Efficient and Flexible Search in Large Scale Distributed Systems

by

Reaz Ahmed

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2007

Authors Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Peer-to-peer (P2P) technology has triggered a wide range of distributed systems beyond simple file-sharing. Distributed XML databases, distributed computing, server-less web publishing and networked resource/service sharing are only a few to name. Despite of the diversity in applications, these systems share a common problem regarding searching and discovery of information. This commonality stems from the transitory nodes population and volatile information content in the participating nodes. In such dynamic environment, users are not expected to have the exact information about the available objects in the system. Rather queries are based on partial information, which requires the search mechanism to be flexible. On the other hand, to scale with network size the search mechanism is required to be bandwidth efficient.

Since the advent of P2P technology experts from industry and academia have proposed a number of search techniques - none of which is able to provide satisfactory solution to the conflicting requirements of search efficiency and flexibility. Structured search techniques, mostly Distributed Hash Table (DHT)-based, are bandwidth efficient while semi(un)-structured techniques are flexible. But, neither achieves both ends.

This thesis defines the Distributed Pattern Matching (DPM) problem. The DPM problem is to discover a pattern (*i.e.*, bit-vector) using any subset of its 1-bits, under the assumption that the patterns are distributed across a large population of networked nodes. Search problem in many distributed systems can be reduced to the DPM problem.

This thesis also presents two distinct search mechanisms, named Distributed Pattern Matching System (DPMS) and Plexus, for solving the DPM problem. DPMS is a semi-structured, hierarchical architecture aiming to discover a predefined number of matches by visiting a small number of nodes. Plexus, on the other hand, is a structured search mechanism based on the theory of Error Correcting Code (ECC). The design goal behind Plexus is to discover all the matches by visiting a reasonable number of nodes.

Acknowledgements

First and foremost I thank Allah, the Exalted in Might and the Wisest, for bringing me into existence and enabling me in completing this work.

I express my warmest gratitude to my supervisor, Professor Raouf Boutaba, for supporting me with a continuous stream of intellectual, moral and financial assistance. He has always encouraged me to pursue my own ideas and provided invaluable feedback to improve the quality of my research. I would like to extend my gratitude and acknowledgements to Professor Burkhard Stiller, Professor Sagar Naik, Professor Alex Lopez-Ortiz and Professor Srinivasan Keshav for their insightful comments and constructive criticisms of the work.

I am truly grateful to my parents for constantly inspiring me in the quest of knowledge and for securing a paved way for my career plans. I owe my deepest gratitude to my beloved wife Shapla and my son Shahriar for their patience and support during the time of hardship. Without the support and security from my family members, it would be impossible for me to complete this work.

I thank all my teachers, especially Professor M. Kaykobad, for their inspiration, wisdom and endless efforts. I am thankful to my colleagues, Mustaq Ahmed and Tarique M. Islam, for the effective discussions and reviews.

I am indebt to my home institute, Bangladesh University of Engineering and Technology (BUET), for entrusting me with the opportunity to pursue this study. I thank the government of Canada for funding my research under the Canadian Commonwealth Scholarship and Fellowship Plan. Last but not the least, I thank everyone who has contributed to this thesis, directly or indirectly.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Efficient discovery of information based on partial knowledge is a challenging problem faced by many large scale distributed systems, including content sharing P2P networks, service discovery systems and distributed XML database systems. Due to the transitory nature of peer population and volatile information content in these peers, users are not expected to have the exact information about the available objects in the system. Rather queries are based on partial information, which requires the search mechanism to be flexible. On the other hand to scale with network size the search mechanism is also required to be bandwidth efficient.

Existing search mechanisms do not provide a satisfactory solution for achieving the conflicting goals of flexibility and efficiency. Unstructured search protocols (as adopted in Gnutella and FastTrack) provide search flexibility but exhibit poor performance characteristics. Structured search techniques (mostly Distributed Hash Table (DHT)-based), on the other hand, can efficiently route queries but support exact-match semantic only.

The focus of this thesis is to devise efficient search techniques without sacrificing the flexibility requirement. We provide a generic framework called Distributed Pattern Matching (DPM) to address the search problem in large-scale, distributed environments. The DPM framework provides a means for facilitating search flexibility. We also present

two different search mechanisms, for achieving the efficiency requirement within DPM framework.

The organization of the rest of this chapter is as follows. The formal definition of the DPM problem is presented in Section 1.1, while Section 1.2 explains the motivation behind this research work. The contributions of this thesis are listed in Section 1.3. Finally, the organization of the thesis is outlined in Section 1.4.

1.1 Problem Definition

The generic pattern matching problem and its variants have been extensively studied in the Computer Science literature. Given a *text* (or raw data) $\mathbb{P}$ and a *pattern* $\mathbb{Q}$, the generic problem of pattern matching is to locate (all) the *occurrence*s of $\mathbb{Q}$ in $\mathbb{P}$. The definition of *text*, *pattern* and *occurrence* depends on the application domain. The *text* and *pattern* are two dimensional arrays in Image Processing applications, *string*s in Text Editing systems, trees in *tree pattern matching* [94], and arrays of sets in *subset matching* [46]. Variations in the definition of *occurrence* include exact matching, parameterized matching [26], approximate matching and matching with "don't cares" [21].

The variant of the pattern matching problem considered in this work is closely related to the *subset matching* problem [46]. In subset matching, pattern $\mathbb{Q} = \{Q_1, Q_2 \ldots Q_m\}$ and text $\mathbb{P} = \{P_1, P_2 \ldots P_n\}$ are collections of sets of characters drawn from some alphabet $\sum$. A pattern $\mathbb{Q}$ occurs at text position i if the set Q_j is a subset of the set P_{i+j-1}, for all $1 \leq j \leq m$.

We define the *Distributed Pattern Matching (DPM)* problem as a variant of the *subset matching* problem with the following restrictions:

- The pattern $\mathbb{Q}$ has a single element in its array, *i.e.*, $m = 1$ and $\mathbb{Q} = Q$.
- The n elements of $\mathbb{P}$ are distributed across a large number of networked nodes.

In many cases, bit-vectors of length $|\sum|$ are used to present texts (*i.e.*, P_i) and pattern (*i.e.*, Q), where a 1 (or 0) at the i^{th} bit of a bit vector represents the presence (or absence) of the i^{th} symbol in $\sum$. In the DPM formulation we assume that each element of $\mathbb{P}$ (*i.e.*, P_i) summarizes the identifying properties (*e.g.*, keywords, service description) of a shared object (*e.g.*, a file or a service). One possible form of such a pattern is a Bloom filter [30] obtained by hashing the properties associated with a shared object.

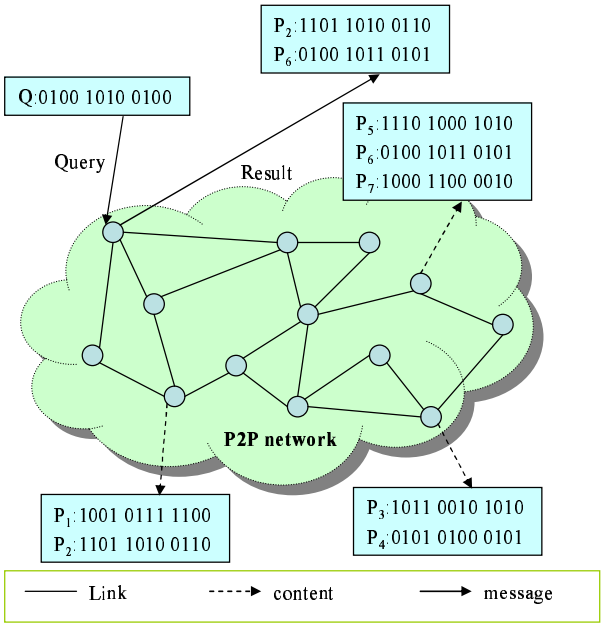


Figure 1.1: The Distributed Pattern Matching (DPM) problem. ($|\sum| = 12$)

Figure 1.1 presents a pictorial view of the DPM problem. Hereafter, we will use advertised pattern or simply advertisement to refer to a P_i and search pattern or simply query to refer to Q. Shared objects ($P_1 - P_7$ in Figure 1.1) and queries (Q in Figure 1.1) are represented as bit vectors. If the query is exactly the same as the advertisement, then the problem can efficiently be solved using conventional DHT techniques. But, we are interested in inexact pattern matching, where the 1-bits of a search pattern can be any subset of an advertised pattern that it should match against. In other words, the

result of a search should contain all the advertised patterns (P_2 and P_6 in Figure 1.1) that constitute supersets of the search pattern (Q in Figure 1.1).

1.2 Motivation

Importance of the DPM problem

Availability and location of content on the Internet have become extremely dynamic with the advent of P2P technology. Efficient discovery of information, based on partial information, has become a challenging problem in such large-scale distributed environments characterized by a transient population of networked nodes with heterogeneous capabilities. Due to the dynamic nature of the content and the participating nodes, users seldom have the exact or complete knowledge of the advertised information in the system. Instead, queries are often based on partial knowledge about a target advertisement. This mandates an efficient search mechanism that is capable of resolving queries based on partial or incomplete information about the target advertisements, and can handle the dynamism in nodes' and content's availability.

Keyword search is one of the essential functionalities offered by any peer-to-peer file-sharing system. A Centralized filesystem, as present in any traditional operating system, permits sophisticated search operations involving wildcards and partial keywords. Enabling existing P2P file-sharing systems with efficient wildcard search capability will allow users to perform flexible and powerful search. Besides inexact keyword matching, many problems, such as partial service description matching for service discovery systems, semantic matching of schemata and data records for P2P database systems, and molecular fingerprint matching in a distributed environment, can be mapped to the DPM problem.

As depicted in Figure 1.2, an advertisement in a P2P content sharing system consists of a number of keywords describing the content being shared. For a file-sharing P2P

system, it is unusual for a user to know the exact name of an advertised file. Instead, queries are based on a subset of the (partial) keywords that may be present in the advertisement. Bloom filters can be used for encoding advertised and queried keywords in the following manner. An advertisement (or query) Bloom filter can be constructed using the trigrams extracted from the advertised (or queried) keywords. Subset relationship between advertised and queried trigrams will hold for advertisement and query Bloom filters. For example, in Figure 1.2 trigrams for the first query constitute a subset of the advertised trigrams; as a result query pattern Q_1 is a subset of the advertisement pattern P. On the other hand, trigrams from the second query do not correspond to any subset of the advertised trigrams, and with high probability Q_2 will not be a subset of P.

