section analyses the search characteristics of the SFC-based index used by Squid. Specifically, the analysis proves that for a query $q$, representing $p$% of the data elements stored in the system, the number of nodes that have matching data elements approaches $p$% when the total number of nodes is sufficiently large. The symbols used in the analysis are summarized in Table 4.1.

To summarize the design of Squid described in Chapter 3, the semantic keywords used to augment data elements in Squid define a multi-dimensional keyword space. This space is discrete where each point or cell represents a specific combination of keywords and may or may not store data elements. A one-dimensional SFC is used to thread the space and to index the data elements, resulting in a series of cells, some with data elements and some empty. The SFC is mapped onto the overlay network of

| Symbol | Definition |
|---|---|
| $d$ | Dimensionality of the keyword space |
| $k$ | Maximum approximation level for the SFC |
| $n$ | Number of nodes in the system |
| $q$ | Generic query |
| $p$ | Percentage of data the query represents at level $k$ of approximation |
| $b$ | The level of approximation for the SFC where a node stores approximately a cell |
| $N_q$ | Number of nodes with matching data for query $q$ |

Table 4.1: Definition of symbols

nodes. Nodes are assigned identifiers from a one-dimensional identifier space, and each identifier is assigned a span of the SFC.

The queries used in the analysis are rectilinear polyhedrons [41], i.e., any *(d-1)*-dimensional polygonal surface of the query is perpendicular to one of the $d$ coordinate axis. Further, the analysis assumes that the system is load balanced and each node stores approximately the same number of data elements. Note that the distribution of data elements in the $d$-dimensional space (and on the curve) may vary, depending on the nature of the data indexed. Further, the density of data elements in different regions of the space also varies, i.e., the space may be more or less sparsely populated with data in different regions. The analysis first considers the general case where the distribution of data elements in the $d$-dimensional space is unknown and derives bounds on the number of nodes with matching data for an arbitrary query. It then tightens this bound for the case when data stored in the system is uniformly distributed on the curve.

### 4.2.1 Distribution of Data not Known

Let $k$ be the maximum approximation level of the curve, i.e., the number of times that the curve can be refined. This implies that the keyword space is partitioned into $2^{kd}$ cells and the curve is $2^{kd}$ cells long. Let $[0, 2^{kd})$ be the identifier space for the nodes in the overlay, assuming that a node stores at least one SFC cell. The upper bound for the number of nodes in the system is $2^{kd}$.

Finally, let queries be expressed at level $k$ of refinement. This is a natural choice since the maximum approximation level for the curve is $k$ and a query can be refined at most $k$ times. As the system is load balanced, each node has the same number of data cells, and a different number of empty cells, which depends on the data distribution. For simplicity, a 2-dimensional keyword space is used to illustrate the reasoning process in the analysis below. However, the analysis applies for all values of $d$.

$$n_1 \times n_2 = p\% \tag{4.1}$$

$$p\% \leq N_q \leq a\% \tag{4.2}$$

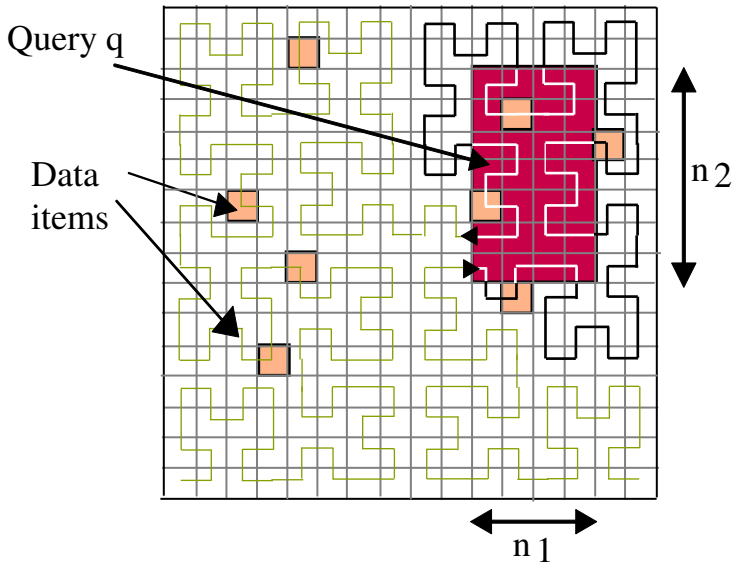



Figure 4.1: Query $q$ defines segments on the curve: the $c$-segments are colored in white and the $s$-segments are colored in black. The light colored cells represent data elements.

For $d = 2$, the generic query $q$ is a rectangle, $n_1 \times n_2$, where $n_1$ is the dimension along $x$-axis, and $n_2$ the dimension along $y$-axis. Let the query $q$ match $p\%$ of the total data stored in the system, i.e., the region $n_1 \times n_2$ defined by the query contains $p\%$ of the total data in the space (see equation 4.1). The curve enters and exits the region defined by the query multiple times and the segments of the curve that lie inside the region are the clusters of interest. These clusters are termed as $c$-segments, and the parts of the curve between successive $c$-segments lying outside the region are called

$s$-segments. As a result, each query defines a continuous span of the SFC, which is a sequence $S$: $c_0 s_1 c_1 ... s_m c_m$ where $m$ is the number of $s$-segments. Figure 4.1 shows an example of a query and the sequence of segments defined by the query for the Hilbert SFC.

Since the SFC index is mapped to successive nodes of the overlay, the sequence $S$ will be mapped to a successive subset of nodes. Further, $S$ will index at least $p$% of the data since the $s$-segments may also index data. Assume that $S$ indexes $a$% of the data. This data will be stored on approximately a% of the nodes (see Appendix A for a proof). However, the nodes of interest are the ones that store the $p$% data that matches the query. The actual percentage of nodes storing this matching $p$% data will be between $p$% and $a$% (see equation 4.2), and will depend on how the data is distributed along $S$. Note that there may be nodes that store parts of both $c$-segment and $s$-segment. These nodes will also have to be queried. Figure 4.2 shows two possible mappings of the sequence $S$ to nodes: in (a) both nodes storing sequence $S$ are queried, since they store $c$-segments; in (b) only three nodes out of four are queried as one of them stores only a $s$-segment.

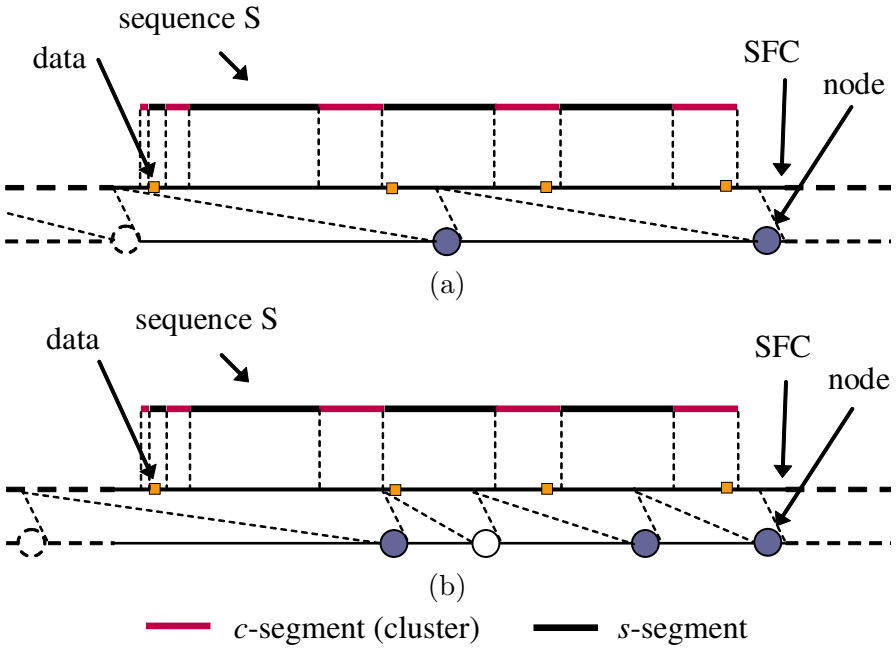



Figure 4.2: (a) The sequence $S$ stored at 2 nodes; both have to be queried, since they store $c$-segments. (b) The sequence $S$ stored at 4 nodes; only 3 nodes have to be queried, the white node stores only a region of an $s$-segment, and no $c$-segment.

As the number of nodes in the system grows and approaches the number of cells in the keyword space, $2^{kd}$, the number of data elements stored at each node becomes less than or equal to 1. As a result, nodes will tend to store *s*-segments or *c*-segments but not both. In this case, the percentage of nodes with matches approaches $p\%$.

$$lim_{n\to\infty} N_q = lim_{n\to 2^{kd}} N_q = n_1 \times n_2 \ ^1$$

The upper bound of $a\%$ in the analysis above is not very precise. However, it is not possible to come with a better upper bound without knowledge of the distribution of data in the *d*-dimensional space (and on the curve), and without considering particular queries. Such a specific case, where the data is uniformly distributed in the space, is considered in the next subsection.

### 4.2.2 Uniform Distribution of Data

Let the data elements be uniformly distributed in the space and consequently on the curve. Assuming that the system is load balanced, each node will store approximately the same number of data elements. Further, assuming that the nodes identifiers are also uniformly distributed in the identifier space, each node will be responsible for approximately the same number of cells in the space.

For simplicity, a 2-dimensional keyword space is once again used to illustrate our reasoning in the analysis below. As for the general case, $k$ is the maximum level of refinement for the curve. The generic query $q$ considered in the analysis is a rectangle in a 2-dimensional space, $n_1 \times n_2$, expressed at level $k$ of approximation. The query has $n_1 \times n_2$ level-$k$ cells, and matches $p\%$ of the total data elements in the system.

Figure 4.3 (a) shows an example of a $n_1 \times n_2$ query, and the corresponding sequence *S*. Figure 4.3 (b) shows the sequence $S$ projected onto the SFC and stored at nodes in the overlay. In this example 4 out of 6 nodes store the *c*-segments (clusters).

The SFC is generated recursively, i.e., at each level of refinement the *d*-dimensional space is partitioned into a number of cells which are traversed by the curve. At level

[1]The keyword space is partitioned into $2^{kd}$ cells. The node identifier space is $[0, 2^{kd})$, and each node stores at least one cell. In this case, $n \to \infty$ is equivalent to $n \to 2^{kd}$.

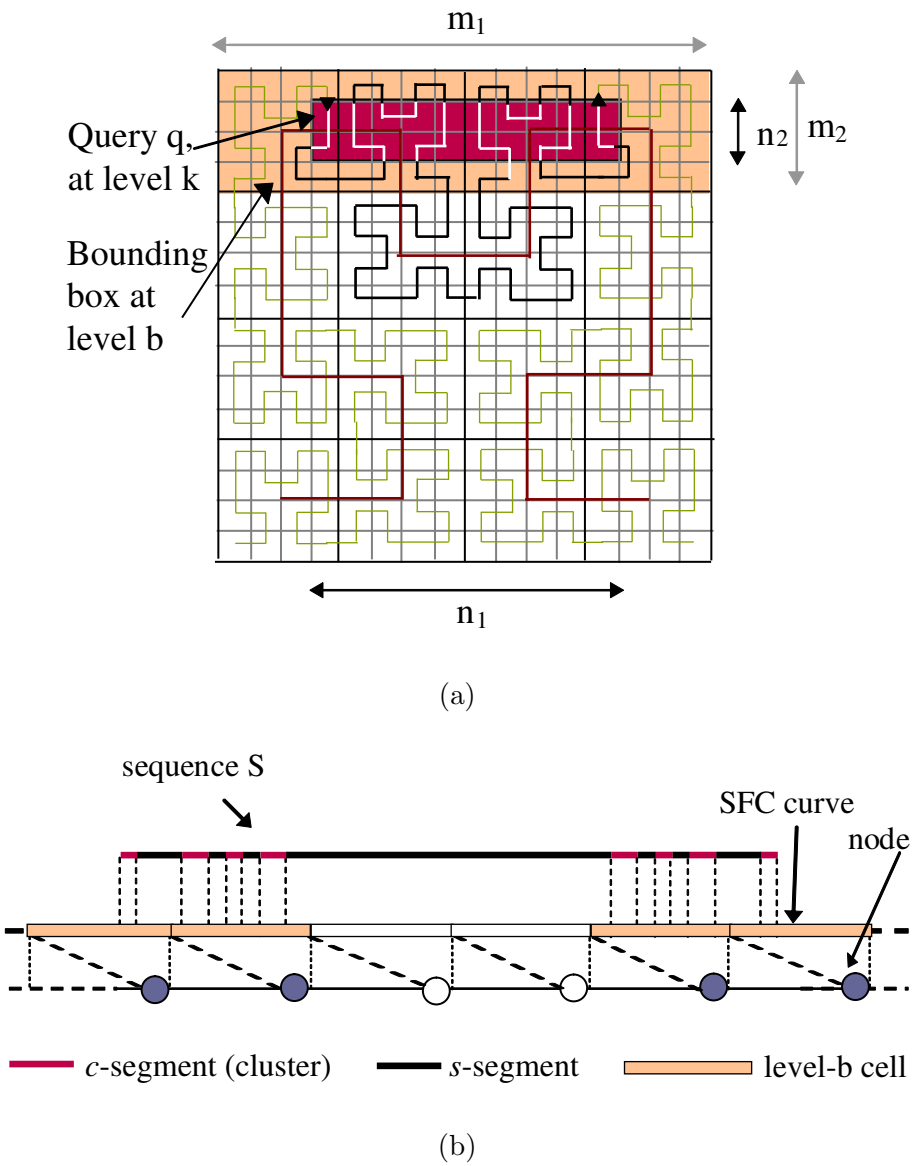



Figure 4.3: (a) Example of a query in a 2-dimensional space, mapped by a Hilbert curve: the query at level 4 of approximation and the bounding box at level 2. (b) The sequence $S$ mapped onto the overlay of nodes, the nodes being uniformly distributed in the identifier space.

$k$ of refinement there are $2^{kd}$ cells, so the curve has $2^{kd}$ cells (points) too. Also, as the data is uniformly distributed in the $d$-dimensional space, there is approximately the same amount of data stored in any cell at any level of approximation less than $k$. Let $n$ be the number of nodes in the system such that, at level $b$ of recursion, $b < k$, each node stores approximately one cell. In this case, $n \simeq 2^{bd}$. As noted earlier, the rectangular query is specified at level $k$ of recursion. At level $b$ of recursion, the number of cells defined by the query will be the same as the number of nodes with data elements that matches the query. The rectangle that the query defines at level $b$ can be regarded as a minimum bounding box for the level $k$ query.

Figure 4.3 (a) illustrates the situation described above. The small rectangle represents the query, and the big one represents the bounding box. The query is defined at level 4 of approximation and has $10 \times 2$ level-4 cells. The bounding box is defined at level 2 of approximation (i.e., $b = 2$) and has $4 \times 1$ level-2 cells, each cell being stored at a node. Figure 4.3 (b) shows how the sequence $S$ is stored at nodes; there are 6 nodes that store $S$, since it is assumed that each node stores a level-2 cell, and the sequence $S$ traverses 6 level-2 cells.