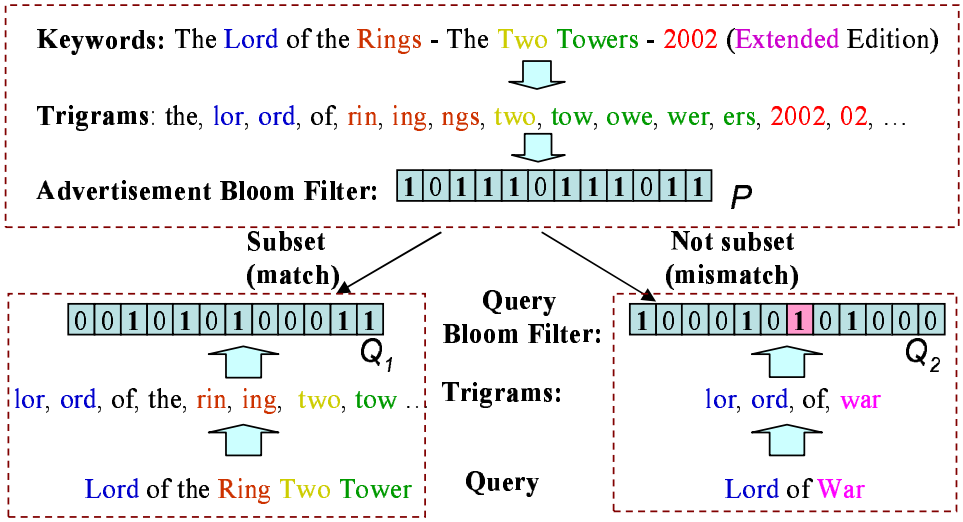


Figure 1.2: Partial keyword matching using DPM

For P2P database systems, as shown in Figure 1.3, XML documents are used as advertisements and XPath [90] is the most commonly used query language. Figure 1.3 presents an XPath query of the form $/a_1[b_1]/a_2[b_2]\ldots/a_n[b_n]$. Here, a_i is an element in an XML document, b_i is an XPath expression relative to element a_i. In this case, path prefixes from an XML document or the XPath expression (*e.g.*, $/a_1$, $/a_1/b_1$, $/a_1/a_2\ldots$) can be used as the *set elements* for Bloom filter construction.

For most service discovery systems a service description is advertised as a set of property-value pairs and a query for a service consists of a subset of the advertised property-value pairs. It is evident as shown in Figure 1.3 that an efficient solution to the DPM problem will enable us to generate satisfactory solutions to the search problem in these three important application domains.

Problems with Existing Search Techniques

State-of-the-art solutions for subset matching ([21], [46], [47]) in centralized environment, hold linear relationship with the number of patterns to be matched against. An equivalent solution in a distributed environment, where the patterns are distributed across networked nodes, will require the search message to be forwarded to all the nodes, *i.e.*, flooding the network with search messages. This solution will certainly not scale with network size.

Existing P2P search techniques are based on either non-structured hint-based routing or structured Distributed Hash Table (DHT)-based routing ([105], [118], [132]). Neither of these two paradigms can provide satisfactory solution to the DPM problem.

Over the last few years, DHT-based structured P2P systems ([132], [118], [105], [150], etc.) have gained importance due to their efficiency in query routing and completeness in search results. These techniques assign a portion of the key space to each peer and offer a single functionality: given a key, find the ID of the peer responsible for that key. Keys and IDs belong to the same space, usually 160-bit integers. A key is associated with a data item, and is computed by hashing the data item or some of its identifying properties. Most of these techniques take binary keys as input and apply prefix matching to route a query to a specific node in $O(\log N)$ hops, where N is the number of peers in the overlay. Despite their efficiency in query routing, these systems are not suitable for solving the DPM problem. The basic idea behind DHT-techniques is to partition the key-space,

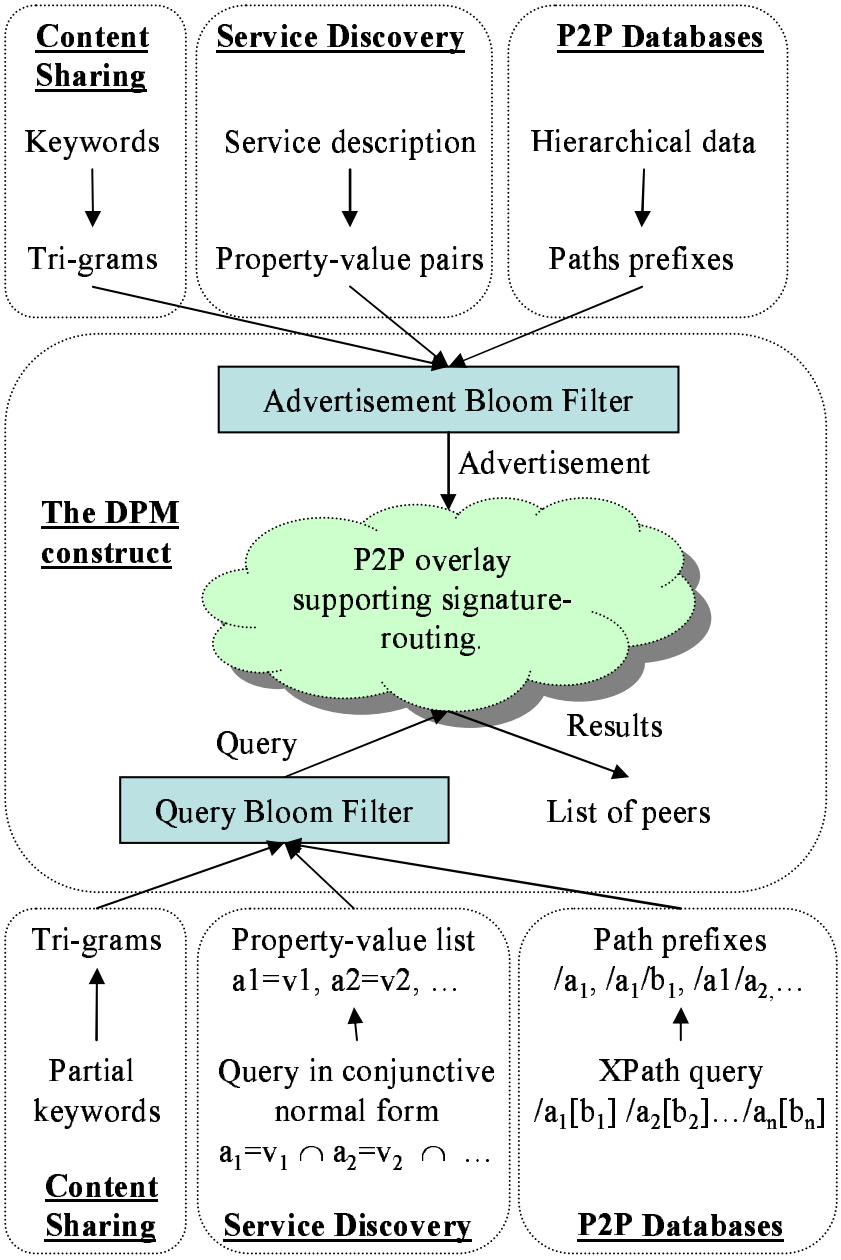


Figure 1.3: Mapping different problems to DPM framework

usually into non-overlapping regions, and to assign each region to a peer bearing an ID from that region. The partitioning of key-space is based on the *numeric distance* among the keys; whereas, for efficient pattern matching keys should be partitioned based on *Hamming distance*. The Hamming distance between patterns X and Y (of same length) is calculated as $d(X,Y) = |X \oplus Y| = no.\ of\ bits\ on\ which\ they\ disagree.$ As a result, these solutions generate several independent DHT-lookups, each followed by local flooding, for resolving a single query.

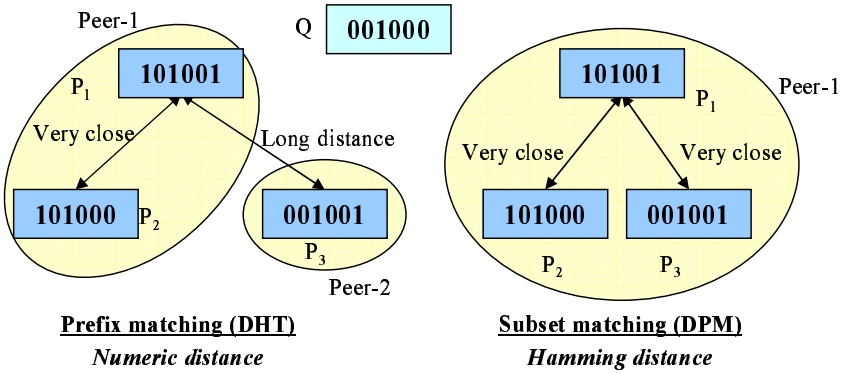


Figure 1.4: Problem with numeric distance based routing

For example, consider patterns $P_1 = 101001$, $P_2 = 101000$ and $P_3 = 001001$, and a query $Q = 001000$ matching all of these patterns (see Figure 1.4). Using numeric distance P_3 is far apart from P_1 and P_2. Thus a DHT-scheme will assign P_3 to a peer that is far apart (in overlay hops) from the peer responsible for P_1 or P_2 . This will require two independent DHT lookups to resolve the query. On the other hand, pair-wise hamming distance between P_1, P_2 and P_3 is only 1 and they will be stored on the same peer with high probability, if hamming distance based clustering is used.

Unstructured search techniques, such as Flooding [4] and Random walk [103], can be used to solve the DPM problem, but the generated search traffic is proportional to the number of peers in the overlay; moreover, there is no guarantee on search completeness. Hence adopting a unstructured search technique is not a good choice for solving the DPM

problem. Semi-structured search techniques [99][44] can also be used to solve the DPM problem. Compared to unstructured techniques, semi-structured techniques generate lower volume of search traffic. Yet, the cost of join and leave is higher in semi-structured techniques than unstructured techniques and no guarantee on search completeness is provided.

1.3 Contributions

This thesis focuses on the search problem in large scale distributed systems characterized by two major properties: transient population of networked nodes and heterogeneity in nodes' capabilities. Flexibility in query expressiveness, efficiency in generated search traffic, completeness in search results and resilience to node failures are the most important characteristics of a search mechanism in such environments. The goal of this thesis is to devise search mechanisms that can satisfy all of these requirements simultaneously, which is not delivered by any of the existing search mechanisms. The major contributions of this thesis are outlined below.

The Distributed Pattern Matching (DPM) Construct

Instead of working with a specific search problem, like keyword search or service discovery, we attack the problem from a generic point of view. We abstract the search problem in different application domains into a generic framework or problem formulation, called DPM [14]. Among the four requirements of a search mechanism, flexibility deals with query language semantics and the rest, *i.e.*, efficiency, completeness, and resilience, relate to system performance and thus are implementation-specific. The DPM construct encapsulates the flexibility requirement, and any solution to the DPM problem should focus on the performance specific requirements.

To our knowledge, this is the first time the DPM problem is identified. We believe that

this concept will aid in solving a number of other problems pertaining to P2P networking research, including P2P databases, P2P semantic search and P2P information retrieval.

The Survey and The Taxonomy

Large scale distributed systems, including P2P file-sharing, service discovery and P2P databases, share a common problem regarding search and discovery of information. This commonality stems from transitory peer population and volatile information content in the networked nodes. Different search techniques proposed for these domains contain many similarities. However there exists no survey consolidating these independent research activities from these important application domains. In this thesis we have presented a survey of existing search techniques in all these domains, which we believe will help in better understanding of the problem.

In addition, we have identified the main components in existing search mechanisms and classified them based on the characteristics of these components. The taxonomy presented in this work is simple and generic, and encompasses the majority of search techniques in large scale distributed systems.

Distributed Pattern Matching System (DPMS)

This thesis presents DPMS [16] (Chapter 3), for efficiently solving the DPM problem. DPMS uses replication and aggregation for distributing patterns advertised by the peers across the P2P overlay. In DPMS, peers collaborate to form a lattice-like indexing hierarchy. This hierarchy is used to efficiently route queries to target peer(s).

DPMS has several properties of an unstructured P2P system. First, it supports flexible queries involving partial and multiple keywords. Second, placement of content or index is not controlled by the system. Third, it can exploit the heterogeneity in peer capabilities. Unreliable and less capable peers contribute to less important parts

of the indexing hierarchy, while reliable (long lived) and powerful (in terms of storage and connection speed) peers take responsibility of more important parts of the indexing hierarchy.

To avoid flooding and to facilitate efficient query routing, DPMS uses the indexing hierarchy. The philosophy behind this architectural choice can be justified as follows. The highest level of an indexing hierarchy of height $O(\log \frac{N}{\log N})$ has $O(\log N)$ peers, where N is the total number of peers in the system. Peers at any level collectively cover all the leaf peers (hence all the advertised patterns) residing at the bottom level of the indexing hierarchy. In other words, we can check all the patterns at the bottom level by probing only $O(\log N)$ peers at level $O(\log \frac{N}{\log N})$. This implies that we can find κ leaf peers, containing match for a given query pattern, in

$$O(\log N + \xi\kappa \log \frac{N}{\log N}) \tag{1.1}$$

probes. The $O(\log N)$ probes constitute the cost of flooding the topmost level, and $O(\xi\kappa \log \frac{N}{\log N})$ is the cost of reaching the κ matching leaf peers along the indexing hierarchy of height $O(\log \frac{N}{\log N})$. The term ξ depends on the amount of "false matches" introduced by the lossy aggregation scheme, described below.

In such a hierarchy the topmost level peers will receive high volume of queries and will become performance bottlenecks. Moreover, fault-tolerance characteristics of the system will be poor; failure of any peer along the indexing hierarchy will result into unreachable leaf peers. To overcome these problems, DPMS uses peer replication at each level of the indexing hierarchy. The number of replicated peer groups increases exponentially as we move up along the indexing hierarchy.

With such a replication strategy, the network and storage overhead for index maintenance will be high. To reduce the impact of replication overhead, advertised patterns are aggregated using a *don't care*-based lossy aggregation scheme at each level of the indexing hierarchy. The proposed aggregation scheme allows the incorporation of mul-

tiple advertised patterns or aggregates in a single aggregate, while making it possible to perform matching of a query pattern against the aggregates. This lossy aggregation scheme introduces chances of "false matches" during the query routing process, and is reflected by the term ξ in Equation (1.1). Simulation results indicate that aggregation reduces storage overhead by $45 - 60\%$, while securing query routing efficiency very close to the ideal case, *i.e.*, without aggregation.

Plexus

Plexus [15, 17] is one of the most important contributions of this thesis. It is a structured search technique based on the application of Error Correcting Codes (ECC) [76] for routing in overlay networks. A structured search mechanism for efficiently solving the DPM problem would require to partition the pattern space based on Hamming Distance. Coding theory literature investigates error correcting codes that essentially strive to partition the pattern space.

The novelty of Plexus lies in the use of Hamming distance based routing, in contrast to the numeric distance based routing adopted in traditional DHT-approaches. This property makes subset matching capability intrinsic to the underlying routing mechanism. Plexus provides an efficient mechanism for advertising a binary pattern, and discovering it by using any subset of its 1-bits. It has a *partially decentralized* architecture involving superpeers.

The routing strategy devised for Plexus is based on the Generator matrix of a linear binary code. The routing efficiency in Plexus is provable and holds linear relationship with the size of the code being used. The number of links maintained by each indexing peer scales logarithmically with the number of peers. There exists multiple routing paths between any two pair of peers. This allows improved fault resilience. The replication strategy adopted in Plexus helps in distributing network load and improves availability.

In this thesis we have implemented Plexus, using the Extended Golay code [65]. However the theoretical model presented in Chapter 4 can be used with other linear codes. As proof of concept, we have also developed a prototype implementation of the Plexus protocol for solving partial keyword search problem and have evaluated its performance. As demonstrated by the simulation results, for a network of about 20000 superpeers, Plexus needs to visit only $0.7\% \sim 1\%$ of the superpeers to resolve a query and can discover about $97\% \sim 99\%$ of the advertised patterns matching the query. For achieving this level of completeness, the query needs to contain only 33% of the trigrams from an advertised pattern that it should match against. Plexus delivers a high level of fault-resilience by using replication and redundant routing paths. Even with 50% failed superpeers, Plexus can attain a high level of search completeness (about $97\% \sim 99\%$) by visiting only $1.4\% \sim 2\%$ of the superpeers.

Comparative Evaluation

The DPM problem can be solved using existing structured and unstructured search techniques. However, none of these techniques can simultaneously fulfill all of the four requirements for a good search mechanism: namely efficiency, flexibility, completeness and robustness. To justify this claim we have compared DPMS and Plexus against representative search techniques from both structured and un-structured paradigms. We have compared against unstructured search techniques like *Flooding* and *Random walk*. We have used a simple inverted indexing technique on top of *Chord* routing as a representative of the structured paradigm. The experimental results prove that DPMS and Plexus exhibit better performance characteristics than other search techniques in solving the DPM problem. This comparative evaluation provides us with a better insight of the performance of different search techniques in different circumstances, *e.g.*, growth in network size, variation in query information content, levels of peer failure *etc.*

1.4 Thesis Organization

The focus of this thesis is on the design and evaluation of flexible and efficient search techniques as a solution to the DPM problem. Applicability of these techniques in P2P keyword search has also been demonstrated. The rest of this thesis is organized as follows.

Chapter 2: Background and Related Work

This chapter is divided into three parts. First part of Chapter 2 presents a discussion of large scale distributed systems, with a particular focus on P2P content sharing, service discovery and P2P databases, followed by the anatomy of Bloom-filters. Second part of this chapter provides a survey of existing approaches for distributed search, and compares these systems against DPMS and Plexus. Finally, in the third part, we have identified the integral components of a distributed search mechanism based on the foregoing discussion of large-scale distributed systems. This provides a context for the discussions in the subsequent chapters.

Chapter 3: Distributed Pattern Matching System

Chapter 3 presents the detailed design of DPMS - a semi-structured solution to the DPM problem. We start with an overview of the indexing hierarchy, followed by the explanation of the aggregation process. Then we focus on the routing and replication mechanisms in DPMS. Topology maintenance protocols like handling node join and failure are also explained. This chapter also includes mathematical models for estimating query routing efficiency, false match probability and advertisement overhead. Simulation results describing different performance aspects of DPMS are also presented.

Chapter 4: Plexus

Chapter 4 presents Plexus - a structured solution to the DPM problem. We start this chapter with a brief introduction to linear binary codes and the extended Golay code. Then we present the theoretical model of Plexus, where we explain the theory of applying ECC to P2P routing. Then we explain the architecture of Plexus along with the protocols for routing and topology maintenance. Finally, we present the simulation results obtained from the application of Plexus to partial keyword search utilizing the extended Golay code.

Chapter 5: Evaluation

Chapter 5 presents a comparative performance evaluation of DPMS and Plexus against three representative search techniques from unstructured and structured paradigms. Structured techniques cannot efficiently solve the exact DPM problem. Hence, we had to use a DHT-application that can be mapped to the DPM construct, here partial keyword search, to compare the performance. We adopt a simple inverted indexing technique (*i.e.*, 3-gram hashing and mapping to key) in conjunction with *Chord* as the underlying routing mechanism. On the other hand, we used *Flooding* and *Random-walk* as representatives for the unstructured paradigm. We present simulation results for assessing efficiency, flexibility, completeness and robustness of the compared search techniques.

Chapter 6: Conclusion and Future Research

Concluding remarks are presented in Chapter 6. In this thesis, we presented two search techniques: DPMS - a *semi-structured* search mechanism, and Plexus - a *structured* search mechanism. The relative advantages and disadvantages of these two techniques are presented in this chapter. This chapter also presents our future research goals and possible research directions.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter provides a context for the research work presented in the subsequent chapters. In particular, this chapter highlights the properties and functional behavior of the distributed systems of interest to us. Since Bloom filter [30] is used by both DPMS and Plexus for index construction, we have included the anatomy of Bloom filter in this chapter. We also identify the components of a distributed search mechanism and present a classification of different distributed search techniques.

2.2 Chapter Organization

This chapter is organized into three sections. Section 2.3 presents the characteristics of large scale distributed systems, with a particular focus on query and advertisement in P2P content sharing systems, service discovery systems and P2P databases systems. This section also presents the anatomy of the Bloom filters. In Section 2.4 we present prominent research works from the three application domains and compare these systems against DPMS and Plexus. Section 2.5 explains the desirable properties of a distributed

search mechanism. Finally, in Section 2.6 we identify the integral components of a distributed search mechanism based on the foregoing discussion of large-scale distributed systems and present a classification of the search techniques in these domains.

2.3 Background

2.3.1 Large-scale Distributed Systems

Networks of tens or hundreds of thousands of loosely coupled devices have become common in today's world. The interconnection networks can exist in physical or logical dimensions as well as wired and wireless domains. The Internet is the largest distributed system that connects devices through TCP/IP protocol stack. On top of this network there exists many logical overlay topologies, where networked nodes federate to achieve a common goal. Examples of such federations include the Domain Name resolution System (DNS), the World Wide Web (WWW), content sharing P2P systems, world wide service discovery systems and emerging P2P database systems. Among these systems, the WWW and the DNS are mature enough and are characterized by relatively static population of hosts. Content dynamism is also much lower in these two systems, compared to P2P and service discovery systems. Centralized and clustered search techniques (*e.g.*, web crawlers and proxy caches) work well for a network of relatively stable hosts (or web sites) or domain name resolvers. Decentralized (control) and distributed (workload) search techniques are required for a network composed of transient populations of nodes having intermittent connectivity and dynamically assigned IP addresses.