The number of nodes with matching data, $N_q$, can be computed as follows. Let $m_1$ and $m_2$ be the dimensions of the bounding box along the $x$ and $y$ axis respectively. The bounding box has $m_1 \times m_2$ level-$b$ cells. At level $k$ of approximation the $d$-dimensional space has $2^{kd}$ cells. At level $b$ of approximation the $d$-dimensional space has $2^{bd}$ cells. Each level-$b$ cell has $2^{d(k-b)}$ level-$k$ cells, i.e., $2^{2(k-b)}$ cells in a 2-dimensional space. $m_1$ and $m_2$ can be approximated as $m_1 = \lceil \frac{n_1}{2^{k-b}} \rceil$, $m_2 = \lceil \frac{n_2}{2^{k-b}} \rceil$. $N_q$ can be expressed as: [2]

$$
\begin{aligned}
N_q &= m_1 \times m_2 \\
&= \lceil \frac{n_1}{2^{k-b}} \rceil \times \lceil \frac{n_2}{2^{k-b}} \rceil \\
&\simeq \frac{n_1}{2^{k-b}} \times \frac{n_2}{2^{k-b}} \\
&= \frac{n_1 \times n_2}{2^{2(k-b)}}
\end{aligned}
$$

[2]The bounding box is approximated without rounding up since the numbers involved are very large. The simple example shown in the figure is for illustration purposes only.

The $d$-dimensional keyword space has $2^{kd}$ cells, and the node identifier space is [0, $2^{kd}$), each node storing at least a cell. In this case, $n \rightarrow \infty$ is equivalent to $b \rightarrow k$.

$$lim_{n\rightarrow\infty} N_q = lim_{b\rightarrow k} N_q = n_1 \times n_2$$

$=>$ $p$% of the nodes are involved.

For a $d$-dimensional space, we have the following:

$$N_q \simeq \prod_{i=1,d} \frac{n_i}{2^{k-b}} = \frac{\prod_{i=1,d} n_i}{2^{d(k-b)}}$$

$$lim_{n\rightarrow\infty} N_q = lim_{b\rightarrow k} N_q = \prod_{i=1,d} n_i$$

$=>$ $p$% of the nodes are involved.

The upper bound for the number of nodes involved in a query depends on the number of nodes in the system. If the system has only one node, it is obvious that 100% of the nodes will be involved. For a system with $n = 2^{bd}$ nodes, the query $q$ defined above will have to touch at most the nodes storing the bounding box cells, i.e., $\lceil \frac{n_1}{2^{k-b}} \rceil \times \lceil \frac{n_2}{2^{k-b}} \rceil$. For a $d$-dimensional space, the upper bound is $\prod_{i=1,d} \lceil \frac{n_i}{2^{k-b}} \rceil$.

## 4.3 Experimental Results

This section presents simulation results that validate the analysis presented in Section 4.2 and demonstrate the effectiveness of the Squid query engine. Two sets of data were used in this evaluation: (1) synthetically generated data that has an uniform distribution on the curve, and (2) real data consisting of CiteSeer HTML files [72]. The objectives of the experiments were: (1) to demonstrate that, as the number of nodes in the system grows, the percentage of nodes with matches for a query approaches the percentage of the data matched, and (2) to demonstrate that the optimization used by the query engine is successful at reducing the number of clusters that have to be generated for a query, and that the number of extra nodes involved in the process is small.

The experiments were performed using the simulator described in Chapter 3. All queries were issued on a load-balanced system where each node stored the same quantity of data. It was assumed that the system stores only the index (SFC numerical index, keywords, and a reference to the data), and not the actual data. Only unique data was used, each data element was described uniquely by keywords, and had a unique SFC index. Queries containing ranges, partial keywords and/or wildcards were evaluated. The results were grouped by query coverage, i.e., the percentage of data matched, and the average was computed for each group.

### 4.3.1 Evaluation Using Synthetically Generated, Uniformly Distributed Data

The first set of experiments used a 3-dimensional keyword space composed of $2^{24}$ cells and systems of sizes $10^3$, $10^4$, $10^5$ and $10^6$ nodes. The system was populated with $2^{24}$ unique, synthetically generated data elements that completely populated the space and resulted in a uniform distribution.

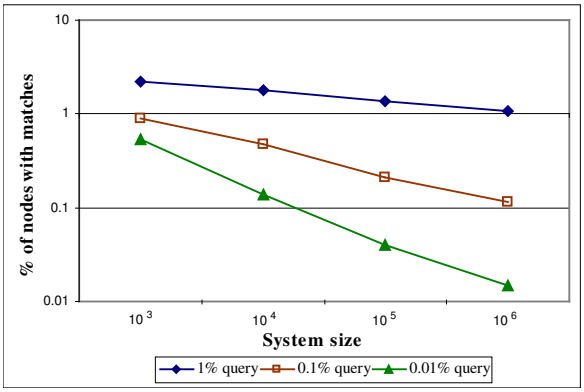



Figure 4.4: Uniformly distributed synthetic 3D data, queries with coverage *1%*, *0.1%* and *0.01%*, plotted using a logarithmic scale on both axis.

Three classes of queries were used in the evaluation providing coverage of *1%*, *0.1%* and *0.01%* respectively. A query with a coverage of *1%* matched 1% of the total data stored in the system. The same set of queries was used for each system size. The results are plotted in Figure 4.4 using a logarithmic scale on both axis: the $x$-axis plots the size of the system, and the $y$-axis plots the percentage of nodes with data

matching the query. The percentage of nodes with matches decreases as the size of the system increases, and for a system with $10^6$ nodes, the percentage of the nodes with matches is approximately the same as the percentage of data the query matches. This is consistent with our analysis, since the $10^6$ node system is close to the number of cells in the keyword space, i.e., $2^{24} \simeq 1.6 \times 10^7$.

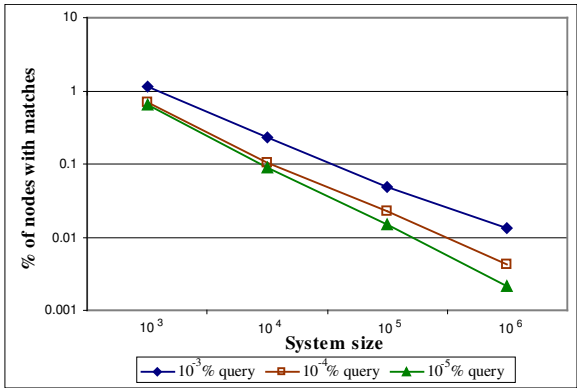



Figure 4.5: Uniformly distributed synthetic 5D data, queries with coverage $10^{-3}\,\%$, $10^{-4}\,\%$ and $10^{-5}\,\%$, plotted using a logarithmic scale on both axis.

A similar set of experiments were carried out using a 5-dimensional keyword space composed of $2^{40}$ cells, for the same four system sizes as above, i.e., $10^3$, $10^4$, $10^5$ and $10^6$ nodes. Three categories of queries were tested with coverage of $10^{-3}\,\%$, $10^{-4}\,\%$ and $10^{-5}\,\%$. The results are plotted in Figure 4.5 using a logarithmic scale on both axes. As the figure shows, the percentage of the nodes with matching data decreases as the system size grows. However, the difference between the percentage of the nodes with matching data and the coverage of the query is greater than in the 3-dimensional case. There are two reasons for this behavior: (1) The maximum system size tested is still small compared with the number of cells in the keyword space - the maximum size tested is $10^6$, while the number of cells is $2^{40} \simeq 10^{12}$. (2) The clustering property of the curve deteriorates as the dimensionality of the space grows. It was proven [41] that the Hilbert SFC begins to loose its clustering property beyond 5 dimensions.

### 4.3.2 Evaluation Using Real Data from CiteSeer

The second set of experiments were performed using real data from CiteSeer [72] consisting of HTML files describing scientific papers. A set of keywords was extracted from the title and abstract of each paper, using a simple Information Retrieval (IR) tool. $4 \times 10^5$ HTML files were indexed using the Hilbert SFC based on the extracted keywords. The experiments were carried out in a 3-dimensional keyword space, composed of $2^{48}$ cells, and three system sizes with $10^3$, $10^4$ and $10^5$ nodes. A $10^6$ nodes system was not used as the number of data elements to be stored in the system is $4 \times 10^5$, and having $10^6$ nodes would lead to nodes without data. Three classes of queries with coverage of *1%*, *0.1%* and *0.01%* were used.

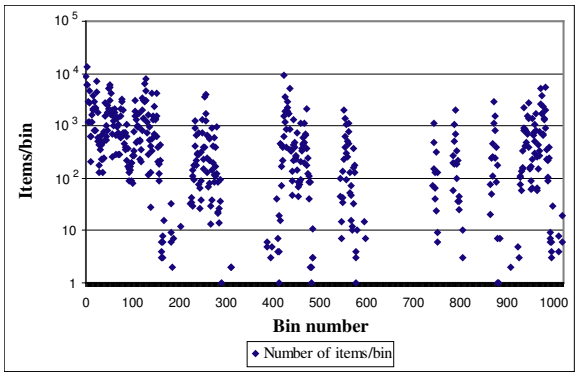



Figure 4.6: Distribution of CiteSeer data on the Hilbert SFC. The curve has $2^{48}$ points which are divided into 1000 bins. The $y$ axis plots the number of data elements in each bin using a logarithmic scale.

The distribution of the data on the SFC curve is shown in Figure 4.6. Since the curve has $2^{48}$ points, it is divided into 1000 bins in the plot. The $x$ axis plots the bin number and the $y$ axis plots the number of data elements in each bin. As the figure shows, the data is not uniformly distributed. Note that there are empty spaces on the curve, which are primarily due to the fact that the curve is in base 32 (the basic step of recursion is the base-2 Hilbert SFC refined 5 times) while the keywords using the English alphabet are in base 26. However, even if these empty spaces are ignored, the distribution is far from uniform.

The results from the simulation are plotted in Figure 4.7. The plots show that

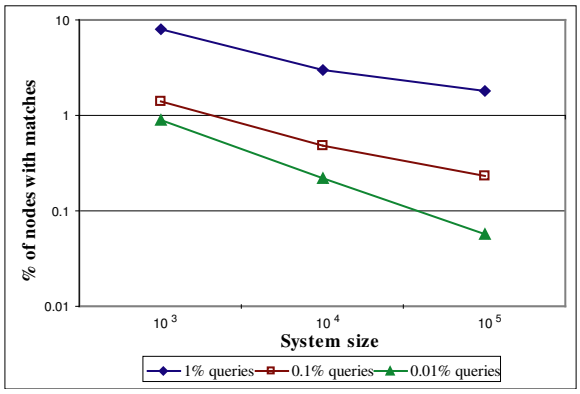


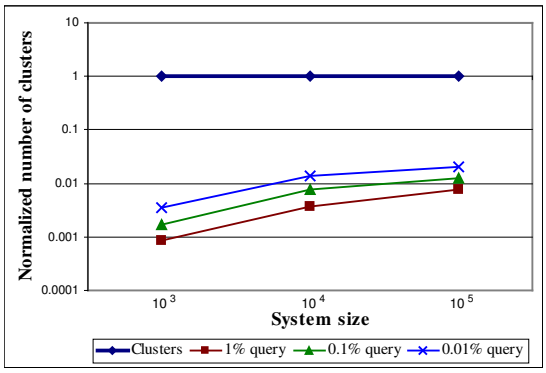

Figure 4.7: CiteSeer 3D data, queries with coverage *1%*, *0.1%* and *0.01%*, plotted using a logarithmic scale on both axis.

the percentage of nodes queried decreases as the size of the system increases, and it approaches the percentage of data that the query matches. This, once again, is consistent with our analysis. Note that the plots are similar to those obtained for the synthetic uniformly distributed data using a 3-dimensional space (see Figure 4.4). The difference between the uniformly distributed and the CiteSeer data is the rate at which the percentage of nodes with matches approaches the query coverage. This convergence is faster for the uniformly distributed data. This is due to the fact that, even though the size of the system is the same in the two experiments, for the uniformly distributed data case the length of the curve and the number of cells in the keyword space is $2^{24}$, while for the CiteSeer data, it is $2^{48}$.

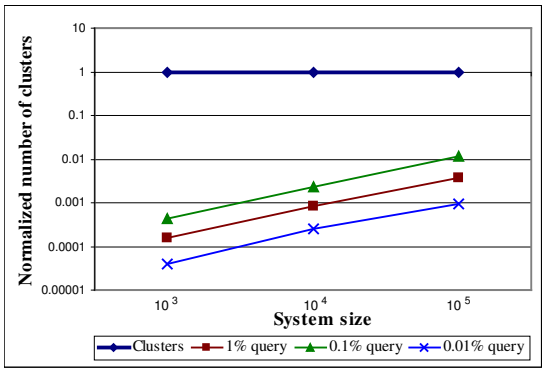
### 4.3.3 Query Engine Optimization

The experiments presented in this section evaluate the optimization strategies used by the query engine. The experiments were carried out for both the uniformly distributed synthetic data and the real data from CiteSeer, using a 3D keyword space, and the same settings as the ones used in the experiments above. Measurements included the number of clusters generated for a query and the number of nodes involved in resolving a query, with and without the optimization.

The number of clusters for queries with coverage of *1%*, *0.1%* and *0.01%* are plotted in Figure 4.8. Two values were measured for each query; (1) the number of clusters




(a)



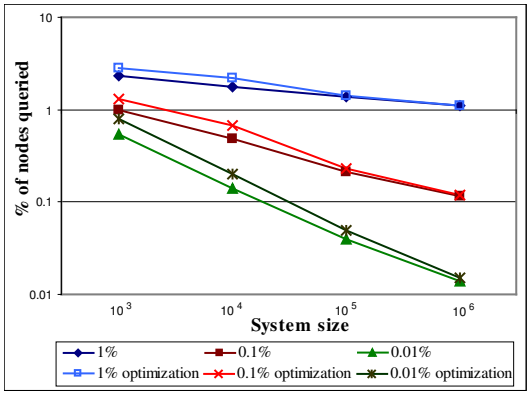

(b)

Figure 4.8: Number of clusters, normalized and plotted on a logarithmic scale. The line at y=1 represents the clusters that the query defines on the curve. The other lines represent the clusters generated using the optimized query engine. (a) Uniformly distributed synthetic 3D data, queries with coverage of *1%*, *0.1%* and *0.01%*; (b) CiteSeer 3D data, queries with coverage of *1%*, *0.1%* and *0.01%*.

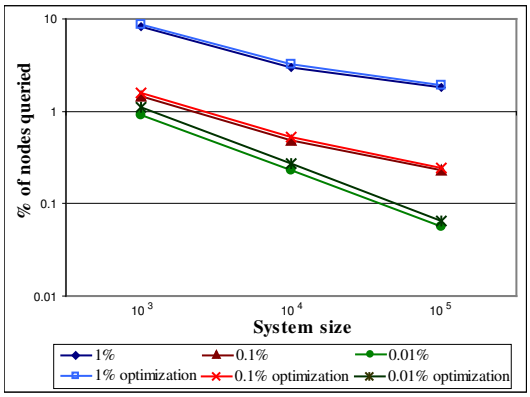
defined by the query on the curve with no optimization, and (2) the number of clusters actually generated in the system when the optimization is used. The values obtained were averaged for each query coverage group and system size. Finally, the results for each query group were normalized and plotted on a logarithmic scale.

As Figure 4.8 shows, the number of clusters that have to be generated is substantially reduced when the optimization is used. For example, for a query with *1%* coverage and a system size of $10^4$, only 0.35% of the clusters need to be generated for the synthetic data (see Figure 4.8 (a)), and 0.08% for the CiteSeer data (see Figure 4.8 (b)). The pruning is more effective for CiteSeer data due to the size of the keyword space. The CiteSeer data uses a longer curve than the synthetic data ($2^{48}$ rather than $2^{24}$), while

the system size is the same in both cases. As a result, a longer span of the curve is stored at a node in the CiteSeer case, and the probability of finding a larger number of the clusters for a query at a node is higher.