This thesis is devoted to the design of decentralized and distributed search techniques for large-scale distributed systems with transient population of nodes. In the following, we will highlight the characteristics of large scale distributed systems in three representative domains: P2P content sharing, service discovery and P2P databases. The identifying

properties of these application domains include:

Population dynamism : Transient population of nodes mandates the routing mechanism to be adaptive to failures. Redundant routing paths and replication can improve availability and resilience in such environments.

Content dynamism : Frequent arrival of new content, relocation (*e.g.*, transfer) of the existing contents and shorter uptime of peers (compared to internet hosts) are the main causes of content dynamism in these systems. Users in these systems often do not have the exact information (*e.g.*, exact filename, or Service Description) about the content they are willing to discover. Rather most of the queries are partial or inexact, which requires the search mechanism to be flexible.

Heterogeneity : In these systems participating population of nodes display wide variation in capacity, *e.g.*, computing power, network bandwidth and storage. This mandates the index information and routing traffic to be distributed based on nodes' capacities.

The rest of this section presents the characteristics of P2P content sharing, service discovery and P2P database systems. We also explain the nature of queries and advertisements in these systems.

2.3.1.1 Peer-to-Peer Content Sharing

Content (*e.g.*, file) sharing is the most popular P2P application. A classification of the topologies adopted in various P2P content sharing systems can be found in [40]. In [22], a survey and taxonomy of content sharing P2P systems are presented. All content sharing P2P systems offer mechanisms for content lookup and for content transfer. Although content transfer takes place between two peers, the search mechanism usually involves intermediate entities. To facilitate effective search, an object is associated with an index

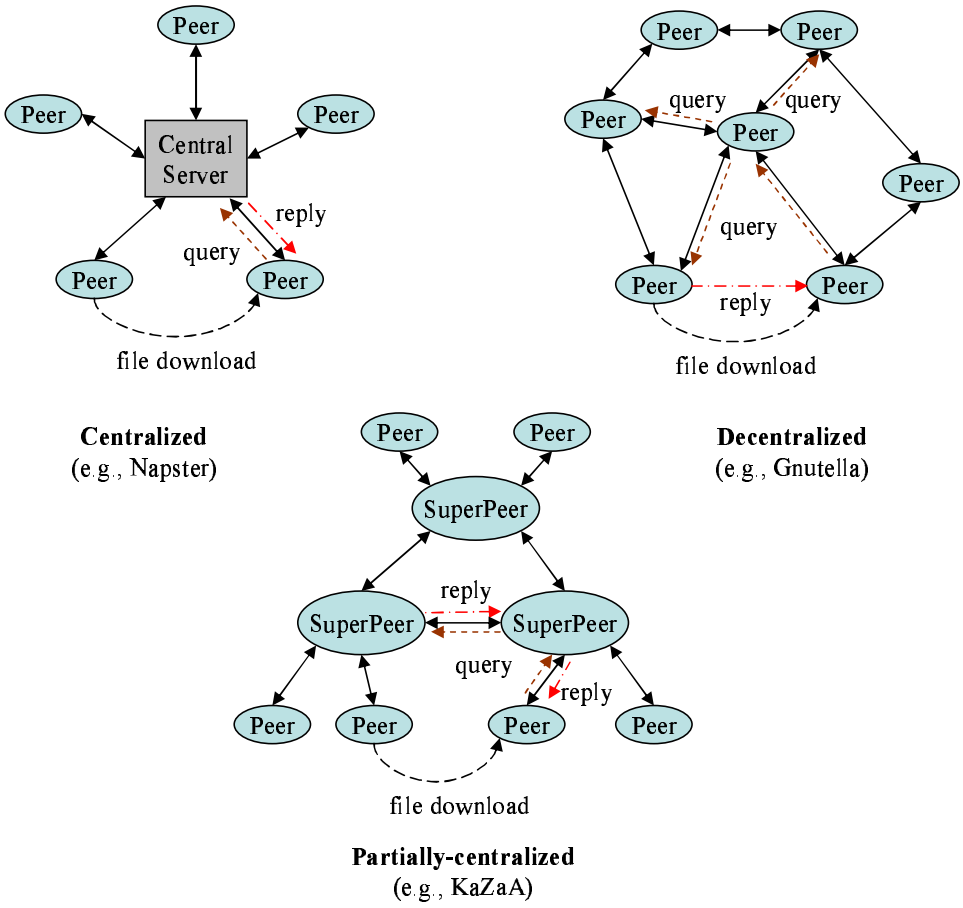


Figure 2.1: Content sharing P2P architectures

file that contains the name, location, and sometimes a description (or keywords) of the content. Search for a content typically involves matching a query expression against the index files. P2P systems differ in how this index file is distributed over the peers (architecture) and what index scheme is used (*i.e.*, index structure). From an architectural point of view (see Figure 2.1), content sharing P2P systems can be *centralized*, *decentralized*, or *partially-decentralized* [22]. **Centralized** P2P systems are characterized by the existence of a central index server, whose sole task is to maintain the index files and facilitate content search. Napster [8] belongs to this category. Centralized P2P systems are highly effective for partial keyword search, but the index system itself becomes a

bottleneck and a single point of failure. **Decentralized** architectures remedy this problem by having all peers index their own local content, or additionally cache the index of their direct neighbors. Content search in this case consists in flooding the P2P network with query messages (*e.g.*, through TTL-limited broadcast in Gnutella [4]). A decentralized P2P system such as Gnutella is highly robust, but the query routing overhead is overwhelming in large-scale networks. Recognizing the benefit of index servers, many popular P2P systems today use **partially-decentralized** architectures, where a number of peers (called superpeers) assume the role of index servers. In systems such as KaZaA and Morpheus, each superpeer has a set of associated peers. Each superpeer is in charge of maintaining the index file for its peers. Content search is then conducted at the superpeer level, where superpeers may forward query messages to each other using flooding. The selection of superpeers is difficult in such a scheme, as it assumes that some peers in the network have high capacity and are relatively static (*i.e.*, available most of the time). A newer version of Gnutella [146] also uses this approach.

The indexing scheme used by content sharing P2P systems can be categorized as *unstructured*, *semi-structured*, or *structured* [22]. **Unstructured** P2P systems use flat index files, where a index file has no relation to other index files. Napster, Gnutella, and KaZaA/Morpheus [5] belong to this category. **Semi-structured** P2P systems, such as Freenet [42] and JXTA [34], use a local routing table at each peer. A search is based on filenames that are hashed to binary keys. The query is routed at each peer to the closest matching key found on the local routing table. To prevent infinite querying, a time-to-live (TTL) value is used. Such mechanism is effective when the content is well replicated over many peers. However, it is virtually impossible to enforce data consistency for file updates. **Structured** P2P systems are specialized in exact matching queries using fully distributed routing structure. Examples of this approach include P2P systems that use distributed indexing and querying schemes, such as Chord [132], CAN

(Content Addressable Network) [118] and Tapestry [150].

Advertisements in these systems mostly contain the filename and author-name. For example a movie file can be advertised as *"The Lord of the Rings - The Two Towers - 2002 (Extended Edition) DVDrip.avi"*. For a user it is very unlikely to know the exact name of the advertised file. Rather the user specifies some keywords that may be present in the advertised file name. For example a typical query for the above movie would be *"Lord of the Ring Two Tower"*. Note the keywords *"Ring"* and *"Tower"*; they do not contain the *"s"* as contained in the advertised keywords. This mandates the support for partial keyword matching in P2P content sharing systems.

2.3.1.2 Service Discovery

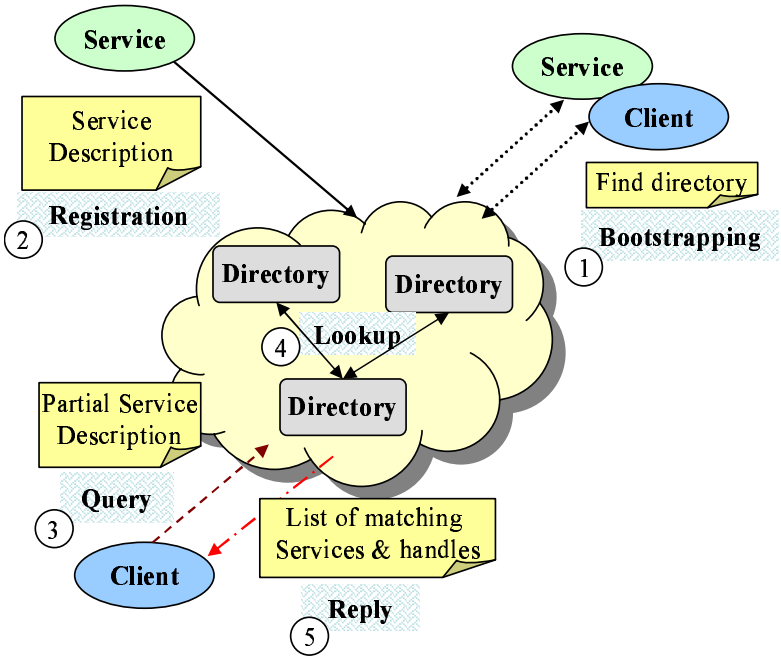


Figure 2.2: Service discovery: Generic architecture and steps

Service discovery is an integral part of any service infrastructure. A large-scale service infrastructure requires a service discovery system that is open, scalable, robust and effi-

cient. Most of the service discovery systems rely on a three-party architecture, composed of *client*s, *service*s and *directory* entities. Directory entities gather advertisements from service providers and resolve queries from clients. The generic service discovery mechanism can be viewed as a five-step process (see Figure 2.2) [18]; (1) *bootstrapping*, where clients and service providers attempt to initiate the discovery process via establishing the first point of contact within the system, (2) *service advertisement*, where a service provider publishes information (a Service Description containing a list of property-value pairs) to a directory entity about the provided service (3) *querying*, where a client looks for a desired service by submitting a query (usually a partial Service Description) to a directory entity, (4) *lookup*, where the directory entity searches the network of directory entities for all Service Descriptions matching the query and (5) *service handle retrieval*, the final step in the discovery mechanism, where a client receives the means to access the requested service. Some of these steps may be omitted in various discovery approaches. Some of the discovery approaches are based on two-party (client-server) architecture without any directory infrastructure.