(a)




(b)

Figure 4.9: Percentage of nodes queried, with and without the optimization, plotted on a logarithmic scale. (a) Uniformly distributed synthetic 3D data, queries with coverage of *1%*, *0.1%* and *0.01%*; (b) CiteSeer 3D data, queries with coverage of *1%*, *0.1%* and *0.01%*.

Figure 4.8 (a) plots the results for the synthetic data. It can be seen that the clusters are pruned more effectively as the coverage of the query grows. This is not the case for the CiteSeer data as seen in Figure 4.8 (b). This is due to the relation between the percentage of the data the query matches and the percentage of the space (or curve) the query defines. In the uniformly distributed case, the relation is 1:1, i.e., if the query matches $p$% of the data, the region in space defined by the query will represent approximately $p$% of the total space. The larger the query region, the bigger

the number of clusters it defines, and the greater the probability that more of them are found at one node. In the case of the CiteSeer data with an unknown distribution, a query that represent $p$% of the total data does not, in most of cases, define a region that represents $p$% of the space - there will be regions in the space that are sparsely populated with data elements and other that are densely populated.

The number of nodes involved in query resolution are plotted in Figures 4.9 (a) and (b). The results with optimization and without optimization are plotted on the same graph, using a logarithmic scale on each axis. The graphs show that, as described in Section 4.2, the optimization results in additional nodes being involved in query resolution. However, this overhead induced by the optimization is small (approximately 15% overhead for synthetic data, and 7.5% overhead for CiteSeer data), when compared to the number of clusters that are pruned.

# Chapter 5

# Reliability and Fault Tolerance

P2P systems are large and dynamic, with nodes joining, leaving and failing relatively often. DHT-based structured keyword search systems, such as Squid, have deterministic routing and data index placement. As long as their structure is stable, these systems are reliable, and offer guarantees. The dynamism of the system can make this structure unstable, resulting in incorrect routing behaviors and loss of guarantees.

Nodes that join or leave the system restructure the overlay by updating routing tables at several nodes in the system. They also cause parts of the data index stored at nodes to be redistributed, i.e., a new node takes a part of the successor's index, and a leaving node gives its data index to its successor. Failing nodes break the structure of the overlay and result in the loss of the index they store. In both cases, outstanding queries may be affected as all the nodes with matching data are not identified.

This chapter presents SquidTON, a two-tier overlay designed to tolerate multiple node failures. The failure semantics of a faulty node can be modeled in two ways; the fail-stop model in which a failed node is deleted from the system, and the Byzantine model in which failed nodes are malicious and harm the system. In this thesis we assume fail-stop failure semantic.

Reliability and fault tolerance are addressed at two levels: (1) at the overlay and routing infrastructure level, and (2) at the storage level. At the overlay level, the P2P system should be able to route around temporary and permanent failures. At the storage level, loss of indices stored at nodes should be prevented. SquidTON addresses these requirements by building redundancy into the overlay and by replicating the index.

Note that the mechanisms presented in this chapter specifically address the reliability and integrity of the index data. Reliability of the actual application data in the P2P system involves application-specific issues (e.g., ownership issues) and is assumed

to be handled at the application level.

## 5.1 Related Work

Fault tolerance at the routing level is typically addressed by building redundancy into the overlay network. Chord [63] addresses fault tolerance as follows: each node maintains a "successor list" of its $r$ nearest successors on the ring. If a node's successor dies, the next node from this list is considered as the new successor. Successors are used for routing when all the entries in the node's routing table are detected to be obsolete. CAN [48] addresses fault tolerance by providing alternative paths between nodes (i.e., constructing multiple overlay networks on the same set of nodes). Tapestry [70] and Bayeux [71] create multiple paths between nodes by providing the routing mechanism with equivalent destinations at each routing step.

Loss of data and/or data index due to failures is typically addressed by replicating the data at multiple nodes. In unstructured systems data is usually replicated at the node that requested it [73]. More proactive replication schemes are proposed and evaluated in [12, 13, 38]. Structured systems use controlled replication. For example, Chord [63] stores data elements at its *k+1* successors rather than only at its immediate successor. CAN [48] replicates data using techniques such as creating multiple indices for the same data set, and replicating popular items at the neighbors of the node that is responsible for storing them. Pastry [51] also replicates the data at a set of neighboring nodes. Tapestry [70] inserts replicas of data items using different keys.

## 5.2 Addressing Reliability at the Overlay Network - The Squid Tiered Overlay Network (SquidTON)

### 5.2.1 Architectural Overview

Squid addresses reliability at the overlay level by making the overlay fault tolerant. The key idea is to build redundancy into the routing tables, so that if nodes fail, equivalent ones can be used to forward messages. In this case the routing tables are fatter as each entry stores multiple equivalent destination nodes. The equivalent nodes form a group,

and are connected to form a small overlay, which can be structured or unstructured. Each group is described by a single group identifier that is shared by all the nodes in the group. This group identifier is different from the individual node identifiers, which are unique to each node. Node identifiers are used only for intra-group routing. Nodes use their group identifiers to join a global structured overlay. As a result, the SquidTON overlay is a two-tiered structure, i.e., an overlay of overlays - groups of nodes connected by a global overlay.

As mentioned in Section 3.2, Squid uses a one-dimensional index space, and consequently, a one dimensional overlay is most suitable. This is because such an overlay uses a one-dimensional node identifier space and enables a straightforward mapping of the one-dimensional index to the nodes. The current implementation of Squid uses Chord as its overlay network, and as a result, the two-tier overlay presented in this chapter is described using Chord. However, any other one-dimensional overlay may be used with SquidTON.

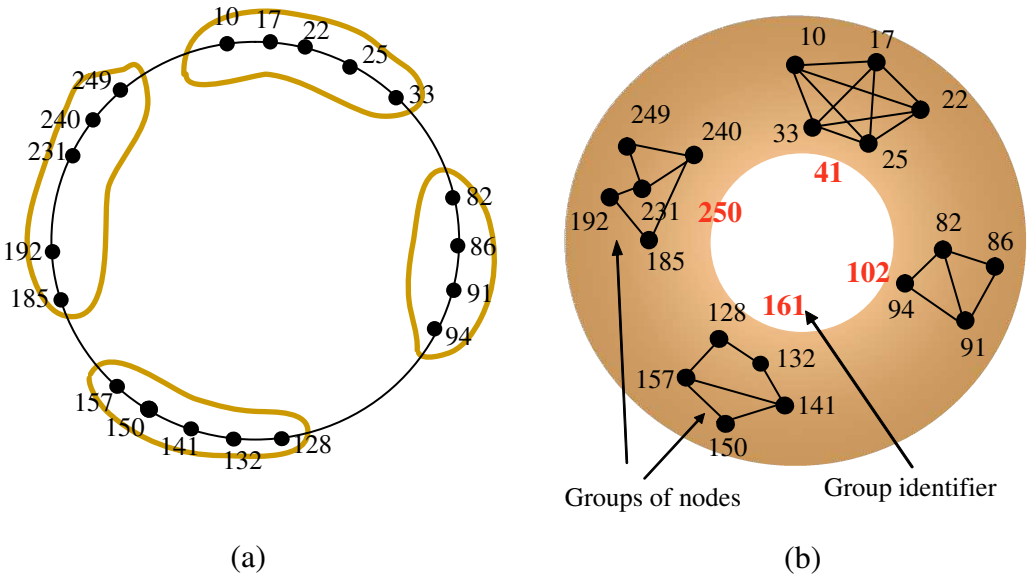



Figure 5.1: (a) Example of a Chord overlay with 19 nodes and an identifier space from 0 to $2^8$-1. Possible grouping of nodes are shown. (b) SquidTON architecture. The disc represents Chord and the black dots are the nodes. The nodes are connected in small overlays (groups), which are connected by Chord. The nodes in a group join Chord using the group identifier. The group identifiers are shown in red in the middle of the disk.

Figure 5.1 (a) shows a one-dimensional overlay (e.g., Chord) with 19 nodes and a

possible grouping of nodes. Figure 5.1 (b) shows the corresponding SquidTON overlay.

The nodes in a group have node identifiers which are consecutive in the identifier space. Note that, as a result of using the load balancing algorithms presented in Section 3.5, the node identifiers tend to follow the distribution of the data.

### 5.2.2 Node Identifiers and Group Identifiers

Node identifiers in SquidTON are the same as node identifiers in Chord and are chosen from an identifier space $[0, 2^m)$. Each node has a unique node identifier.

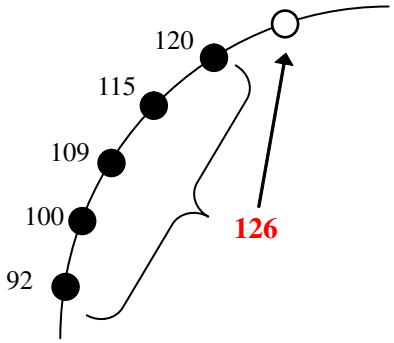



Figure 5.2: Example of a group of nodes and their group identifier. The white node shows where the group identifier fits on the circle. A node with this identifier may or may not exist in the system.

Nodes with consecutive node identifiers in the identifier space form groups, and share a unique group identifier. A group identifier belongs to the same identifier space as a node identifier, i.e., $[0, 2^m)$, and is the highest identifier in a group. Note that while the group identifier typically corresponds to the highest node identifier in the group, it may be higher than any node identifier in the group, e.g., if the node with the highest (group) identifier has failed. This situation will change if a new node joins to replace the failed node. Figure 5.2 presents a group of nodes, and their group identifier.

### 5.2.3 SquidTON Structure and Management

Each node in the SquidTON overlay may belong to two overlay networks at the same time. As a result, each node has two routing tables: a local routing table for routing within the group, and the finger table for Chord. Figure 5.3 illustrates the structure

of a node. The local routing table keeps information about neighboring nodes in the group and its structure depends on the type of overlay used for the group.

The finger table is similar to Chord's finger table, with one modification: instead of storing one finger node per entry, SquidTON stores several fingers for each entry in a *finger list*. The nodes in a *finger list* belong to the same group and share the same group identifier. The nodes in the $i^{th}$ *finger list* of the routing table are the nodes whose group identifier is the successor of the identifier (group identifier + $2^{i-1}$) mod $2^m$, i.e., the same as in Chord, but using group identifiers instead of node identifiers.

The maximum size of the *finger list* is specified a priori. Typically the size of a *finger list* is smaller than the maximum size of a group. As a result, only a part of a group is represented in a particular finger list.

The successor and predecessor entries in the SquidTON finger table are different from Chord. SquidTON maintains a reference to several nodes belonging to the successor group rather than just the next node in the identifier space. The same is true for the predecessors.

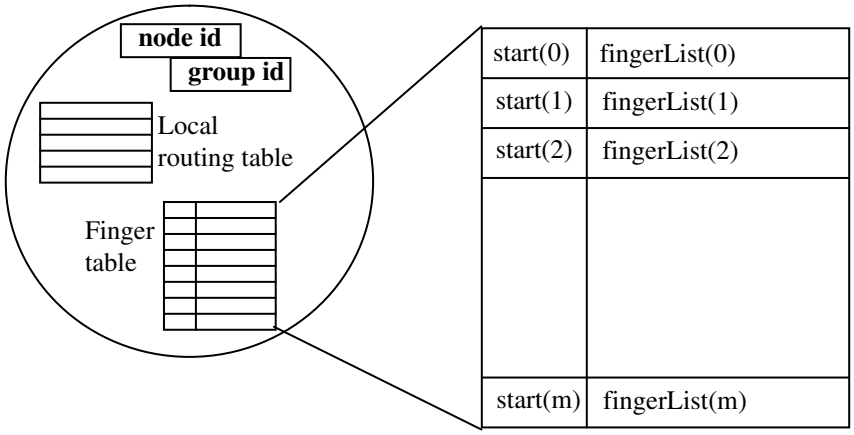



Figure 5.3: SquidTON node.

The rest of this section describes the management of SquidTON.

- **Node Joins**: The node join process is similar to Chord's node join process, with

small differences. In SquidTON, the node first uses Chord to find its corresponding group and then joins the group. The node may or may not join Chord. Before joining the group the node may change its joining identifier (see Section 3.2.1), if required by the load balancing algorithm.

Note that it is not necessary for all incoming nodes to join Chord. Each group of nodes has a set of representatives that are a part of the Chord overlay. If the current number of representatives is sufficient, the node does not join Chord. The node may join Chord at a later time, as explained later in this section.

The cost of a SquidTON join operation is composed of the cost to locate the group, which is O(log N), the cost to join the group, and, for some of the nodes, the cost to join Chord, which is $O(\log^2 N)$. N is the number of groups in the system. The cost to join the group depends on the overlay used within the group. If the group is a completely connected graph, the cost is $O(r)$, where $r$ is the number of nodes in the group. Since $r$ is bounded by a constant (the maximum size of a group), the cost is not significant. As a result, the overall cost to join SquidTON is $O(\log^2 N)$ for the nodes that join Chord, and O(log N) for the nodes that only join a group.

- **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 where N is the number of groups in the system. This cost is the same as that of a join to the Chord overlay.

- **Node Failures**: When a node fails, the finger tables at nodes containing references to the failed node will have invalid entries. Since each entry in the finger tables contains a list of finger nodes, the routing is not affected as long as at least one node in the list is alive. SquidTON detects failed nodes while attempting to route to them. The failed nodes are removed from the routing table. Additionally, each node periodically runs a repair algorithm in which it chooses a random entry in its finger table and updates it by removing failed nodes and adding valid nodes. The process is explained in the next paragraph.

- **Publishing**: Publishing in SquidTON is the same as in Chord, except that the published index is replicated in the group according to the replication algorithm used. Note that only the index is replicated. The actual data is manipulated at the application layer as it is constrained by application specific issues, i.e., ownership issues may prevent data replication at unauthorized nodes. The cost for publishing a data element in the system is the cost to locate the appropriate group, O(log N) plus the cost to replicate the index in the group. If the group has $r$ nodes and full replication is used, the cost of data replication is O($r$). The overall cost for publishing a data element remains O(log N).

- **Routing**: Routing is performed using the algorithm presented in Figure 5.4. After identifying the correct entry in the finger table as in Chord, a node from the corresponding *finger list* is selected. This may be done randomly, using a round-robin algorithm, or some other selection algorithm. If the node is not currently in the system, it is deleted from the finger list and another node is chosen. The process is repeated until a valid node is found. Routing is accomplished in O(log N) hops, as in Chord. If all nodes in the list have failed, the message is routed to the successor group, from which it proceeds to the destination.

```
route(message)
 do
   //select a node from FingerList(i), e.g. randomly
   n = FingerList(i).getNode()
   if (n is alive)
     send(n, message) //send the message to node n
   else
     FingerList(i).remove(n)
 while (n not alive)
 //all nodes in the finger list failed
 if (FingerList(i).size() == 0)
   //send the message to a node in the successor group
   send(successor, message)
```

Figure 5.4: The pseudo-code for routing in SquidTON.

As mentioned before, routing considers the group identifiers instead of node identifiers. A lookup message is sent to a node whose group identifier succeeds the

lookup identifier on the circle. This node may or may not store the desired data element, depending on the replication algorithm used. However, the node can forward the lookup message to the node in its group that contains a replica of the desired data element.

### Finger Table Maintenance

The finger tables at nodes are updated when nodes join and leave the system, during routing where failed nodes are deleted from the *finger lists*, and by the repair mechanism that is periodically run by each node. The nodes that join or leave the system send update requests to other nodes in order to maintain the structure of the overlay. The overlay structure breaks when nodes fail, and is repaired by the overlay repair algorithm which runs in background.

```
repairFingerTable()
   //select an entry from finger table, randomly
   i = getRandomIndex(1, fingerTable.size)
   for (each n ∈ FingerList(i))
     if (n not alive)
       FingerList(i).remove(n)
   //if all nodes in the finger list failed, perform a lookup
   if (FingerList(i).size() == 0)
       FingerList(i).addNode(lookup(start(i))
  //if the finger list is too short, add nodes from the same group
  if (FingerList(i).size() < α)
     n = FingerList(i).getNode()
     refresh(FingerList(i), n)
```

Figure 5.5: The pseudocode for finger table repair algorithm.

Every node runs the repair algorithm periodically. The first step in this algorithm is to check the validity of predecessors and successors. The approach used is similar to that used by Chord's stabilization mechanism [63]. The second step is to update the finger table entries. A random finger list in the finger table is chosen and the references in that finger list are checked. References to failed nodes are removed. If the finger list becomes too small, a *refresh* message is sent to one of the nodes in the list, asking for additional nodes in its group that can be added to the finger list. If the finger list becomes empty, a lookup for that entry is performed (as in Chord), and the finger

list is updated. The pseudo-code of the finger table repair algorithm is presented in Figure 5.5.

SquidTON also uses a "passive" mechanism to refresh finger tables. Each node can update its finger table upon receiving a query or reply message from another node, as the message carries the address of the node where it originated (source node). As messages are short lived there is a high probability that the source node is alive. The source node's group identifier can be used to extend the corresponding *finger list*. However, the *finger list* is not updated if it contains enough alive nodes in it.

### 5.2.4 Management at the Group Level

The nodes in a group can be connected using any type of overlay, unstructured or structured. Our current implementation uses a completely connected graph where each node is connected to every other node in the group. The rest of this section assumes a completely connected graph topology for groups.

The size of a group is constrained by a maximum and a minimum value. These limits are global parameters, and are known by every node in the system. The maximum size is chosen so that a group's routing and local routing table synchronization are efficient. The minimum size is chosen so that the desired degree of replication is achieved at the finger table level and at the stored information level.

The basic operations performed by a group include a group *split* when it becomes too large, and group *dissolve* that merges it with a neighboring group when it becomes too small.

#### Splitting a Group

A group splits when its size reaches the maximum allowed and creates two new groups. Figures 5.6 (a) and (b) illustrate the splitting process. After a split, one of the newly created groups keeps the original group identifier and the other group gets a new group identifier.

In order to maintain a degree of redundancy in the routing tables the resulting groups should have approximately the same size. Further, due to the load balancing

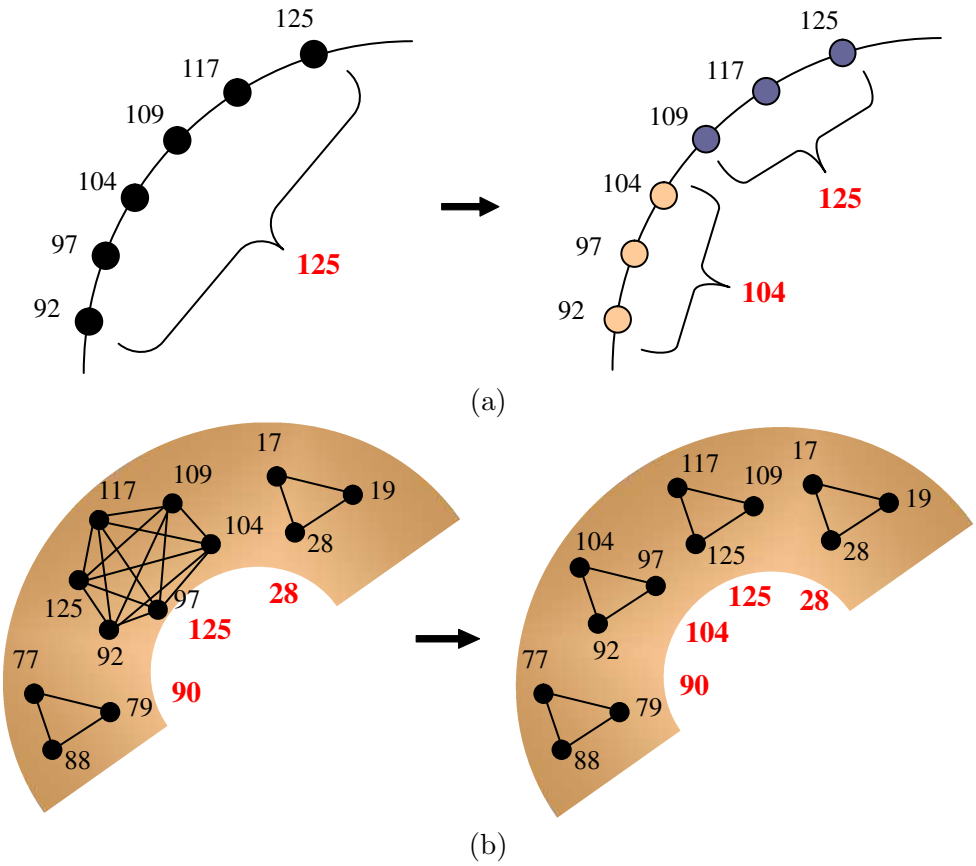



Figure 5.6: Splitting a group: (a) Chord view; (b) SquidTON view.

mechanism each node stores approximately the same number of data indices. As a result, the new groups will split the data approximately in half.

Before splitting, the nodes in the group must agree on the splitting point. Each node chooses the median node identifier in the group as the splitting point. As each node has a complete view of the group, it should be able to choose the splitting point independently. However, the dynamism of the system, with nodes joining, leaving and failing, can result in inconsistency and prevent the nodes from reaching the same conclusion independently. To agree on a splitting point, the group uses the Paxos [34] leader-based consensus algorithm. First, the nodes agree to enter SPLIT mode and postpone all future node *join* and *leave* requests, and then they agree on a splitting point. The Paxos consensus algorithm can also deal with node failure.

After selecting the splitting point, each node runs the splitting algorithm which consists of the following steps: (1) adjust the group identifier, if needed, (2) update the

local routing table, (3) update Chord and (4) delete the data indices that now map to the other group. Figure 5.7 lists the pseudo-code for the splitting algorithm. The steps of the algorithm are described in detail below.

```
group_split(splitPoint)
 //change the group identifier for this node, if necessary
 oldGroupID = groupID
 if (nodeID <= splitPoint)
     groupID = splitPoint

 //remove from the routing table all the nodes that do not belong to the new group
 for (each n ∈ routingTable)
   if ((oldGroupID == groupID AND n.nodeID  <= splitPoint) OR
       (oldGroupID != groupID AND n.nodeID > groupID))
          routingTable.remove(n)

 //update Chord
 if (nodeID is the biggest identifier in the local routing table)
          joinChord(groupID)

 //update the index stored at nodes
 for (each ind ∈ indexRepository)
   if ((oldGroupID == groupID AND ind  <= splitPoint) OR
       (oldGroupID != groupID AND ind > groupID))
          indexRepository.remove(ind)
```

Figure 5.7: The pseudo-code for the group splitting algorithm run at each node.

- **Adjusting the group prefix:** The node compares its node identifier with the splitting point. If the splitting point is its successor in the identifier space, the node changes its group identifier to the splitting point. Otherwise the node keeps its original group identifier.

- **Updating the local routing table:** The node updates its local routing table, by comparing each entry with the splitting point and its new group identifier. The nodes that are no longer in the group after splitting are removed. The cost of updating the local routing table is $O(r)$ where $r$ is the size of the original group.

- **Updating Chord:** If the node changes its group identifier it has to update Chord. Specifically, the nodes that have changed their group identifier have to (1) leave Chord using the old group identifier, and (2) join Chord again using the new group identifier.

To perform step (1), two approaches can be used. The first approach is to explicitly update Chord. Since a node knows about all other nodes in a group, this operation can be done once, by one of the nodes in the group (e.g., the leader), the update message including a list of all the nodes in the group. This update operation is equivalent to a node join and has a cost of $O(\log^2 N)$, where N is the number of groups in the system. The second approach is "passive" and does nothing leaving the maintenance algorithm to remove these nodes from the corresponding *finger lists*.

Step (2) has to be explicitly performed since the new group identifier does not exist in Chord. The same approach as node leave may be used, i.e., only one node in the group (e.g., the leader) performs the join. The cost is $O(\log^2 N)$. The overall cost of a splitting algorithm at the Chord level remains $O(\log^2 N)$, where N is the number of the groups in the system.

Note that while performing the Chord updates, some queries may end up at nodes with the original group identifier instead of the new one. The nodes with the original identifier may be able to answer these queries if the location of the data indices was not updated as yet. Else, the query has to be redirected to the correct node, which is close on the ring.

- **Updating the stored information:** The index is replicated at the nodes in a group. When a group splits, each node keeps only the data indices that map to its new group, and deletes the rest. This process may be delayed until Chord has been successfully updated. This allows outstanding queries that end up at the wrong node to be answered directly and not forwarded.

### Dissolving a Group

A group has to be dissolved when its size reaches the minimum allowed. Considering the way data indices are stored at nodes, they should be moved to the successor group. As a result, the nodes in the dissolving group join the successor group. Figure 5.8 illustrates the process of dissolving a group. The nodes in the group 153 join the successor group,

with identifier 232.

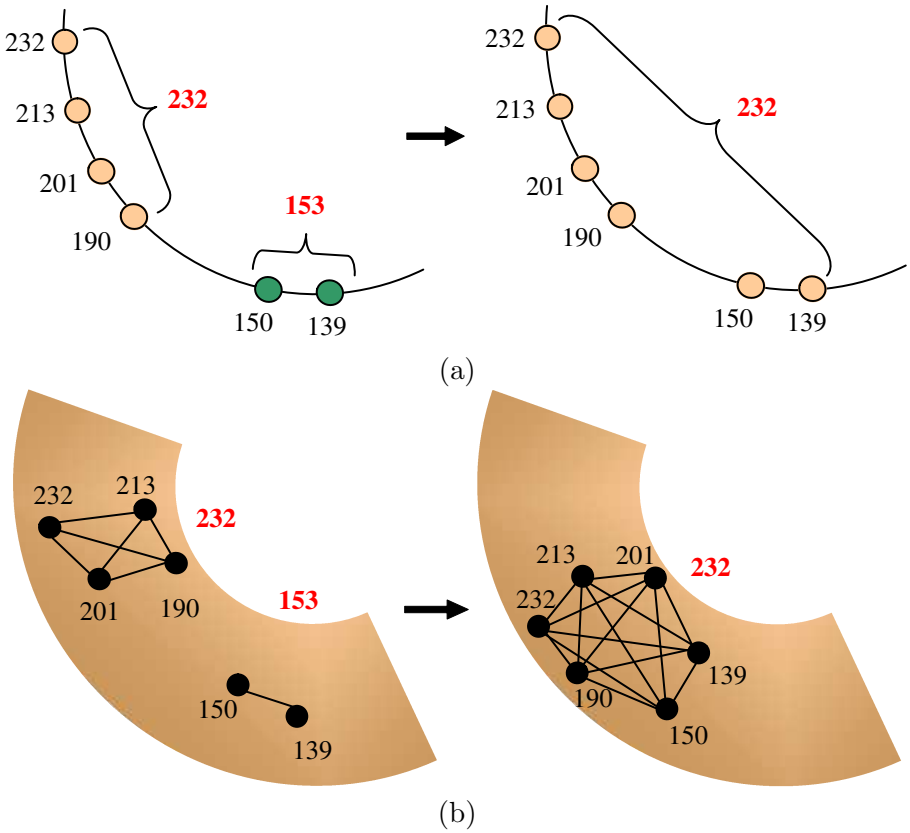



Figure 5.8: Dissolving group 153: (a) Chord view; (b) SquidTON view.

Before joining the successor group, the nodes in the group must agree to enter the DISSOLVE mode. Each node has a complete view of the group, and based on its current size, they should be able to decide independently whether to dissolve the group or not. However, new nodes joining the group may prevent the nodes in the group for reaching the same conclusion independently. To agree if the group needs to be dissolved or not, the group runs a leader-based consensus algorithm, similar to the one for group split.

While in DISSOLVE mode, each node rejects any *join* request, and runs the dissolving algorithm, which is a special case of the node join algorithm. The main difference is that the node already stores data indices, associated with its node identifier. Figure 5.9 presents the pseudo-code for the dissolving algorithm.

The dissolving of a group may trigger the splitting of its successor group. The splitting algorithm divides a group in two new groups of approximately the same size. After the splitting algorithm runs, the new formed groups are stable, and the load is

```
group_disolve()

  //change the group prefix to the successor's
  oldGroupID = groupID
  groupID = successors.getGroupID()

  //update the local routing table and notify the nodes in the successor group
  for (each n ∈ successors.getNode().routingTable)
     routingTable.add (n)
     n.routingTable.add (this)

  //update Chord
  updateChord(oldGroupID)

  //replicate index in the group
  replicateIndex()
```

Figure 5.9: The pseudocode for the group dissolving algorithm run at each node.

balanced.

Note that Chord, the overly network used by the current Squid prototype, does address fault tolerance. As mentioned above, a node in Chord periodically checks the validity of its successor, predecessor, and finger table entries and replaces them when they are found to be obsolete. Each node also keeps a list of $r$ consecutive successors instead of only one successor to facilitate the repair process. Lookups may be disrupted while the repair algorithm fixes obsolete finger table entries. During this time the lookup mechanism uses successor nodes to route causing the lookup to take more than O(log N) hops. In Squid, queries containing wildcards, partial keywords and ranges are resolved using multiple lookups. As a result, the efficiency of lookups is critical. SquidTON builds redundancy into the finger tables and reduces lookups using successors. It also provides a less expensive repair mechanism and speeds up query resolution.

## 5.3 Addressing Reliability at the Storage Layer. Index Replication

As noted in the introduction to this chapter, Squid is concerned only with index integrity. Data is manipulated at the application layer since there may be application-specific constraints (e.g., ownership issues). If the application looses parts of the data queries will return false positives. However, each data index has a lifetime associated with it and the index is deleted if it is not refreshed before this lifetime expires. In this

way obsolete data elements are garbage collected and false positives are minimized.

An index is replicated at multiple nodes to ensure index reliability. The replication is done at the group level and each node replicates its portion of the index at other nodes in the group. Since groups are small (e.g., at most 64 nodes) we chose to fully replicate the index. A data element published at one node in the group will be published at all other nodes within the group. At join, a node will copy the index from one of the nodes already in the group. A gossiping algorithm [17] is used to ensure that the indices stored at all the nodes are consistent. Each node periodically chooses a node from its local routing table, at random, and exchanges summaries of its index and its local routing table.

## 5.4 Load Balancing

SquidTON uses the load balancing algorithms presented in Section 3.5, with small modifications. The object of these load balancing algorithms is to distribute the storage load evenly among the nodes, i.e., to have each node store approximately the same amount of data indices. However, some nodes may store more popular data than others, and will receive a larger number of requests. Future work will consider this computational load and construct a load imbalance index for each node as function of both storage and computational loads. The rest of this section presents load balancing algorithms modified for SquidTON.