Directory architectures adopted by different service discovery approaches can broadly be classified as *centralized* and *decentralized* (see Figure 2.3) [18]. In a centralized architecture, a dedicated directory entity or registry maintains the whole directory information (as in centralized UDDI [141]), and takes care of registering services and answering to queries. In decentralized architectures, the directory information is stored at different network locations. Decentralized systems can be categorized as *replicated*, *distributed* or *hybrid*. In the replicated case, the entire directory information is stored at different directory entities (as in INS [13]). In the distributed case, the directory information is partitioned, and the partitions are either stored in dedicated directory entities (DA) (as in SLP [70], Jini [136] and SSDS [50]) as per a three-party model or cached locally by the service providers in the system (*e.g.*, UPnP [106] and SLP in DA-less mode), according

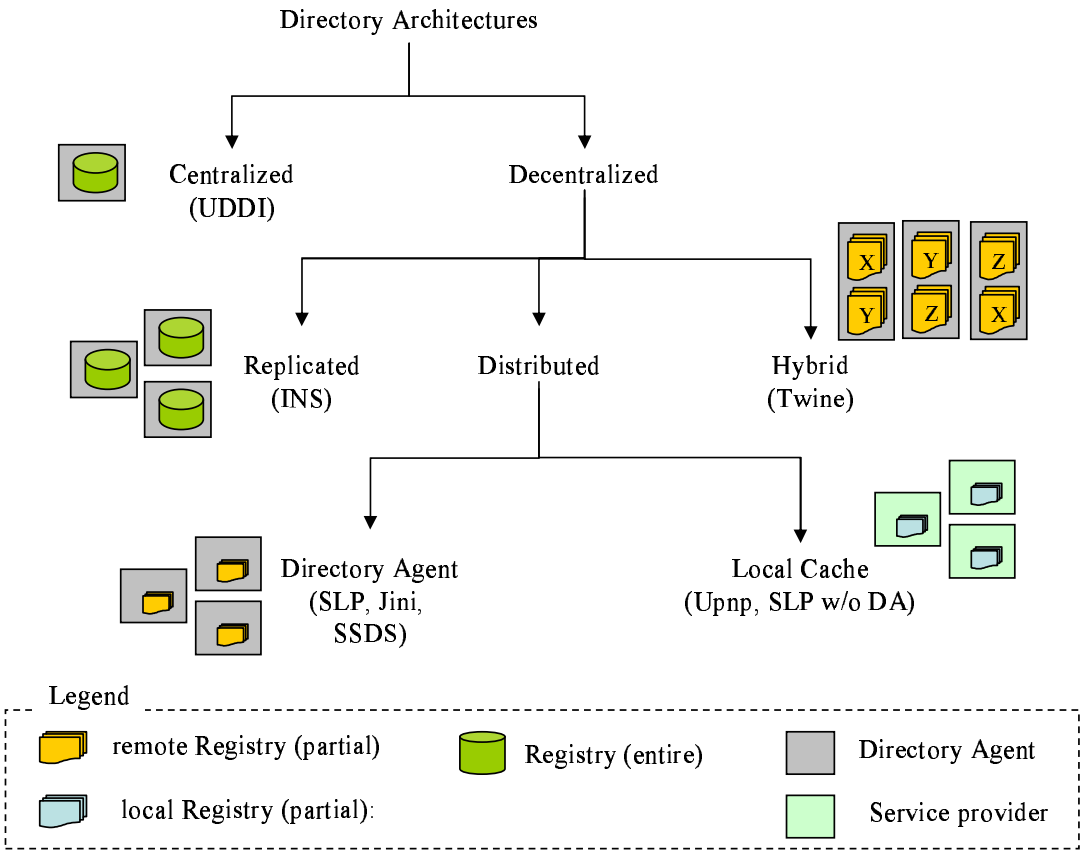


Figure 2.3: Taxonomy of the directory architectures

to a two-party model. Finally, in the hybrid case, the system stores multiple copies of the entire directory information without assigning the entire registry to a single directory entity (as in Twine [27]).

In large-scale networks, a centralized directory becomes a performance bottleneck and a single point of failure. Consistency of the replicas is a major issue in the replicated architecture (like INS), since maintaining consistent replicas is usually bandwidth-consuming. On the other hand, when the directory information is distributed, *e.g.*, partitioned among dedicated directory entities, the failure of one of them leads to the unavailability of part of the directory information. The fully distributed two-party architecture attempts to remedy all these issues, however these systems generally do not scale well, since they use

multicast-like communications which are expensive in terms of bandwidth. Hybrid architectures seem to offer the best compromise between bandwidth consumption, scalability, and fault-tolerance.

Advertisement	***Query***
Service-type=service:print	Service-type=service:print
Scope-list=staff, grad	*Scope-list=grad*
Location=DC3335	*Paper-size=A4*
color=true	
language=PS	
Paper-size= legal, A4, B5	
URL=diamond.uwatreloo.ca/PCL8	

Figure 2.4: Example advertisement and query in Service Discover systems

Figure 2.4 gives an example of a generic advertisement and a query in service discovery systems. In these systems a service is advertised using a list of descriptive property-value pairs, called a *Service Description*. A Service Description typically contains service type (*e.g.*, *Service-type=service:print*), service invocation information (*e.g.*, *URL=diamond.uwatreloo.ca/PCL8*) and service capabilities (*e.g.*, *Paper-size= legal, A4, B5*). In most cases a Service Description is instantiated from a *Service Schema*, which contains meta-information regarding the Service Descriptions for a given class of service (*e.g.*, print service or *service:print*). A Service Schema governs the allowable properties and their types (*e.g.*, string, integer, float, *etc.*) within the Service Descriptions of a given class of services. In most service discovery systems it is assumed that the available Service Schemas are globally known.

Queries in these systems (see Figure 2.4) usually contain the requested service type and a list of required capabilities of the service (*e.g.*, Paper-size=A4). The list of capabilities provided in a query is a subset of the capabilities list provided in the advertisements it should match against. The result of a query consists of a list of Service Descriptions

matching the query.

2.3.1.3 Peer-to-Peer Databases

Peer-to-peer Database Systems (PDBS) have been investigated, more recently, following the success of P2P file-sharing. A P2P database system can be thought of as a data sharing network built on top of a P2P overlay substrate. Search in P2P database systems demands more flexibility than that required by the P2P file-sharing systems. This requirement stems from the existence of semantic (schema) information associated with the shared data. Most of the research works focus on building an additional layer on top of the existing P2P search techniques.

Though PDBSs evolved as a natural extension of Distributed Database Systems (DDBS) [113], they have a number of properties that distinguish them from the DDBS and traditional Database Management Systems (DBMS) [117]. Unlike DDBS, PDBS has no central naming authority, which results into heterogenous schemas in the system. Due to the absence of any central coordination and the large-scale evolving topology, a peer knows about only a portion of the available schemas and data. This mandates a mechanism (*e.g.*, ontology) for unifying semantically close schemas. In a DDBS, arrival or departure of nodes is performed in a controlled manner, which is not true for PDBSs. Finally, in contrast to DDBS, a peer in a PDBS has full control over its local data.

In PDBS, semantic mapping of schema is a challenging problem. It requires interoperation between heterogenous data models. XML [130] is used as the defacto standard for this purpose. A survey on the use of XML in PDBS can be found in [92]. In PDBS, XML is used in two ways. Firstly, XML is used for representing data and data models (*i.e.*, schema information). Secondly, XML is used to represent semantic relationships among heterogeneous data models at three different levels: schema level, element level and data level. These levels of granularity also influence the indexing mechanism adopted

in these systems.

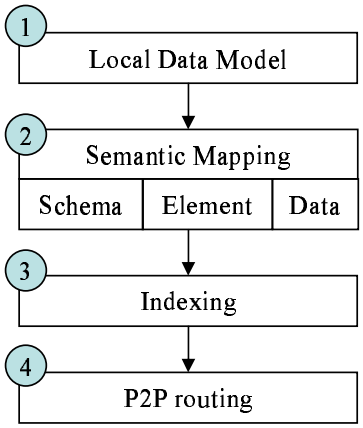


Figure 2.5: Functional layers in a PDBS system

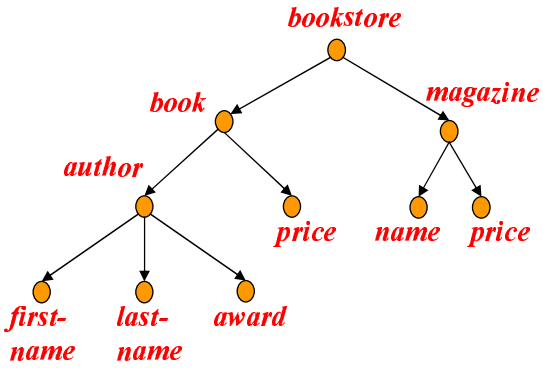
Figure 2.5 presents the possible functional layers in a PDBS. Each peer in the system has its own local data model independent of the other peers' data models. The process of translating a local query to other peers' data models is performed by the semantic mapping layer at different granularities. The third layer is optional, and can maintain indices at different granularities. Finally, the fourth layer is usually one or a combination of the routing mechanisms present in traditional file sharing P2P systems.

Many research works on PDBS assume the existence of an underlying P2P substrate for efficient and flexible query routing, and concentrate on higher level issues including semantic mapping between heterogenous schemas and distributed query processing and optimization. In this section, we will consider only those research activities on PDBS that have focused on the issues and challenges related to the query routing mechanism.

Advertisement and query in PDBS are more complicated than that in P2P content sharing and service discovery systems. Figure 2.6 depicts an example of an XML advertisement which contains information about two books and one magazine. A tree representation of the corresponding *XML Schema* [57] has been presented in Figure 2.6(b). Analogous to Service Schema, an XML Schema contains meta-information regarding a

```
<bookstore specialty="novel">
  <book style="autobiography">
     <author>
          <first-name>Joe</first-name>
          <last-name>Bob</last-name>
          <award>Trenton Literary </award>
     </author>
     <price currency=CAD>12</price>
  </book>
  <book style="textbook">
     <author>
        <first-name>Mary</first-name>
        <last-name>Bob</last-name>
     </author>
     <price>55</price>
  </book>
  <magazine>
     <name>Times</name>
     <price currency=USD>4</price>
  </magazine>
</bookstore>
```

(a) An XML advertisement



(b) Tree representation

Figure 2.6: Advertisement in PDBS

class of XML documents. However, the syntax used for describing XML documents and XML Schema are standardized and widely used, compared to the variations in Service Description and Service Schema definition syntaxes used by different service discovery systems.

The most popular query syntax used in PDBS is XPath. Figure 2.7 presents two examples of XPath queries based on the advertisement presented in Figure 2.6. The first query finds all *author*s having at least one *award*. The second example finds all books

XPath Query

- `//author[award]`
- `/bookstore/book[author/last-name=`*Bob*`]`

Figure 2.7: XPath query examples

for which *last-name* of the *author* is *Bob*.

2.3.2 Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure that is used to represent a set. Bloom filters can support set membership test operations with a small probability of false (erroneous) positives. Bloom filters and their variants are used to solve several network problems, due to their compact representation. For network applications, bandwidth saving achieved with Bloom-filters outweighs the drawback of false positive in membership test. A comprehensive list of such applications can be found in [33].