### 5.4.1 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. Groups of nodes that are logical successors of these identifiers respond and report their load. The new node uses the identifier that will place it in the most loaded group. Note that since we use complete replication of index data within a group, each node in the group stores all the index data in the group, i.e., a new node joining the group will not reduce the storage load per node. However, increasing the size of the group will eventually cause it to split in half and this will also

split the storage load of the group in half. If $i$ identifiers are generated during join, the cost of finding the best successor is O($i$ log N), i.e., the join cost remains the same. However, this step is not sufficient by itself. The runtime load-balancing algorithms presented below further improve load distribution.

### 5.4.2 Load Balancing at Runtime

Section 3.5 presented two runtime load balancing mechanisms. The first one can be used as is with only one restriction: the nodes that participate in this algorithm belong to the same group. This algorithm may require the participating nodes to change their node identifiers. However, as the participating nodes are in the same group the group identifier is not changed, and as a result Chord is not updated. Further, since the index is fully replicated, changes in the node identifier do not trigger index relocation. Only the local routing tables need to be updated. The cost of this algorithm is O($r$) where $r$ is the size of the group.

The second load balancing algorithm, using virtual nodes, is not used. Instead, the nodes migrate from one group to another. A group with many nodes but few data indices may send some of its nodes to a group that has fewer nodes and a large number of data indices. The nodes that migrate leave their group and join the new one. The cost of this algorithm is the same as the cost of a node join or leave, which is O($\log^2$N), where N is the number of groups in the system.

## 5.5 Experimental Evaluation

This section evaluates the performance of SquidTON and its ability to tolerate multiple node failures. SquidTON's performance depends on the level of redundancy built into the overlay and the level of index replication used. Several experiments have been performed by varying the size of the groups and the size of the finger lists.

As mentioned in Section 5.2.4, the size of a group is constrained by a maximum and a minimum value. The maximum size has to be chosen such that the cost of maintaining the group is not high. The minimum size has to be chosen such that the desired degree

of replication is achieved, both at the finger table level and at the stored information level. The size of the finger lists is also constrained by a maximum value, since the finger tables should be kept small. A small finger table is easier to maintain and faster to lookup, but might not provide enough redundancy. The following issues should be considered when tuning the system:

- **Intra-group communication:** A group uses a completely connected graph as its topology. Every operation in the group (e.g., routing table update, split, dissolve, etc.) involves all the nodes in the group. If the size of the group is $r$, each operation has a cost of $O(r)$. As a result, the smaller the size of a group, the less expensive it is to maintain it.

- **Storage space:** The index is replicated at every node in the group. Each node has limited storage and it is desirable to ensure that the indices stored at a node fit in its core memory.

- **Local routing table size:** The size of the routing table is also important. A smaller routing table is easier to maintain and leads to faster lookups. A small group results in smaller routing tables.

- **Finger table size:** The size of the finger table should be as small as possible. The redundancy built in the finger table in SquidTON increases its size by a factor equal with the maximum size of the finger lists. The finger lists should have the smallest size that enables the degree of fault tolerance desired.

- **Fault tolerance:** The system's ability to tolerate faults is direct proportional to the degree of redundancy built in the system. As a result, the larger the size of the group and the finger lists the more fault tolerant the system is.

To summarize, the group size and the finger list size have to be chosen to provide the desired degree of fault-tolerance while ensuring efficiency.

The experiments presented below were performed using a SquidTON simulator. The starting point for the SquidTON simulator was the Squid simulator described

in Chapter 3. The overlay in the simulator was modified to implement the two-tier structure and each node maintains two routing tables. The routing mechanism was also modified to enable a node to choose between multiple equivalent destinations at each routing step. The query engine did not change.

Two sets of experiments were performed. The first experiment measured the ability of SquidTON to cope with simultaneous node failures. The second experiment measured the effect of the two-tier overlay on the query engine. The experiments used a 3-dimensional keyword space and systems of sizes $10^3$, $5 \times 10^3$ and $10^4$ nodes. The data used were $4 \times 10^5$ HTML CiteSeer [72] files describing scientific articles. The nodes in the simulation stored indices (SFC numerical index, keywords, and a reference to the data), corresponding to the data elements and not the actual data. Only unique data elements were used and each data element was uniquely described by keywords and had a unique SFC index. The experiments were performed on a stable and load-balanced system.

### 5.5.1 Simultaneous Node Failures

This experiment evaluated the ability of SquidTON to operate when simultaneous faults occur. The system was tested for finger lists of sizes ($s$) 4, 8 and 16 nodes. The size of the groups was not fixed as it was not relevant to this experiment. However, groups did have a minimum size of at least $s$ nodes.

The experiment simulated the simultaneous failure of randomly chosen $p$% of the nodes. A set of lookups for randomly chosen 1% of the data were performed after these nodes were failed. The same set of lookups was performed in each experiment. The originating nodes for the lookups were chosen randomly. A lookup was considered to have failed if the finger list needed for any of its steps failed (e.g., all the nodes with references in the finger list failed). Each experiment was repeated 100 times and the results were averaged. The following metrics were measured:

- **GF**: Percentage of failed groups. A group is considered to have failed if all its representatives in SquidTon failed.

- **FLF**: Percentage of failed finger lists.
- **LF**: Percentage of failed lookups.
- **NFLF**: Percentage of nodes with failed finger lists.

| Failure rate | Finger list size | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 entries/finger | | | | 8 entries/finger | | | | 16 entries/finger | | | |
| | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF |
| **10%** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **20%** | 0* | 0.1 | 1.78 | 0.83 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **30%** | 0.3 | 0.44 | 5.16 | 8.42 | 0* | 0* | 1.5 | 0* | 0 | 0 | 0 | 0 |
| **40%** | 2.2 | 2.08 | 18.98 | 19 | 0* | 0* | 1.9 | 0.65 | 0 | 0 | 0 | 0 |
| **50%** | 5.92 | 5.92 | 32.8 | 47.04 | 0.3 | 0.4 | 14.45 | 3.05 | 0 | 0 | 0 | 0 |

(a)

| Failure rate | Finger list size | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 entries/finger | | | | 8 entries/finger | | | | 16 entries/finger | | | |
| | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF |
| **10%** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **20%** | 0* | 0* | 2.05 | 2.25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **30%** | 0.28 | 0.26 | 6.82 | 8.48 | 0* | 0* | 0* | 0* | 0 | 0 | 0 | 0 |
| **40%** | 2 | 2.04 | 21.66 | 26.04 | 0* | 0* | 5.4 | 0.4 | 0* | 0* | 3.4 | 0.05 |
| **50%** | 5.91 | 5.72 | 42.29 | 52.4 | 0* | 0* | 6.94 | 3.38 | 0* | 0* | 5.2 | 0.12 |

(b)

| Failure rate | Finger list size | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 entries/finger | | | | 8 entries/finger | | | | 16 entries/finger | | | |
| | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF | GF | FLF | LF | NFLF |
| **10%** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **20%** | 0* | 0* | 4 | 2.5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **30%** | 0* | 0* | 13.26 | 8.52 | 0* | 0* | 0* | 0* | 0 | 0 | 0 | 0 |
| **40%** | 2.1 | 2.12 | 27.99 | 28.65 | 0* | 0* | 0* | 0.7 | 0 | 0 | 0 | 0 |
| **50%** | 5.77 | 5.84 | 49.06 | 58.57 | 0* | 0* | 2.36 | 3.72 | 0* | 0* | 1.35 | 0* |

(c)

Table 5.1: Effects of simultaneous node failures: (a) system with $10^3$ nodes; (b) system with $5 \times 10^3$ nodes; (c) system with $10^4$ nodes. 0* denotes a value very close to 0.

Tables 5.1 (a), (b) and (c) present the results obtained for systems with $10^3$, $5 \times 10^3$ and $10^4$ nodes respectively. The results obtained for each system size are similar indicating that the size of the system does not matter when choosing the size of the finger lists. The results vary with the size of the finger lists. As the size of the finger lists increase, the system performs better and can tolerate a larger fraction of failed nodes.

For finger lists with 4 entries the system can tolerate the simultaneous failure of 10% of the nodes. For finger lists with 8 entries the system tolerates the simultaneous failure of 20% of the nodes. Further, in this case, if 30% of the nodes fail the system is minimally affected. A system with 16 entries per finger list can tolerate the simultaneous failure of almost 40% of the nodes. However, the size of the finger lists cannot be increased too much without affecting the cost of maintaining the finger tables and the overall performance of the system. The maximum size of a finger list should be chosen such that it can handle the expected simultaneous node failures in the system. For example, if the system is expected to experience at most 20% simultaneous failures, the maximum size of a finger list can be set to between 8 and 16, independent of the system size.

### 5.5.2 Query Engine Performance

This experiment evaluated the performance of the query engine within SquidTON. In this experiment, group sizes of 1 (regular Squid), 4, 8, 16 and 32 nodes were used. The set of queries from Chapter 4 was used and the number of nodes contacted by each query was measured. The results were averaged and normalized against the measurements for group size 1, i.e., against Squid. The results are plotted in Figure 5.10.

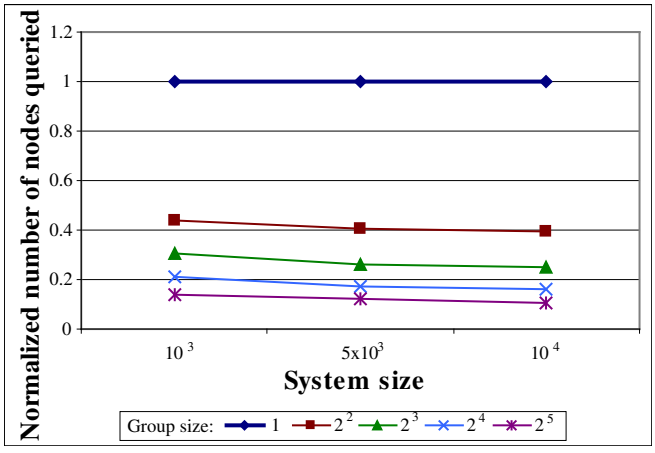



Figure 5.10: Query engine performance within SquidTON.

In the figure, the line at y = 1 represents the nodes queried by Squid (i.e., group size = 1). The other lines represent the nodes queried by SquidTON with 4, 8, 16 and respectively 32 nodes per group. As expected, the number of nodes queried decreases

as the group size increases, for the same system size. This is because that data is fully replicated within each group and only one node from the group needs to be contacted. Note that from the query routing point of view, SquidTON with larger groups is equivalent to a Squid overlay with fewer nodes, i.e., "total nodes/group size" nodes.

Note that the queries used in this experiment have coverage of 1%, 0.1% and 0.01% of total data in the system. These queries define many clusters, and query many nodes. Queries with small coverage typically contact only a few nodes, and increasing the size of the groups may not improve the performance of these queries.

The results presented in Figure 5.10 suggest that, in order to make query processing more efficient, it is enough to increase the size of the groups. Note that the results were obtained using full replication within the group. Partial replication schemes typically lead to more than one node in a group being contacted to solve a query and result in a smaller or possibly no query improvement. For example, if data is replicated randomly within a group, the query has to be broadcast/flooded in the group. Full replication is expensive in large groups, both in terms of storage and in terms of communication costs. To be efficient, the groups should be kept as small as possible.

In conclusion, the minimum size for a group should be at least as large as the maximum size of the finger lists. The maximum size for a group is system dependent. The expected quantity of data stored per node, the storage capacity of the nodes, their processing power, and the latencies of the links need to be considered when choosing the maximum group size.

# Chapter 6

# Applications

DHT-based P2P systems offer attractive solutions for a large set of applications, ranging from data sharing to rendezvous-based semantic messaging for pervasive applications. This chapter presents a short survey of existing applications built on top of DHT-based systems, and two applications designed using Squid: (1) a data discovery infrastructure for Tissue Microarray (TMA) data [59] and (2) Meteor [31], a content-based middleware for decoupled interactions in pervasive environments. The second application was implemented and tested, and the first is currently under development. Note that Squid has also been used for Grid resource discovery [56], for Web Service discovery [57] and to build Rudder, a decentralized coordination middleware [37].

## 6.1 Related Work

Several applications have been built over DHT-based P2P systems. Storage systems, an early and significant class of DHT-based applications, include PAST [18], OceanStore [33], CFS [16], Ivy [42] and UsenetDHT [61]. Information discovery systems include systems for resource discovery in Grid environments [1, 56], Web Services discovery systems [55, 57] and data sharing systems [59].

DHT-based P2P systems have been also used as a rendezvous infrastructure and content routing in systems such as i3 [62] built over Chord, Place Lab [10] built over a third party DHT, and Meteor [31] built over Squid.

Content distribution is another application area where DHT-based P2P systems have been successfully used. Scribe [52], an early topic-based publish subscribe system built on top of Pastry [51] uses DHTs to construct multicast and anycast trees. SDIMS [67] also uses trees within the DHT topology to perform aggregation. Coral [24] is a content distribution network built on a *distributed sloppy hash table*, based on Kademlia [39], offering weaker consistencies than traditional DHTs.

## 6.2 A Peer-to-Peer Collaboratory for Tissue Microarray Research

This project is joint work with Dr. David Foran, and his research laboratory at the University of Medicine and Dentistry of New Jersey (UMDNJ).

### 6.2.1 Overview

The tissue microarray (TMA) technique enables researchers to extract small cylinders of tissue from histological sections and arrange them in a matrix configuration on a recipient paraffin block such that hundreds can be analyzed simultaneously. A key advantage of TMA technology is that it allows amplification of limited tissue resources by providing the means for producing large numbers of small core biopsies, rather than a single section. Using conventional protocols, a study composed of 300 tissue samples would involve processing of 300 slides, i.e., at least 20 batches of 15 slides. Using TMA the entire cohort can be processed on a single slide. As a result, TMA technology holds great potential for reducing the time and cost associated with conducting research in tissue banking, proteomics, and outcome studies. However capturing, organizing, analyzing and characterizing, and sharing TMA data presents a number of significant challenges.

A significant challenge is the large volume of TMA data. Today, TMAs can contain from tens to hundreds of samples (0.6 to 2mm in diameter) arranged on a single slide. A digitized TMA specimen containing just 400 discs can easily approach 18GB in size. Given the increasing number of institutions and investigators utilizing TMA technology it is likely that modern facilities may easily generate tens of thousands of entries and terabytes of data. Clearly archiving, indexing and cataloging and mining this data across the TMA research community is a significant challenge and centralized solutions quickly become infeasible.

Finally, the increasing popularity of TMA has lead to more and more medical and research institutions being interested and conducting research in this area. While the exact focus of the research conducted by each of these groups may differ in terms of the patient group, the type of cancer, and/or the nature of the staining, being able to share

data and meta-data has many advantages. Sharing experimental results and clinical outcomes data could lead to huge benefits in drug discovery and therapy planning. Some leading institutions are developing data management systems for TMA data. However, these systems are only minimally useful if the data isn't accessible to others in the scientific community. Also, there are ongoing efforts focused on developing standards to represent TMA data (e.g., [3]). These existing efforts on sharing microarray data are based on centralized databases. However, the size of the data involved as well as issues of ownership can limit the scalability and feasibility of these approaches.