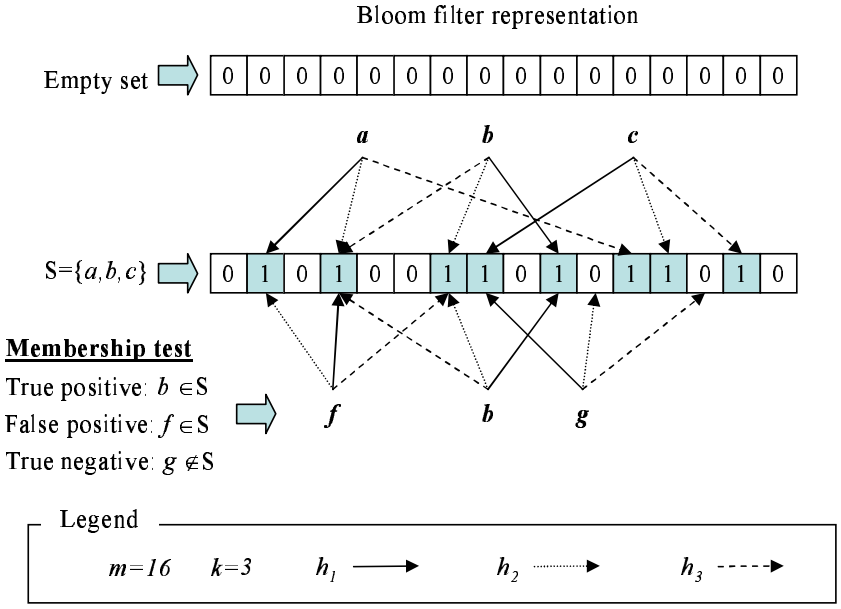


Figure 2.8: Bloom filter: Construction and membership test

The algorithm for Bloom filter construction is simple. A Bloom filter, is represented as an m-bit array. $\hbar$ different hash functions $(h_1, h_2, \ldots, h_{\hbar})$ are also required. Each of these hash functions should return values uniformly distributed over the range of $[0..m)$. In an empty Bloom filter all of the m-bits are set to 0. An element (a string or keyword)

s_i of a set S, to be inserted into a Bloom filter, is hashed with the $\hbar$ hash functions and corresponding $\hbar$-bits in the array are set to 1. To test set membership for an element, it is hashed with the same $\hbar$ hash functions to obtain $\hbar$ array positions. If all of these $\hbar$-bits are 1, then the element is a member of the set represented by the Bloom filter with high probability, otherwise it is not. As depicted in Figure 2.8, the possible outcomes of the membership test operation are as follows:

1. **True positive**: The set contains the element and membership test operation returns true.

2. **True negative**: The set does not contain the element and membership test operation returns false.

3. **False positive**: The set does not contains the element but membership test operation returns true.

It should be noted that the fourth case, *i.e.*, the **false negative**, where the set contains the element but the test membership test operation returns false, will never occur. False positive probability, say ϵ, for a Bloom filter containing ω elements can be estimated as follows:

$$\epsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{\omega\hbar}\right)^{\hbar}$$

here, $p = 1-(1-1/m)^{\omega\hbar}$ is the probability that a particular bit is set to 1. False positives probability is minimized when $\hbar = \ln 2 \cdot (m/\omega)$. For example with $\frac{m}{\omega} = 6$ and $\hbar = 3$ the probability of false positives is about $\epsilon \approx 0.06$. For a well designed Bloom filter with low false positive rate, about $33 \sim 50\%$ of the bits are equal to 1.

Now we focus on an important property of the Bloom filters. Let $\mathcal{B}(X)$ denote the Bloom filter representation of set X. For any two sets S_1 and S_2 we can get

$$\mathcal{B}(S_1 \cup S_2) = \mathcal{B}(S_1) \vee \mathcal{B}(S_2) \tag{2.1}$$

In other words, the Bloom filter obtained from the union of two sets is the same as the bit-vector obtained from the bit-wise OR of the individual Bloom filters representing the constituent sets. This property can be generalized for any number of sets. We will use this property of Bloom filters for the aggregation process in DPMS.

2.4 Related Work

This section is organized into three subsections corresponding to three application domains: P2P content sharing, service discovery and PDBS. We consider the research works that focused on solving some variant of the partial matching problem that can be mapped to the DPM construct. For example we do not consider here DHT techniques, like Chord [132], CAN [118], Pastry [121], Kademlia [105] *etc.*, since these techniques focus on exact matching and hence are not related to the problem at hand. Rather we discuss the research works like Twine [27], Squid [127], pSearch [138] which strive to extend DHT-functionality for achieving partial matching capability.

2.4.1 Peer-to-Peer Content Sharing

2.4.1.1 Structured techniques

DHT-based Solutions

In general DHT-techniques (Chord [132], CAN [118], Tapestry [150]) are not suitable for solving the partial keyword matching problem (and the DPM problem) mainly for two reasons. Firstly, DHT-techniques use numeric distance based clustering of patterns which is not suitable for pattern matching and results into multiple DHT-lookups per search (see Section 1.2). Secondly, DHT-techniques cannot handle *common keywords problem* [102] well. Popular keywords (or η-grams like "tion" or "ing") can incur heavy load on the peers responsible for these keywords (or η-grams); as a result, the distribution of load

will become unbalanced among the participating peers.

Inability to support partial keyword matching is considered a handicap for DHT-techniques. In the last few years a number of research efforts have focused on extending DHT-techniques for supporting keyword search. Most of these approaches adopted either of the following two strategies:

- Build an additional layer on top of an existing routing mechanism, like Chord [132], CAN [118] or Tapestry [150]. The aim is to reduce the number of DHT lookups per search by mapping related keywords to nearby peers on the overlay. This strategy is proposed in a number of research works including [85], [102], [127] and [138] .

- Combine structured and unstructured approaches in some hierarchical manner to gain the benefits of both paradigms. Few research works focus on this strategy including [59], [82] and [137].

However none of these approaches achieve satisfactory performance for all of the design requirements described in Section 2.5.

A **generic inverted index** on top of a DHT-based network for solving partial-keyword matching has been proposed in [72]. A keyword can be fragmented into η-grams, and each η-gram can be hashed and stored at the responsible peer. This approach can solve partial keyword matching problem in $O(\omega \log N)$ time, where ω is the number of η-grams in a query and N is the number of peers in the system. However, solving the generic DPM problem with this approach will require $O(2^{\lambda} \log N)$ time, where λ is the number of 1 bits in a query (or advertisement) pattern.

Keyword fusion, presented in [102], is also an inverted indexing mechanism on top of Chord. It supports keyword search only. A document advertised with keywords $\{k_1, k_2, \ldots, k_t\}$ is routed to peers responsible for keys $h(k_1), h(k_2), \ldots, h(k_t)$, where $h(\cdot)$ is the DHT hash function. To reduce the number of DHT-lookups per search, a system-wide dictionary of common keywords is maintained. A query is routed using the most specific

keyword and then filtered using the more common keywords specified in the query. In contrast to DPMS and Plexus, advertisement and replication overhead in this system is proportional to the number of keywords associated with the document. This fact is also reflected in the comparative evaluation presented in Chapter 5. A similar inverted indexing mechanism for web page indexing has been proposed in [148].

Joung et al. [85] proposed a distributed indexing scheme, build on a logical, *d*-dimensional hypercube vector space over a DHT network (they used Chord for their experiment). In this scheme each advertised object is mapped to a d-bit vector according to its keyword set (similar to Bloom filter construction). They treat d-bit vectors as points in d-dimensional hypercube. No restriction on the mapping of a d-dimensional point to a 1-dimensional key space (required for Chord) has been specified. An advertisement is registered to the peer responsible for the d-bit advertisement vector. A query vector (say Q) is computed in the same manner as the advertisement vector. A query is routed to the peers in the Chord ring that are responsible for a key (say P_i) that is a superset of the query vector Q.

The work by Joung et al. [85] and the inverted indexing method presented in [102] represent the two extremes of advertisement and query traffic trade off. In [85], an advertisement is registered at one peer (responsible for the advertised bit vector) and a query is routed to all possible peers that may contain a matching advertisement. In turn, in [102] an advertisement is registered at all the peers responsible for the advertised keywords and the query is routed to the peer responsible for the most uncommon keyword specified in the query.

pSearch [138] utilizes Information Retrieval (IR) techniques on top of CAN (Content-Addressable Network) for facilitating content-based full-text search. Keywords associated with an advertised document (or query) are represented as unit vectors. IR techniques like vector space model (VSM) and latent semantic indexing (LSI) are used to compute a unit

vector from the keyword list specified with a document (or query). Similarity between a query and a document (or between two documents) is measured using the cosine (*i.e.*, vector dot product) of the vector representation of the corresponding documents or query. Semantically close documents and queries are expected to be mapped to geometrically close point vectors in the Cartesian space. Now the semantic point vectors from LSI or VSM are treated as geometric points in the Cartesian space of CAN. CAN partitions a d-dimensional, conceptual, Cartesian space into zones and assigns each zone to a peer. However this mapping technique uses the same dimensionality for LSI space and CAN. Thus it needs to have a priori knowledge of the possible keywords (or terms) in the whole system. In reality there can be hundreds of possible keywords, and CAN performance degrades at higher dimensions. Moreover pSearch supports multiple keywords search only, whereas Plexus and DPMS support multiple partial-keyword queries.

Squid [127] has been designed to support partial prefix matching and range queries on top of DHT-based structured P2P networks. In this system Hilbert Space-filling Curve (HSFC) [122] has been used on top of Chord. HSFC is a special type of locality preserving hash function that can map points from a d-dimensional grid (or space) to a 1-dimensional curve in such a way that the nearby points in d-dimensional space are usually mapped to adjacent values on the 1-dimensional curve. Squid converts keywords to base-26 (for alphabetical characters) numbers. A d-dimensional point is constructed from d keywords specified in the query or advertisement. Then a d-dimensional HSFC is used to map a d-dimensional region (*i.e.*, set of points) specified by the query into a set of curve segments in 1-dimension. Finally, each segment is searched using a DHT-lookup followed by a local flooding. Squid supports partial prefix matching (*e.g.*, queries like compu* or net*) and multi-keyword queries; however, Squid does not have provision for supporting true inexact matching of queries like **net**, as supported by Plexus and DPMS. Another major problem is that the number of (partial) keywords specified in a

query or advertisement is bounded by the dimensionality d of the HSFC in use.