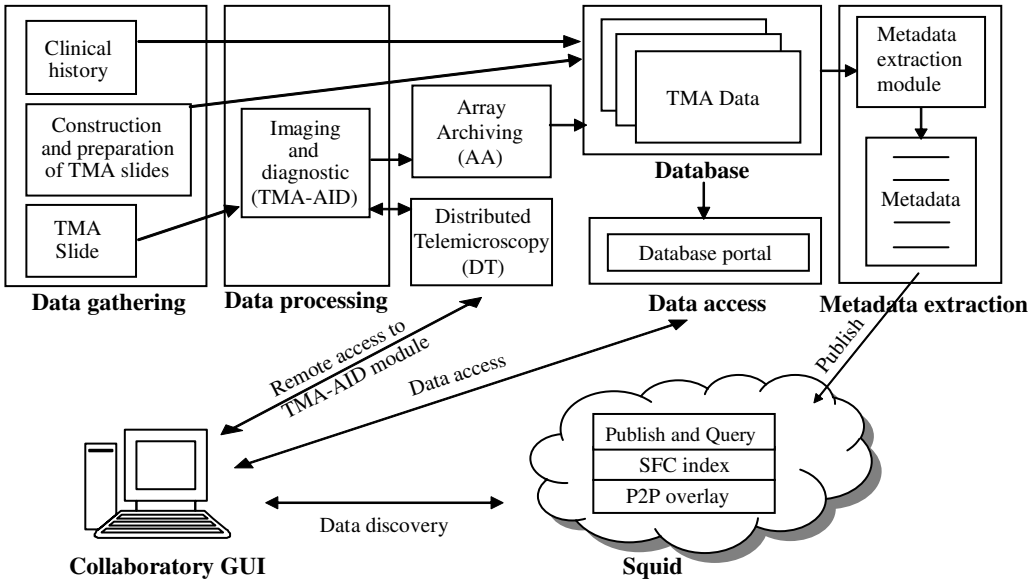



Figure 6.1: A schematic overview of the peer-to-peer TMA collaboratory.

We have designed a prototype peer-to-peer collaboratory for imaging, analyzing, and seamlessly sharing tissue microarrays (TMA), correlated clinical data and experimental results across a consortium of distributed clinical and research sites. A schematic overview of the collaboratory is presented in Figure 6.1. The key components are described below.

- The *data gathering module* collects the data that is to be processed and shared. There are three major sources of data: TMA slides, which are constructed as illustrated in Figure 6.2, clinical history of the donors, and information related with the construction and preparation of TMA slides.

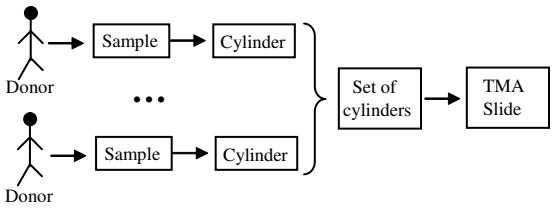



Figure 6.2: TMA slide preparation.

- The *data processing module* uses imaging and analyzing algorithms to extract relevant data from a TMA slide (i.e., the TMA-AID system). The data gathered is stored in the database using the Array Archiving (AA) subsystem. The TMA-AID system can be accessed remotely using the Distributed Telemicroscopy (DT) subsystem. The *data access module* enables remote access to the data stored in the local database.

- The *metadata extraction module* extracts metadata describing the shared data from the local database. The metadata is published in the *Squid* P2P storage and discovery system. Finally the *collaboratory GUI* allows users to flexibly search TMA data and metadata in Squid and access it through the Database Portal.

- The Squid module, a key component of the collaboratory, indexes the TMA metadata, and provides a way to search the metadata using flexible queries. Each peer (e.g., research institution) in this system maintains ownership of its data and only publishes (in a controlled manner) metadata describing its data, which can then be discovered and searched externally. Note that access to TMA data in this system is always controlled by the owner of the data.

### 6.2.2 Integrating Squid in the TMA Collaboratory

#### The P2P Infrastructure

The participating peers in the P2P infrastructure typically run on machines at hospitals, research centers and universities. Specialized software agents at each local site extract metadata from the local database, and publish it in the Squid P2P storage and discovery system. However, rather than storing the data, only references to the data described

by the metadata are stored. This behavior is desired because access to data is typically restricted based on access credentials. The Squid P2P infrastructure thus enables global discovery (with appropriate access control restrictions) of metadata while allowing peers to maintain ownership and locally control access to their data.

Since peers typically run on dedicated machines, the machines will be likely to be robust and stay alive for longer periods of time, and the P2P system can become quite stable. While this property is not necessary for Squid, it can be exploited to reduce the maintenance costs of the overlay network.

### Metadata Extraction

A part of the data available locally is indexed using metadata, and will be used in queries. Certain image files that don't have very suggestive metadata associated with them are not used. The metadata associated with a piece of data is extracted from the local database by a software agent, which then publishes it to Squid along with references to the data. The agent checks the database for changes at regular intervals, looking for new data. The process is illustrated in Figure 6.3.

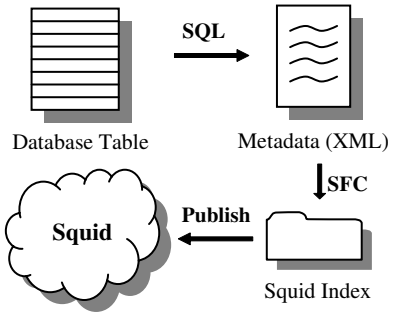



Figure 6.3: The process of publishing data in Squid: each piece of data has associated an XML file with metadata. The metadata is used to publish the XML and the location of the data in Squid.

An example of the XML metadata extracted from a database record (a case) is presented in Figure 6.4. The values of the attributes are used as keywords. Squid constructs the index using the keywords. The index is then used to locate the peer node where the XML metadata will be stored, together with the address of the database containing more information about this case (e.g., images, case history, etc).

```
<tma_entry>
    <CaseOrigin>New Jersey</CaseOrigin>
    <Age>61</Age>
    <CancerType>breast cancer</CancerType>
    <Marker>Her2</Marker>
    <DrugRegimen>Arimidex</DrugRegimen>
    <ResponseToTreatment>high</ResponseToTreatment>
<tma_entry>
```

Figure 6.4: Example of metadata (XML) extracted from the database.

### Searching for Data

The system is queried through a friendly graphical user interface (GUI). The user query is presented to Squid as an XML file. Squid parses the document, extracts the user query and resolves it. The results are presented to the user and consist of links to relevant data in databases maintained by hospitals, research centers, etc. The user can then contact the owners of the data to obtain required permissions to access the data using the database portal. Note that the access to the data is outside Squid and is subject to the hospital's (or research center's) regulations. Figure 6.5 illustrates this process.

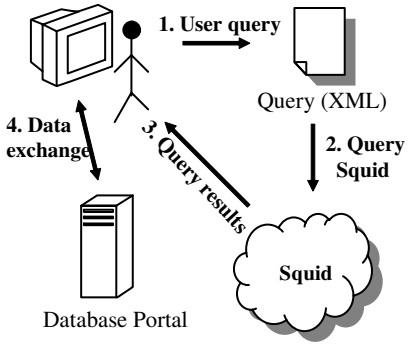



Figure 6.5: Searching information using Squid.

```
<tma_query>
    <CaseOrigin>*</CaseOrigin>
    <Age>30-40</Age>
    <CancerType>breast cancer</CancerType>
    <Marker>*</Marker>
    <DrugRegimen>chemo</DrugRegimen>
    <ResponseToTreatment>high</ResponseToTreatment>
<tma_query>
```

Figure 6.6: Example of a user query.

An example of a possible user query is presented in Figure 6.6. In this example, the user is interested in data about patients between ages 30 and 40, with breast cancer, who are treated with chemo, and have a high response to the treatment. The marker type and the case origin are wildcards.

The TMA collaboratory presented above is an example of how large collections of scientific data can be shared between research groups. Note that Squid can also be used as a discovery mechanism for other types of scientific data.

## 6.3 Meteor - Content-based Middleware for Decoupled Interactions in Pervasive Environments

This project was designed and developed at TASSL, in collaboration with Vincent Matossian and Nanyan Jiang.

### 6.3.1 Overview

The growing ubiquity of sensor/actuator devices with embedded computing and communications capabilities [19], and the emergence of pervasive information and computational Grids, lead to the birth of a new generation of applications based on seamless access, aggregation and interactions. Low-power sensor/actuator devices will permit remote monitoring of buildings, streets, hospitals, factories, etc. For example, one can conceive a fire management application where computational models use streaming information from sensors embedded in the building along with realtime and predicted weather information (temperature, wind speed and direction, humidity) and archived history data to predict the spread of the fire and to guide firefighters, warning of potential threats (blowback if a door is opened) and indicating most effective options. This information can also be used to control actuators in the building to manage the fire and reduce damage. Other examples include scientific and engineering applications that combine computations, experiments, observations, and realtime data to manage and optimize its objectives (e.g., oil production, weather prediction).

These applications are large, distributed, heterogeneous and dynamic. They require

a middleware infrastructure that: (a) is scalable and self-managing, (b) is based on content rather than names and/or addresses, (c) supports asynchronous and decoupled interactions rather than forcing synchronizations, and (d) provides some interaction guarantees.

We designed Meteor, a layered middleware that meet all the requirements stated above. Squid is used by Meteor as a content-based routing layer. Querying process can be viewed as consistent-based routing; the nodes defined by the queries are the desired destinations and are specified by content rather than addresses. Squid is scalable, does not require synchronization, and offers guarantees.

The theoretical base for content-based decoupled interactions for pervasive applications is the *Associative Rendezvous (AR)* paradigm. AR extends the conventional name/identifier-based rendezvous [20, 62] in two ways. First it uses flexible combinations of keywords (i.e., keywords, partial keywords, wildcards, ranges) from a semantic information space, instead of opaque identifiers that have to be globally synchronized. Second, it enables the reactive behaviors at rendezvous points to be embedded in the message or message request. AR differs from emerging publish/subscribe paradigms in that individual interests (subscriptions) are not used for routing and do not have to be synchronized - they can be locally modified at a rendezvous node at anytime.

Metor is a content-based middleware infrastructure for decoupled interactions in pervasive environments based on the Associative Rendezvous model. It is essentially a dynamic P2P network of Rendezvous Peers (RP). To use Meteor, applications must have access to at least one RP so they can post messages to this RP. A schematic overview of the Meteor stack is presented in Figure 6.7. It consists of 3 key components: (1) a self-organizing overlay (Chord), (2) a content-based routing infrastructure (based on Squid), and (3) the Associative Rendezvous Messaging Substrate (ARMS).

The rest of this section presents the AR paradigm, the content routing infrastructure, and summarizes ARMS.

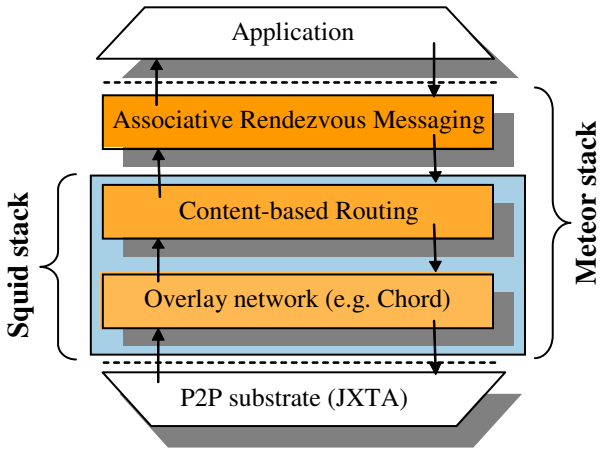



Figure 6.7: A schematic overview of the Meteor stack

### 6.3.2 Associative Rendezvous

Associative Rendezvous (AR) is a paradigm for content-based decoupled interactions with programmable reactive behaviors. Rendezvous-based interactions [20] provide a mechanism for decoupling senders and receivers. Senders send messages to a rendezvous point without knowledge of who or where the receivers are. Similarly, receivers receive messages from a rendezvous point without knowledge of who or where the senders are. Note that senders and receivers may be decoupled in both space and time [20]. Such decoupled asynchronous interactions are naturally suited for large, distributed and highly dynamic systems such as pervasive Grid environments.

AR extends the conventional name/identifier-based rendezvous in two ways. First, it uses flexible combinations of keywords (i.e., keywords, partial keywords, wildcards and ranges) from a semantic information space, instead of opaque identifiers (names, addresses) that have to be globally known. Interactions are based on content described by keywords, such as the type of data a sensor produces (temperature or humidity) and/or its location, the type of functionality a service provides and/or its QoS guarantees, and the capability and/or the cost of a resource. Second, it enables reactive behaviors at the rendezvous points to be encapsulated within messages, increasing flexibility and expressivity and enabling multiple interaction semantics (e.g., broadcast, multicast, notification, publisher/subscriber, mobility, etc.).

The AR interaction model consists of three elements: *Messages*, *Associative Selection*, and *Reactive Behaviors*.

### AR Messages

An AR message is defined as the triplet: (header, action, data). The data field may be empty or may contain the message payload. The header includes a *semantic profile* in addition to the credentials of the sender, a message context and the TTL (time-to live) of the message. The *profile* is a set of attributes and/or attribute-value pairs, and identifies the recipients of the message. The attribute fields must be keywords from a defined information space while the values field may be a keyword, partial keyword, wildcard, or range from the same space. At the rendezvous point, a *profile* is classified as a *data profile* or an *interest profile* depending on the *action* field of the message. The *action* field of the AR message defines the reactive behavior at the rendezvous point (i.e., the action to be taken when the message reaches the RP). Examples of a *data profile* and an *interest profile* are shown in Figure 6.8. The data is produced by a sensor, and it is sent to be stored at a RP. An interest profile (Figure 6.8 (b)) may also be stored in the system as a persistent query and its owner is notified when matching data profiles are found.

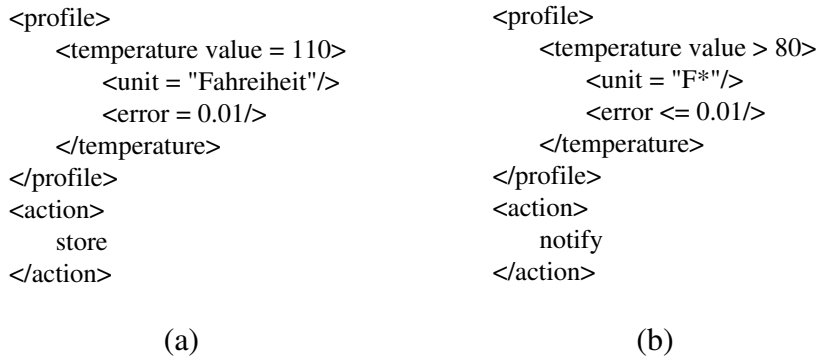



Figure 6.8: Example of semantic profiles: (a) data profile; (b) interest profile.

The AR interaction model defines a single symmetric *post* primitive. To send a message, the sender composes a message by appropriately defining the header, action and data fields, and invokes the post primitive. The post primitive reaches the Squid

layer at the source rendezvous point, which resolves the profile (which is the same as resolving a query) and delivers the message to relevant rendezvous points. The profile resolution guarantees that all the rendezvous points that match the profile will be identified. However, the actual delivery relies on existing transport protocols. A receive operation is similar except that the action field is defined appropriately and the data field is empty. Figure 6.9 presents examples of *post* messages. Figure 6.9 (a) shows the message sent by a sensor that reads temperature and wind. Figure 6.9 (b) shows the message posted by a firefighter application, requesting data from temperature and wind sensors.

```
<message id = "1211121">
 <header>
    <credentials sensorID = "s2"/>
    <expiration date ="mm/dd/yy"/>
    <profile>
        <temperature value = 110
            <unit = "Fahrenheit"/>
            <error = 0.01>
        </temperature>
        <wind>
            <speed value=30>
                <unit ="mph"/>
                <error = 0.1/>
            </speed>
            <direction value = "NW"/>
        </wind>
    </profile>
 </header>
 <action>
      store
 </action>
 <payload>
      ....
 </payload>
</message>
```

(a)

```
<message id = "32321121">
 <header>
    <credentials fireFighterID = "f22" />
    <expiration date ="mm/dd/yy"/>
    <profile>
        <temperature value <= 80>
            <unit = "F*"/>
            <error <= 0.01/>
        </temperature>
        <wind>
            <speed value >= 25>
                <unit = "mph"/>
                <error = "*"/>
            </speed>
            <direction value = "S"/>
        </wind>
    </profile>
 </header>
 <action>
   retrieve
 </action>
 <payload>
    ....
 </payload>
</message>
```

(b)

Figure 6.9: Example of *post* messages: (a) sending data; (b) retrieving data.