In [82] a hybrid approach to keyword search, named **MKey**, has been proposed. Architecturally there exists a DHT (here Chord) backbone. A backbone node in the Chord ring works as a head for a cluster of nodes, organized in an unstructured fashion. Search within a cluster is based on flooding. On the other hand, Bloom filter is used as index in the backbone. But DHT techniques do not allow Hamming distance based indexing as required for matching Bloom filters. For allowing pattern matching on Chord, the following strategy is used. Nodes on the Chord ring are allowed to have an ID with at most two 1-bits. An advertisement pattern, say 01010111, is advertised to peers 01010000, 00000110 and 00000001; *i.e.*, DHT-keys are obtained from an advertisement pattern by taking pairs of 1-bits in sequential order from left to right. To construct DHT-keys from a query pattern, say 01010011, only the leftmost three 1-bits are used. In this example the 1-bits at 2^{nd},4^{th} and 7^{th} positions. The DHT-keys are obtained by taking the 1-bit in center position (here 4^{th}) and another bit within the left position (here 2^{nd}) and the right position (here 7^{th}). Hence for the query pattern 01010011, generated DHT-keys are 01010000, 00110000, 00011000, 00010100 and 00010010. The indexing scheme used in this work is a good illustration of the weakness of DHT techniques in solving the DPM problem, as explained in Section 1.2. Evidently the number of DHT-lookups per search or advertisement depends linearly on the number of keywords and the size of the used Bloom-filter. This can be more inefficient than simple inverted indexing mechanism for inappropriate parameter settings. Besides, the backbone nodes may become performance bottlenecks for the system.

Non-DHT Solutions :

There exists only a few non-DHT structured approaches to the search problem in P2P networks. **SkipNet** [73] and **SkipGraph** [24] are prominent among them. Both of these approaches use *Skip List* [115], which is a probabilistic data structure. A Skip

List consists of a collection of ordered linked lists arranged into levels. The lowest level (*i.e.*, level 0) is an ordinary, ordered linked list. The linked list in level i skips over some elements from the linked list at level $(i-1)$. An element in level i linked list can appear in level $(i+1)$ linked list with some predefined, fixed probability, say p. Storage overhead can be traded for search efficiency by varying p.

Search for an element say Q starts at the topmost level. Level i list is sequentially searched until Q falls within the range specified by current element and next element in the list. Then the search recurs to level $i-1$ list from the current element until level 0 is reached.

In both SkipGraph and SkipNet, nodes responsible for the upper level elements of the Skip List become potential hot spots and single points of failure. To avoid this phenomena, additional lists are maintained at each level. This in turn increases the degree of each node.

A multi-level indexing mechanism for keyword search based on SkipNet has been proposed in [129]. However, none of these approaches can efficiently support partial keyword search because the underlying data structure used by these techniques, *i.e.*, Skip List, supports prefix matching only.

2.4.1.2 Non-structured Techniques

Unstructured systems ([4],[3]) identify objects by keywords. Advertisements and queries are expressed in terms of the keywords associated with the shared objects. Structured systems, on the other hand, identify objects by keys, generated by applying one-way hash function on keywords associated with an object. Key-based query routing is much efficient than keyword-based unstructured query routing. The downside of key-based query routing is the lack of support for partial-matching semantics as discussed in the previous section. Unstructured systems, utilizing blind search methods such as *Flooding* [4] and

Random-walk [103], can easily be modified to support partial-matching queries, and in general to solve the DPM problem. But, due to the lack of proper routing information, the generated query routing traffic would be very high. Besides, there would be no guarantee on search completeness.

Many research activities are aimed at improving the routing performance of unstructured P2P systems. Different routing hints are used in different approaches. In [39], routing is biased by peer capacity; queries are routed to peers of higher capacity with higher probability. In [140] and [146], peers learn from the results of previous routing decisions and bias future query routing based on this knowledge. In [44], peers are organized based on common interest, and restricted flooding is performed in different interest groups. Many research works ([39], [99], [146], *etc.*) propose storing index information from peers within a radius of 2 or 3 hops on the overlay network. All of these techniques reduce the volume of search traffic to some extent, but none provides guarantee on search completeness.

Bloom filters are used by a few unstructured P2P systems for improving query routing performance. In [99] each peer stores Bloom filters from peers one or two hops away. Three ways of aggregating Bloom-filters are also presented. Experimental results presented in [99] show that logical OR-based aggregation of Bloom filters is not suitable for indexing information from peers more than one hop away. In [119] each peer stores a list of Bloom filters per neighbor. The i^{th} Bloom filter in the list of Bloom filters for neighbor M summarizes the resources that are $i-1$ hops away from neighbor M. A query is forwarded to the neighbor with a matching Bloom filter at the smallest hop-distance. This approach aims at finding the closest replica of a document with a high probability. An approach similar to [119] has been presented in [96], which uses an exponentially decaying Bloom filter for indexing neighbor content.

2.4.2 Service Discovery

Service discovery is another application area of the DPM problem. Many service discovery systems rely on a three-party architecture, composed of clients, services and directory entities. Directory entities gather advertisements from service providers and resolve queries from clients. Major protocols for service discovery from industry, like SLP [70], Jini [136], UPnP [106], Salutation [123], etc, assume a few directory agents, and do not provide any efficient mechanism for locating Service Descriptions. Solutions from academia, like Secure Service Discovery Service (SSDS) [50] and Twine [27], target Internet-scale service discovery and face the challenge of achieving efficiency and scalability in locating Service Descriptions based on partial information. There also exists a few discovery architectures that address the inter-operability issue and try to achieve service discovery capability across heterogeneous technology domains. Such approaches, including OSDA (Open Service Discovery Architecture) [101], Meta Service Discovery [35], [20] and [93], strive to achieve the conflicting goals of efficiency and flexibility as well. A survey of the prominent service discovery mechanisms can be found in [18], while a survey on the naming and Service Description schemes used in service discovery techniques and in general in distributed systems can be found in [19].

From an architectural point of view, Secure Service Discovery Service (SSDS) [50] is the closest match to DPMS. Like DPMS, SSDS uses Bloom filters and aggregation, and index distribution is through a hierarchy of indexing nodes. SSDS uses a tree-like hierarchy for index distribution, in contrast to the lattice-like hierarchy used by DPMS. Unlike DPMS, SSDS does not use any replication in the indexing hierarchy; as a result, higher level nodes in SSDS index tree handle higher volumes of query/advertisment traffic and the system is more sensitive to the failure of these nodes. Another major drawback of SSDS, compared to DPMS, lies in its aggregation mechanism. SSDS uses bitwise-OR for index aggregation. On the contrary, the don't care based aggregation scheme, adopted

in DPMS, retains unchanged bits from the constituent patterns and provides more useful information during query routing.

Twine [27] uses a hierarchical naming scheme and relies on Chord as the underlying routing mechanism. A resource is described using a *name-tree*, composed of the properties and values associated with the resource. Hierarchical relations between properties are reflected in the tree, *e.g.*, while describing the location of a resource, "room no." appears as a child of the "building" in which it resides. Twine generates a set of strands (substrings) from the advertisement or query (which are expressed in XML format), computes keys for each of these strands, and finally performs search or advertisement using these keys. The number of DHT-lookups increases with the number of property-value pairs in the advertisement (or query) and consequently the amount of generated traffic becomes high. Load-balancing is another major problem in this system. Peers responsible for small or popular strands become overloaded, and the overall performance degrades. The stranding algorithm in Twine is designed to support partial prefix matching in a name-tree. This problem can be easily mapped to the DPM problem. Strands of different lengths can be hashed into a Bloom filter, which can be used as the signature of a name-tree describing a resource.

Web Services (WS) [32] provide a standard way of interoperating between different software applications, running on a variety of platforms and/or frameworks. Universal Description, Discovery and Integration (UDDI) [141] is the defacto standard for WS discovery. Many research activities are devoted to enhancing and overriding the legacy UDDI specification thriving for efficiency, scalability and flexibility in the discovery mechanism. A detailed survey of such activities can be found in [61]. Table 2.1 summarizes some of the proposed architectures for WS discovery. Among these approaches we are interested in decentralized P2P-based architectures. Based on the use of WS ontologies, these approaches can be broadly classified as *semantic-laden* and *semantic-free*.

Table 2.1: Summary of Web Service discovery architectures

Centralized	Registry	Authoritative, centrally controlled store of service descriptions, *e.g.*, UDDI registry [141]	
	Index	Non-authoritative, centralized repository of references to service providers; see [32] for details. Web crawlers are used for populating an index database.	
Decentralized	Federation	Publicly available UDDI nodes collaborate to form a federation and act together as a large scale virtual UDDI registry [120].	
	P2P-based	Semantic-laden	In [125] peers are arranged into a hypercube topology [52] and ontology [142] is used to facilitate efficient and semantically-enabled discovery. An agent-based approach is proposed in [108]. It uses DAML [36] representation for ontology and relies on unstructured search techniques.
		Semantic-free	Both [100] and [128] use Chord overlay for indexing and locating service information. [100] extracts property-value pairs from service descriptions and uses MD5 hashing. [128] uses Hilbert Space Filling Curves for mapping similar Service Descriptions to nearby nodes in the Chord ring. These two approaches are similar to Twine [27] and Squid [127], respectively. In [78], another Chord based solution has been proposed. Here, the ID-space is partitioned in numerically ordered subspaces, and each peer in the Chord-ring maintains links to one peer in each subspace in addition to the regular Chord links.

Semantic-laden approaches rely on WS ontology mapping techniques like OWL (Web ontology language) [23] or DAML (DARPA Agent Markup Language) [36] for incorporating intelligence to the discovery process, *i.e.*, for intelligently mapping conceptually related terms in queries and advertisements. Semantic-free approaches, on the other hand, do not utilize WS ontology mapping techniques. These techniques are closely related to the traditional service discovery approaches. A number of research work in this category rely on locality preserving hash techniques for mapping queries to semantically close advertisements.

All of the decentralized techniques for WS discovery are aimed at achieving flexibility (*i.e.*, partial matching capability) without sacrificing efficiency in the search mechanism. The basic problem of semantic matching in a distributed environment can be mapped to the DPM problem in different ways. For example, while generating an advertisement pattern from a Service Description, multiple synonyms of the properties and values can be hashed and inserted into the Bloom-filter. This will require larger Bloom-filters, yet will enable one to discover a service by specifying any of the synonyms of a property (or value) specified in the advertisement. It is also possible to use ontologies during Bloom filter construction, rather than using simple synonym list. Use of Ontology can be more efficient than synonym list if there exists a global Ontology in the system.

2.4.3 Peer-to-Peer Databases

2.4.3.1 Structured Techniques

Several research works on PDBS have adopted DHT-based and structured P2P techniques, such as Chord [132], CAN [118] and Hypercube [125], for routing. A number of these proposals, including [31], [37] and [58], rely on Chord as the underlying P2P substrate. Hypercube topology has been used in [111].

XP2P, presented in [31], uses XML data model for schema representation, and pro-

vides support for resolving XPath [90] queries. Any XML document can be represented as a tree, and an XPath query is used to specify a subtree using a prefix-path originating from the root of the document. For supporting partial prefix-path matching, all possible paths, originating from the root, have to be registered with the Chord ring. To reduce the number of paths to be hashed in the Chord ring during the advertisement and query process, XP2P adopts a fingerprint construction technique presented in [116]. In this technique, the fingerprint of a binary string $A(t) = (a_1, a_2, \ldots, a_m) = a_1 \times t^{m-1} + a_2 \times t^{m-2} + \cdots + a_m$ is computed as $f(A) = A(t) \mod P(t)$, where $P(t)$ is an irreducible polynomial. A useful property of the fingerprint function, utilized by XP2P, is that $f(A \odot B) = f(f(A) \odot B)$, where $\odot$ is the concatenation operator.