### Associative Selection

The user profiles are converted into a format that can be efficiently stored and evaluated by a filter engine [6]. Any information that is persistent in the system should be stored

and indexed in order to make scalable and efficient matching possible. In our system, every new arrival profile will be matched to the profiles stored in the system.

### Reactive Behavior

The action field of the message defines the reactive behavior at the rendezvous point. Basic reactive behaviors currently defined include *store, retrieve, notify*, and *delete*. The *notify* and *delete* actions are explicitly invoked on a data or an interest profile. The *store* action stores data and data profile at the rendezvous point. It also causes the message profile to be matched against existing interest profiles and associated actions to be executed in case of a positive match. The *retrieve* action retrieves data corresponding to each matching data profile. The *notify* action matches the message profile against existing interest/data profile, and notifies the sender if there is at least one positive match. Finally, the *delete* action deletes all matching interest/data profiles.

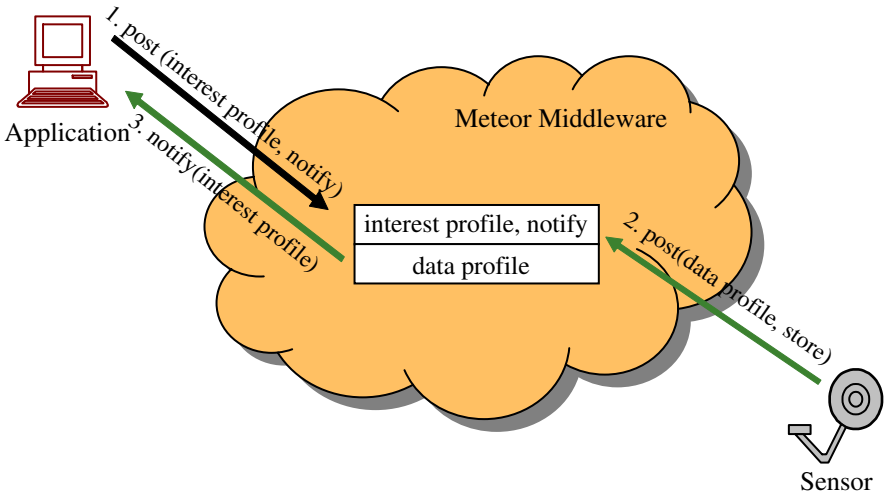



Figure 6.10: Example of interactions using Associative Rendezvous.

Figure 6.10 shows a simple interaction scenario using Associative Rendezvous. The application posts its interest profile in Meteor. The interest profile acts as a persistent (until its expiration date) query, being stored at one or multiple RPs. The action associated with this interest profile is *notify*, indicating that the application expects notifications when data matching its interest profile is published. The sensor periodically publishes data. The data will be stored at a rendezvous point, and if it matches

the interest profiles stored, notification messages are sent to the owners of the matched interest profiles. After receiving a notification the application can retrieve the data.

### 6.3.3 Squid - Content-based Routing Infrastructure

Meteor uses Squid as a content-based routing infrastructure. Squid provides a simple primitive to the upper layer; *route*. This primitive comes in two flavors: route to a single destination, and route to multiple destinations. The destination is specified semantically using keywords, partial keywords, wildcards and ranges.

Keywords can be common words or values of globally defined attributes, depending on the nature of the application that uses Squid. These keywords form a multidimensional keyword space; keyword tuples represent points in this space and the keywords are the coordinates. A *keyword tuple* is defined as a list of $d$ keywords, wildcards and/or ranges, where $d$ is the dimensionality of the keyword space. For example (110F, 30mph) and (110F, *) are valid keyword tuples in a 2-dimensional space. A keyword tuple containing ranges can be (90-120F, *, 30mph - *) and specifies a query for data regarding temperature, humidity and wind, produced by sensors. If the keyword tuple contains only complete keywords, it is called *simple*, and if it contains partial keywords, wildcards and/or ranges it is called *complex*.

A *simple keyword tuple* specifies a single destination and it corresponds to Squid's *publish* primitive. A *complex keyword tuple* specifies multiple destinations and it corresponds to Squid's *query* primitive.

#### Routing Using Simple Keyword Tuples: Unicast

The routing process for a simple keyword tuple corresponds to Squid's *publih* operation, and is illustrated in Figure 6.11. It consists of two steps: first, the SFC-mapping is used to construct the index of the destination RP node from the simple keyword tuple, and then the overlay network lookup mechanism is used to route to the appropriate RP node in the overlay.

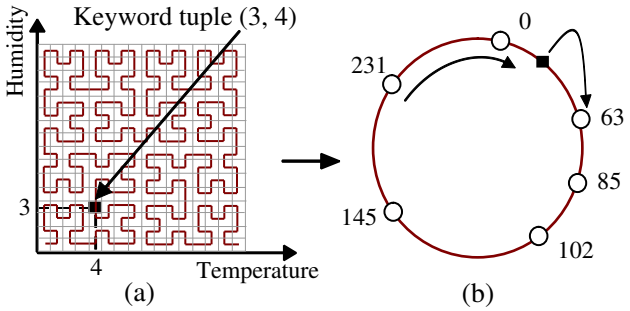



Figure 6.11: Routing using a simple keyword tuple in Squid: (a) the simple keyword tuple (4, 3) is mapped to index 31, using Hilbert SFC; (b) the message will be routed to the successor of 31, RP node 63. The overlay network has 6 RP nodes, and an identifier space from 0 to $2^8$-1.

### Routing Using Complex Keyword Tuples: Multicast

A complex keyword tuple identifies a region in the keyword space, which in turn corresponds to clusters of points in the index space. For example, in Figure 6.12 (a), the complex keyword tuple (4-7, 3-12) representing data read by a sensor (temperature between 4 and 7 units and humidity between 3 and 12 units) identifies 3 clusters.

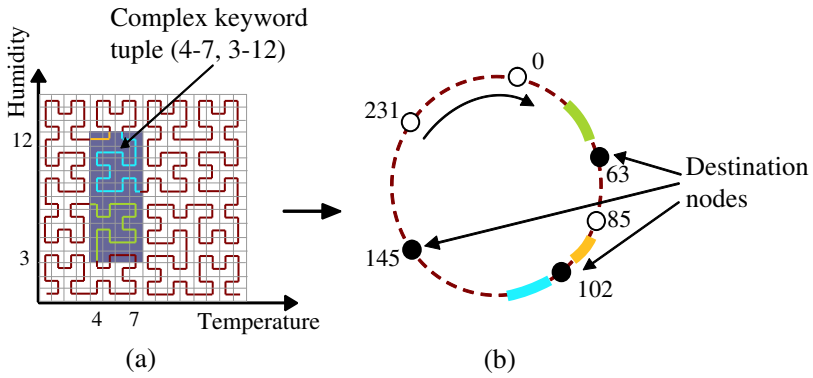



Figure 6.12: Routing using complex keyword tuple (4-7, 3-12): (a) the keyword tuple defines a rectangular region in the 2-dimensional keyword space consisting of 3 clusters (3 segments on the SFC curve); (b) the clusters (the solid part of the circle) correspond to destination RP nodes 63, 102 and 145.

A complex keyword tuple is mapped by the Hilbert SFC to clusters of SFC indices and correspondingly, multiple rendezvous point identifiers. Once the clusters associated with the complex keyword tuple are identified, the query engine described in Chapter 3 distributes the message to the destination rendezvous points.

### 6.3.4 Messaging Layer - ARMS

The ARMS layer implements the Associative Rendezvous interaction model. At each RP, ARMS consists of two components: the profile manager and the matching engine. The profile manager manages locally stored profiles. Profiles are implemented as XML files. The matching engine component is essentially responsible for matching profiles. An incoming message profiles is matched against existing interest and/or data profiles depending on the desired reactive behavior. If the result of the match is positive, then the action field of the incoming message is executed first and then the action field of the matched profile is evaluated.

### 6.3.5 Implementation

The current implementation of Meteor builds on Project JXTA [77]. The overlay network, Squid and the ARMS layers of the Meteor stack are implemented as event-driven JXTA services. Each layer registers itself as a listener for specific messages, and gets notified when a corresponding event is raised.

Meteor was evaluated on a Linux cluster consisting of 64 1.6 GHz Pentium IV machines and an 100 Mbps Ethernet interconnect. Each of the 64 nodes acted as a RP and ran the complete Meteor stack. Profiles at each RP were locally stored in a MySQL database. The overheads at each layer of the stack were measured. Section 3.6.2 presents the results for the overlay layer, and routing infrastructure. The complete results can be found in [31].

# Chapter 7

# Conclusions and Future Work

## 7.1 Summary

Information discovery in large distributed systems in the absence of global knowledge of naming conventions is a very important and challenging problem. Existing solutions are either limited in the complexity of the query (e.g., ranges, wildcards) that they can support, or in their scope (e.g., limited-scope flooding). This thesis presented a fundamentally new approach for indexing and searching DHT-based systems, one that can support complex searches using keywords, partial keywords, wildcards and ranges, while guaranteeing that all data elements in the system matching the query are found with bounded costs in terms of the number of messages and nodes involved in the query. The design, implementation and evaluation of Squid, a P2P information discovery system based on this approach, were presented.

A key component of the approach presented in the thesis 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 Space Filling Curves, and ensures that indices that are close together in the 1-dimensional index space come from keywords that are lexicographically close in the multi-dimensional keyword space. Furthermore, the hierarchical and 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 designed and developed to improve system stability and performance.

The behavior of the locality preserving indexing mechanism and the performance of the query engine was analyzed and was demonstrated experimentally. The analysis shows that in large systems, for a generic query matching $p$% of the total data in the system, the number of nodes with matching data approaches $p$% of all nodes in

the system. It also shows that by optimizing the search process using recursive query refinement and pruning, the number of nodes involved in the querying process is close to the number of nodes storing data matching the query.

To addresses the reliability and fault tolerance of the system, the SquidTON overlay was designed. SquidTON, a two-tier overlay, builds redundancy into the routing tables, and replicates the indices. The nodes form small overlay networks called groups, which are connected by a Chord overlay network. The index stored at nodes is replicated within each group.

Squid has been used as a discovery mechanism in computational Grids [56], for Web Service discovery [57], and to share TMA data across scientific communities [59]. Systems such as Meteor [31] and Rudder [37] use Squid as a content routing mechanism; Meteor is a content-based middleware for decoupled interactions in pervasive environments, and Rudder is a decentralized coordination middleware.

## 7.2 Conclusions and Contributions

Content-based information discovery and sharing is becoming an important requirement of many emerging distributed applications. The ability to support complex queries and to offer guarantees are desired features for these applications. Squid addresses these challenges in a decentralized manner and demonstrates that it is feasible and advantageous to use structured P2P systems to search for information. It uses a locality-preserving indexing scheme and a query engine optimized for the indexing scheme to enable search in structured, DHT-based P2P systems.

The theoretical analysis, simulations, and experimental results confirm that Squid scales well with the number of nodes, the indexing scheme preserves data locality, the query engine is efficient, and the system answers most queries correctly even when a large number of nodes fail. Squid has also been integrated in several applications, proving that it is a valuable component for decentralized large-scale distributed applications.

Key contributions of this research are summarized below.

### 7.2.1 Locality Preserving Indexing Scheme

Squid's SFC-based indexing scheme can naturally support ranges, wildcards and partial keywords. Finally, it also preserves locality. The only constraint is in the number of keywords that a data element can use for indexing (ideally at most five keywords). We are not aware of any other indexing mechanism that better preserves locality for large number of keywords. To support larger numbers of keywords, several approaches are proposed in Section 3.4.2.

### 7.2.2 Distributed Query Engine

The query engine uses the properties of the indexing scheme to optimize the querying process. Each query is divided in sub-queries which are resolved in parallel at nodes with matches for that subquery. The experiments show that the query engine works well, and scales to large systems.

The theoretical analysis presented shows that, if a query matches $p$% of the data stored in the system, the number of nodes that store matches approaches $p$% of the total nodes in the system, as the system grows. This result is also verified experimentally.

### 7.2.3 Load Balancing

The indexing scheme does not distribute the data uniformly and as a result Squid uses load balancing mechanisms to distribute the data evenly across the nodes. The cost of load balancing depends on the cost of maintaining the overlay, and the cost of moving parts of the index between nodes. Squid uses two main load balancing algorithms. The first is inexpensive since it is integrated into the node "join" mechanism. The second is a runtime load balancing algorithm that further improves load balance. These load balancing mechanisms are evaluated and their effectiveness is demonstrated.

### 7.2.4 Fault Tolerance

The ability to tolerate the failure of nodes is essential in structured P2P systems. The reliability of the system depends on its ability to function correctly in the presence of

failure. SquidTON, a two-tier overlay, builds redundancy into the routing tables, and replicates the index. As the experiments show, SquidTON can tolerate a large number of failures, and at the same time speed up query processing.

### 7.2.5 Applications

Squid was integrated in several applications, proving that it is a valuable component for large-scale decentralized distributed applications. Future application domains to be investigated include content distribution mechanisms for publish-subscribe applications.

## 7.3 Future Work

Future work includes core issues as well as application specific issues. The core issues are: (1) using caching to address the system's availability, (2) optimizing the querying process using "hints", (3) addressing keyword space dimensionality, (4) designing custom SFCs, (5) building awareness of geographical locality into SquidTON, and (6) exploiting node heterogeneity to further improve the reliabiltiy of the system. The application specific issues include (1) ranking and (2) designing new applications over Squid. The rest of this section discusses these issues.

- **Availability of the system using caching:** It is possible that different users issue the same query in a short period of time. Further, it is highly probable that the result of the query will be the same in both cases. This is because, if the interval between queries is short, it is unlikely that the index will be updated with information matching the issued query. As a result, it makes sense to cache the results of a query for a short period of time. Such a caching mechanism can improve response time as well as the availability of the system.

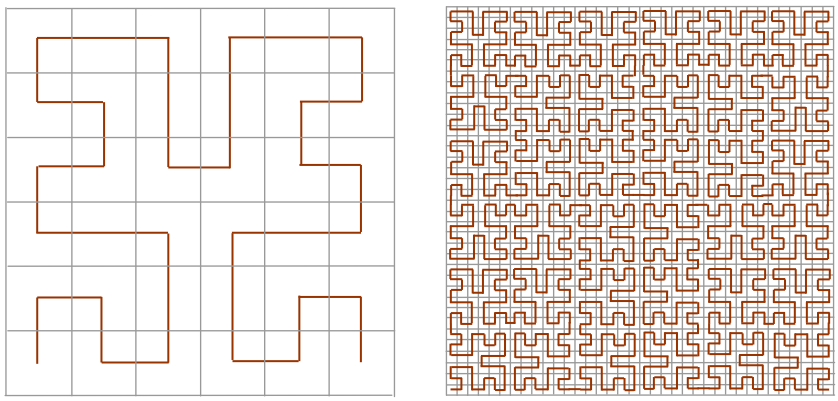
- **Query engine optimization using "hints":** The results presented in Chapter 3 show that not all queried nodes have data that matches a query. Further optimizations can be defined to query only the nodes that store data elements that match the query. Future work includes the use of "hints" to implement such an optimization. A hint at a node is a condensed summary about the content of

its neighbors. Hints are constructed when data is inserted into the system, and kept up-to-date using a gossiping algorithm [17].

- **Keyword space dimensionality:** As noted in Section 3.4.2, the locality preserving property of SFC begins to deteriorate for information spaces of dimensionality greater than five [41]. The dimensionality of the keyword space dictates the maximum number of keywords a data element can use for indexing. This means that a data element can be indexed using at most five keywords. Some application may want to use more than five keywords to describe a data element. This thesis addresses this problem partially as described in Section 3.4.2. Further investigation is necessary.

- **Custom SFCs:** As presented in Section 3.1.2, it is possible that a portion of the multidimensional keyword space is not used. Custom SFCs can be used, so that the unused portion is minimized. The challenge is to find appropriate SFC curves so that the locality is preserved. If the type of queries that will be issued is known, the curve can be chosen such that it will preserve locality for that type of queries. If nothing is known about the queries, the curve should preserve locality along all axes. The best candidate in this case is a Hilbert-like curve. Custom SFC curves need to be designed and their clustering properties evaluated. For example, the base-6 curve presented in Figure 7.1 can be used to index keywords containing English characters and digits. The keywords in this case are base-36 numbers. As Figure 7.1 (b) shows, the second approximation of this custom SFC can map 36 digits. As a result it can be used as a base recursion step when mapping base-36 numbers.

- **Building awareness of geographical locality into SquidTON:** SquidTON is a two-tier overlay where nodes are connected in small groups and the groups are connected using Chord. Having a group consists of nodes that are geographically close together can reduce delays and the amount of communication between nodes in a group.



(a) (b)

Figure 7.1: Modified base-6 Hilbert curve: (a) first approximation; (b) second approximation.

- **Exploiting node heterogeneity:** To further improve the reliability of the system, the nodes joining Chord should be relatively stable and powerful. SquidTON treats nodes equally and any node in a group may join Chord. However, if the nodes that join Chord are stable and powerful in terms of computation power and connectivity, the system as a whole will be more stable.

- **Ranking:** The number of matches for a query can be very large. Consequently, a ranking algorithm can be used to list the results based on their relevancy to the query. Some ranking mechanisms, such as the one introduced by Google [27], use the references (links) between documents to compute their "rank". VSM and LSI [4] mechanisms are also used to rank text documents based on content. However, if the data elements are metadata describing scientific data, the algorithms mentioned above are not suited and a more appropriate algorithm for this kind of data is one that incorporates the reputation of the data source.

  Squid can be used by various applications, each handling its own type of data. Consequently, no single ranking algorithm can be used for all these different data collections. It is therefore more appropriate to deploy the ranking algorithm at the application level, where it can be better customized to the particular data type. The applications built over Squid will be enhanced with appropriate ranking mechanisms.

- **New applications that use Squid:** Chapter 6 present two applications designed using Squid, and mentions other applications that use Squid. However, new application domains will be investigated. For example, Squid can be used to build multicast trees for content distribution in publish-subscribe systems.

# References


[1] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing (P2P2002)*, pages 33–40, Sweden, September 2002.

[2] J. Aspnes and G. Shan. Skip Graphs. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, Baltimore, MD, January 2003.

[3] J.J. Berman, M.E. Edgerton, and B.A. Friedman. The Tissue Microarray Data Exchange Specification: A Community-based, Open Source Tool for Sharing Tissue Microarray Data. *BMC Medical Information and Decision Making*, 3(1):5, May 2003.

[4] M. Berry, Z. Drmac, and E. Jessup. Matrices, Vector Spaces and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999.

[5] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of SIGCOMM 2004*, pages 353–366, Portland, OR, 2004.

[6] T. Bially. *A Class of Dimension Changing Mapping and its Application to Bandwidth Compression.* PhD thesis, Polytechnic Institute of Brooklyn, June 1967.

[7] E. Bugnion, T. Roos, R. Wattenhofer, and P. Widmayer. Space Filling Curves versus Random Walks. *LNCS: Proceedings of Algorithmic Foundations of Geographic Information Systems*, 1340:199–211, 1997.

[8] A. R. Butz. Alternative Algorithm for Hilberts Space-Filling Curve. *IEEE Trans. Computers*, pages 424–426, April 1971.

[9] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 80–87, Berkeley, CA, February 2003.

[10] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A Case Study in Building Layered DHT Applications. Technical report, Intel Research Berkeley, 2005.

[11] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A Prototype Implementation of Archival Intermemory. In *In Proceedings of the 4th ACM Conference on Digital Libraries*, pages 28–37, Berkeley, CA, August 1999.

[12] 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*, pages 311–320, Berkeley, California, June 2000.

[13] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proceedings of ACM SIGCOMM*, pages 177–190, October 2002.

[14] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-Ring: An Index Structure for Peer-to-Peer Systems. Technical Report TR2004-1946, Cornell University, 2004.

[15] 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)*, pages 23–32, Vienna, Austria, July 2002.

[16] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide Cooperative Storage with CFS. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Banff, Canada, October 2001.

[17] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *In Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, Canada, August 1987.

[18] P. Druschel and A. Rowstron. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *In Proceedings of The 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 75–80, Schoss Elmau, Germany, May 2001.

[19] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99)*, pages 263–270, Seattle, WA, August 1999.

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[21] C. Faloutsos and S. Roseman. Fractals for Secondary Key Retrieval. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252, 1989.

[22] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128, Berkeley, CA, February 2003.

[23] I. Foster and C. Kesselman. *The GRID - Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[24] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing Content Publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 239–252, San Francisco, CA, 2004.

[25] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule them All: Multidimensional Queries in P2P Systems. In *Proceedings of Seventh International Workshop on the Web and Databases (WebDB 2004)*, pages 19–24, Paris, France, 2004.

[26] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, 2003.

[27] T. Haveliwala. Efficient Computation of Pagerank. Technical Report 1999-31, Standford Digital Library Technologies Project, 1999.

[28] D. Hilbert. Ueber stetige Abbildung einer Linie auf ein Flachenstuck. *Mathematische Annalen*, 38:459–460, 1891.

[29] A. Iamnitchi, I. Foster, and D. Nurmi. A Peer-to-Peer Approach to Resource Discovery in Grid Environments. Technical Report TR-2002-06, University of Chicago, 2002.

[30] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 332–342, Atlantic City, New Jersey, 1990.

[31] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar. Content-based Middleware for Decoupled Interactions in Pervasive Envionments. Technical Report 252, Wireless Information Network Laboratory (WINLAB), Rutgers University, April 2004.

[32] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing (STOC)*, pages 654–663, El Paso, Texas, May 1997.

[33] J. Kubiatowicz. OceanStore: An Architecture for Global-Scalable Persistent Storage. In *Proceedings of the ASPLOS 2000*, pages 190–201, Cambridge, MA, 2000.

[34] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[35] A. Lempel and J. Ziv. Compression of Two-Dimensional Data. *IEEE Transactions On Information Theory*, 32(1):2–8, 1986.

[36] W. Li, Z. Xu, F. Dong, and J. Zhang. Grid Resource Discovery Based on a Routing-Transferring Model. In *Proceedings of the 3rd International Workshop on Grid Computing*, pages 145–156, Baltimore, MD, 2002.

[37] Z. Li and M. Parashar. Rudder: A Rule-based Multi-Agent Infrastructure for Supporting Autonomic Grid Applications. In *Proceedings of the International Conference on Autonomic Computing*, pages 278–279, New York, NY, 2004.

[38] 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*, pages 84–95, New York, NY, June 2002.

[39] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System based on the XOR Metric. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, Cambridge, MA, 2002.

[40] D. A. Menasce. Scalable P2P Search. *IEEE Internet Computing*, 7(2):83–87, March 2003.

[41] 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):124–141, 2001.

[42] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Yvy: A Read/Write Peer to Peer File System. In *Proceedings of Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, 2002.

[43] A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly Press, 2001.

[44] M. Parashar and J. C. Browne. On Partitioning Dynamic Adaptive Grid Hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 604–613, Maui, Hawaii, 1996.

[45] G. Peano. Sur une Courbe qui Remplit Toute une Aire Plane. *Mathematishe Annalen*, 36:157–160, 1980.

[46] C. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the ACM SPAA*, pages 311–320, Newport, Rhode Island, June 1997.

[47] B. D. Goodman R. Subramanian, editor. *Peer to Peer Computing, The Evolution of a Disruptive Technology*. Idea Group Publishing, 2004.

[48] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, 2001.

[49] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, volume 2672, pages 21–40, Rio de Janeiro, Brazil, June 2003. Springer-Verlag.

[50] R. L. Rivest. Partial Match Retrieval Algorithms. *SIAM J. Comput.*, 5(1):19–50, 1976.

[51] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Largescale Peer-to-Peer Systems. In *In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[52] A. Rowstron, A.M. Kermarrec, M. Castro, and P. Druschel. Scribe: The Design of a Large-Scale Event Notification Infrastructure. In *Networked Group Communication, Lecture Notes in Computer Science*, volume 2233, pages 30–43, 2001.

[53] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[54] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, USA, 2002.

[55] 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, September 2002.

[56] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proceedings of the 12th High Performance Distributed Computing (HPDC)*, pages 226–235, Seattle, WA, June 2003.

[57] C. Schmidt and M. Parashar. A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web Journal*, 7(2):211–229, 2004.

[58] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *Internet Computing Journal*, 8(3):19–26, 2004.

[59] C. Schmidt, M. Parashar, W. Chen, and D. Foran. Engineering a Peer-to-Peer Collaboratory for Tissue Microarray Research. In *Proceedings of CLADE Workshop, at HPDC 13th*, pages 64–73, Honolulu, HI, June 2004.

[60] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.

[61] E. Sit, F. Dabek, and J. Robertson. UsenetDHT: A Low Overhead Usenet Server. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS'04)*, pages 206–216, San Diego, CA, 2004.

[62] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proceedings of ACM SIGCOMM*, pages 73–86, Pittsburgh, PA, 2002.

[63] 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*, pages 149–160, San Diego, CA, August 2001.

[64] R. Subramanian and B. D. Goodman. Peer-to-Peer Corporate Resource Sharing and Distribution with Mesh. *E-Collaborations and Virtual Organizations*, 2004.

[65] C. Tang, Z. Xu, and M. Mahalingam. PeerSearch: Efficient Information Retrieval in Peer-to-Peer Networks. Technical Report HPL-2002-198, HP Labs, 2002.

[66] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, Tamper-Evident, Censorship-Resistant Web Publishing System. In *In Proceedings of the 9th USENIX Security Symposium*, August 2000.

[67] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of ACM SIGCOMM*, pages 379–390, Portland, OR, 2004.

[68] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *In Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, Roma, Italy, September 2001.

[69] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Systems. In *In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 5–14, Vienna, Austria, July 2002.

[70] 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, April 2001.

[71] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 11–20, Port Jefferson, NY, June 2001.

[72] CiteSeer webpage: http://citeseer.ist.psu.edu/.

[73] Gnutella webpage: http://gnutella.wego.com/.

[74] Groove webpage: http://www.grove.net/.

[75] ICQ webpage: http://web.icq.com/.

[76] Jabber webpage: http://www.jabber.org/.

[77] JXTA webpage: http://www.jxta.org/.

[78] KazaA webpage: http://www.kazaa.com/.

[79] Morpheus webpage: http://www.morpheus-os.com/.

[80] Napster webpage: http://napster.com/.

[81] SETI@home webpage: http://setiathome.ssl.berkeley.edu/.

[82] Universal description, discovery and integration: Uddi technical white paper, September 2000.

## APPENDIX A

*Statement:*

Assume that the system is load balanced, i.e., each node stores approximately the same quantity of data. If an SFC segment $S$ indexes $a\%$ of the data, then the number of nodes that store the segment $S$ is approximately $a\%$ of the total number of nodes in the system (assuming that the total number of nodes is greater than *100/a*).

*Proof:*

Let $n$ be the number of nodes in the system. Let $t$ be the number of data elements stored in the system. As the system is load balanced, there is approximately the same number of data elements per node. This means that there are $\frac{t}{n}$ data elements per node.

Segment $S$ stores $a\%$ of the data $\Rightarrow$ $S$ contains $\frac{a}{100} \times t$ data elements. The number of nodes storing the segment $S$ is:

$$\frac{a}{100} \times t \times \frac{n}{t} = \frac{a}{100} \times n$$

$= a\%$ of the total nodes.

# Curriculum Vitae

## Cristina Simona Schmidt

**2005** Ph.D. Candidate in Computer Engineering, GPA 4.0/4.0; Rutgers University, NJ, USA.

**1999** MS in Computer Science, GPA 10.00/10.00; "Babeş-Bolyai" University, Cluj-Napoca, Romania

**1998** BS in Computer Science, GPA 9.78/10.00; "Babeş-Bolyai" University, Cluj-Napoca, Romania

**2000-2005** Graduate Assistant, Center for Advance Information Processing, Rutgers University, NJ, USA

**1998-2000** Software Developer, SC Transart SRL, Cluj-Napoca, Romania

**1998-2000** Teaching Assistant, "Babeş-Bolyai" University, Cluj-Napoca, Romania

### Publications

C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. In *Proceedings of the 12$^t$h High Performance Distributed Computing (HPDC)*, Seattle, WA, USA, pages 226-235, June 2003.

M. Agarwal, V. Bhat, Z. Li, H. Liu, B. Khargharia, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri and M. Parashar. AutoMate: Enabling Autonomic Applications on the Grid. In *Proceedings of the Autonomic Computing Workshop, 5th Annual International Active Middleware Services Workshop (AMS2003)*, June 2003.

C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *Internet Computing Journal*, 8(3):19-26, May/June 2004.

C. Schmidt and M. Parashar. A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web Journal*, 7(2):211-229, June 2004.

C. Schmidt, M. Parashar, W. Chen and D. Foran. Engineering A Peer-to-Peer Collaboratory for Tissue Microarray Research. In *Proceedings of CLADE Workshop, at HPDC 13th*, pages 64-73, June 2004.

N. Jiang, C. Schmidt, V. Matossian and M. Parashar. Content-based Middleware for Decoupled Interactions in Pervasive Envionments. *Technical Report Number-252*, Wireless Information Network Laboratory (WINLAB), Rutgers University, April 2004.

C. Schmidt and M. Parashar. Peer to Peer Information Storage and Discovery Systems. *Book chapter, "Peer-to-peer computing: The evolution of a disruptive technology"*. Editors: R. Subramaniam and B.D. Goodman, Idea Group Publishing, 2004.

C. Schmidt and M. Parashar. Analyzing the Search Characteristics of Space Filling Curve-based Indexing within the Squid P2P Data Discovery System. *Technical Report TR-276*, Center for Advanced Information Processing (CAIP), Rutgers University, December 2004.

M. Parashar, Z. Li, H. Liu, V. Matossian and C. Schmidt. Enabling Autonomic Grid Applications: Requirements, Models and Infrastructures. *"Self-Star Properties in Complex Information Systems", Lecture Notes in Computer Science*, Springer Verlag. Editors: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, Vol. 3460, 2005.