In [58], a Chord-based framework for supporting XPath queries is presented. XPath queries of the form $/a_1[b_1]/a_2[b_2]/\ldots/a_n\, op\ value$ and queries containing relative path operator (*i.e.*, //) are supported. Here, a_i is an element in an XML document, b_i is an XPath expression relative to element a_i, *op* is an XPath operator like = or <, and *value* is an atomic element in the XML document. The core idea is to build a distributed catalog, where a peer in the Chord ring stores all the prefix-paths for a given element in any XML document stored in the network. In other words, if E is an element in some XML files, then the peer responsible for the key $hash(E)$ stores all the absolute paths (*i.e.*, $/a_1/a_2/\ldots/E$) leading to E in any document stored in the network and the contact information of the peers storing those documents. An XPath query of the form $/a_1/a_2/\ldots/a_k//E$ is routed to the peer (say N) responsible for the key $hash(E)$ and the list of all peers containing XML documents matching the query are extracted. Finally the query is forwarded and executed in the corresponding peers. This approach of hashing is similar to the approach adopted in Twine [27], where paths of different lengths are hashed and stored in the Chord network, instead of hashing individual elements along the path.

Another Chord-based framework, named RDFPeers, for distributed search on PDBS is presented in [37]. In this work, Resource Description Framework (RDF) [97] is used as data model. An RDF document is essentially a set of $< Resource, Property, Value >$ triples presented in XML format. Each triple is stored in three peers (in the Chord ring) responsible for the keys $hash(Resource)$, $hash(Property)$ and $hash(Value)$, respectively. For string literals SHA1 hash function is used. For numeric values (in the *value* component of a triple) locality preserving hash function is used. A query can be constructed by specifying any of the three components in a triple. It should be noted that RDFPeers assumes XML documents of fixed depth (*i.e.*, 3), in contrast to variable depth XML documents assumed in [31] and [58].

2.4.3.2 Non-structured Techniques

PeerDB [112] uses an agent-based framework on top of an unstructured P2P overlay to achieve distributed data sharing. To accommodate heterogeneity in schema definitions from autonomous peers in the system, PeerDB associates keywords as synonyms with each schema and elements under that schema. These keywords are used as a means of semantic mapping and for finding semantically similar schemas using P2P keyword search techniques. Mobile agents are sent to appropriate peers and a query is executed locally at the target peer, which helps in reducing the volume of network traffic.

JXTA [34] routing protocol has been used in [55] and [91]. In JXTA architecture a loosely-consistent distributed hash table is maintained by a special set of peers called Rendezvous peers. Each rendezvous peer maintains a list of known Rendezvous peers and the range of keys associated with each of them. Query routing is performed based on the local information at each Rendezvous peer. In [91], a fixed global schema has been used, whereas existence of heterogenous schema is allowed in [55].

A hybrid technique, named Humboldt discoverer, has been presented in [75]. RDF [97]

has been used for describing an advertised resource. SPARQL (Simple Protocol and RDF Query language) [114] has been used for constructing query expressions. SPARQL is a query language for RDF documents developed by W3C. A three tier architecture has been proposed. Regular peers providing information sources are placed at the bottom tier of the hierarchy. These peers are clustered into groups based on the similarity of used ontologies. A peer at the middle tier is responsible for an ontology and manages a single cluster of regular peers (at bottom tier). Middle tier peers advertise their existence to top level peers, which are organized in a Chord ring and are addressed by the hash of the URIs of the ontologies. In effect, middle tier peers covering the same ontology are grouped under the same top level peer. To resolve a query, all the required ontologies are first determined. For a given ontology, the set of responsible middle-tier peers can be reached through the top-tier Chord network. Finally, the query is forwarded to each of the middle-tier peers that are responsible for the ontology.

2.4.3.3 Relation to the DPM Problem

Distributed search in PDBSs faces two new challenges, in addition to the ones present in P2P content sharing and service discovery: continuously changing schemas in the system and the requirement for semantic mapping. It is possible to cope with these challenges using the DPM construct. In order to accommodate continuously changing schemas, advertised patterns should be constructed from both the descriptive properties and values of a shared object. Thus, schema information gets incorporated within each advertisement. The requirement for semantic mapping can be satisfied in few ways, including the one discussed in the last paragraph of Section 2.4.2. It is also possible to reserve a pre-specified number of bits in the advertisement/query pattern, and use these bits as a separate Bloom-filter for storing the ontologies used by the advertisement/query. Query routing mechanism can use this additional information for making semantic laden

decisions at each hop.

2.5 Distributed Search Requirements

Search is an essential functionality offered by any distributed system. A search mechanism in a distributed system can be either centralized or distributed. For *Centralized Search* there exists a central core of one or more machines responsible for indexing the contents distributed across the network and for responding to user queries. For networks with lesser degree of dynamism, centralized search mechanisms prove to be adequate. Google [28], Yahoo [12], Alta vista [1] *etc.* are the living examples of centralized search mechanisms, where a set of crawlers running on a cluster of computers index the Webpages around the globe. Compared to the lifetime of the contents shared in P2P networks, Webpages are long lived. Centralized search techniques do not prove to be efficient in large scale distributed systems due to content and node dynamism (as explained in Section 2.3.1). *Distributed Search* mechanisms assume that both indexing mechanism (analogous to crawlers) and indexed information are distributed across the network. Consequently the design requirements for Distributed Search techniques are different from that for Centralized Search techniques. In the following, we present the most import design requirements for a Distributed Search mechanism.

Decentralization : For a Distributed Search mechanism to be successful, decentralization of control and data are necessary. Decentralization of control refers to the distribution of the index construction process among the participating nodes. There should not be any central entity governing the index construction process in different nodes. Unlike web search engines, the index itself should be distributed across the participating nodes for achieving uniform load distribution and fault-resilience.

Efficiency: The search mechanism should be able to store and retrieve index infor-

mation without consuming significant resource: storage and bandwidth. In LSDS advertisements are frequent due arrival of new documents and relocation of existing documents. The large user base generates queries at a high rate. This mandates both advertisement and search process to be bandwidth efficient.

Scalability: Efficiency of the search mechanism should not degrade with increase in network size. In addition the number of links per node should not increase a lot with growth in network size. Join and topology maintenance overhead depends largely on the number of links that a node has to maintain, especially in dynamic environments.

Flexibility: Due to content dynamism, users do not usually have the exact information about the advertised objects. The query semantics offered by the search mechanism should be flexible to support inexact or subset queries. The scalability and efficiency requirements should not be sacrificed for achieving the flexibility requirement.

Search completeness: Search completeness is measured as the percentage of advertised objects (matching the query) that were discovered by the search. Required level of search completeness varies from application to application. A search mechanism should have guarantee on the discovery of rare objects. In the case of popular or highly replicated objects, only a predefined number of matches would suffice for most cases. For specific queries, the number of matching objects would be low and all of them should be discovered by the search. Broad queries, on the other hand, would match a large number of advertised objects. In this case search result may be restricted within a predefined limit to avoid high bandwidth consumption.

Fault-resilience: In LSDSs, participating nodes connect autonomously without administrative intervention. Nodes depart from the network without a priori notification. The search mechanism is expected to advertise and discover objects in a continu-

ously evolving overlay topology, resulting from the frequent arrival and failure of nodes. In many cases index replication and pair-wise, alternate routing paths are used to improve availability.

Load distribution: Heterogeneity in nodes' capabilities, including processing power, storage, bandwidth and uptime, is prominent in LSDSs. To avoid hot spots and to ensure efficiency, the advertisement and search mechanisms should distribute routing, storage and processing loads according to the capabilities of the participating nodes. In other words, uniform distribution of load may result into poor system performance in a LSDS.

In addition to the above mentioned design requirements, a number of other requirements of secondary importance may arise in different scenarios. For example,

- *autonomy* of index placement and routing path selection may be required for security and performance reasons;
- *anonymity* of the advertising, indexing and searching entities may be required in censorship resistance systems;
- *ranking* of search results may be required for full-text search or information retrieval systems; *etc.*

2.6 Components of a Distributed Search System

Based on the foregoing discussion of existing research works on distributed search in the three studied representative domains, we can identify the following integral components of a distributed search mechanism (see Figure 2.9).

1. **Query semantics** refer to the expressiveness of a query and the allowed level of semantic heterogeneity in queried and advertised information.

2. **Translation** is a function governing the transformation of the semantic information present in a query to a form, suitable for query routing.

3. **Routing** refers to the mechanism of forwarding a query to the nodes suitable for answering the query.

In the following sections we explain each of these components in greater detail and present the various solutions available for each of them.

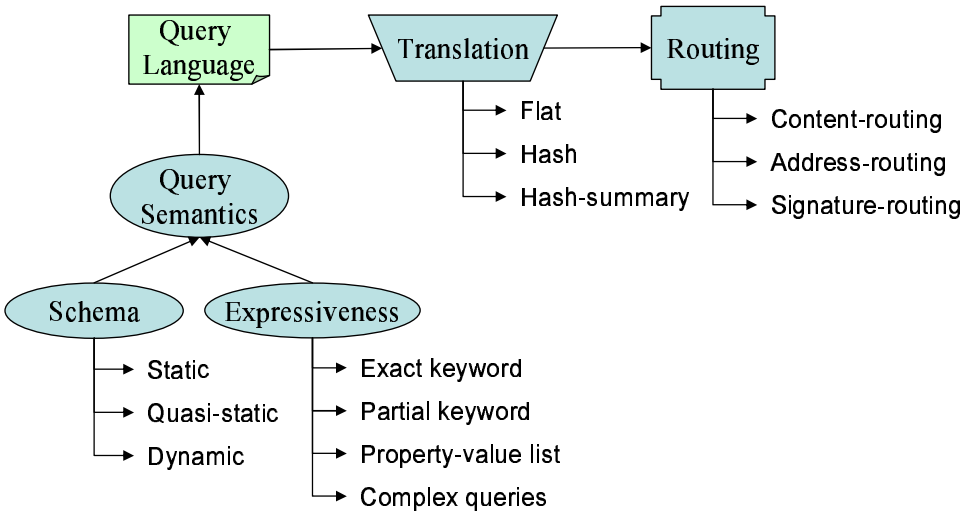


Figure 2.9: Content sharing P2P architectures

2.6.1 Query Semantics

Any visible (*e.g.*, shared or advertised) object in a distributed system is associated with a set of properties describing the behavioral and functional aspects of that object. Meta information on a set of related properties associated with a class of objects is defined as the *schema* for that class of objects. In a distributed search system, structure and scope (temporal and spatial) of the available schemas influence the query language capability and underlying routing mechanism. The rest of this section highlights two aspects of query semantics: *schema* and *query expressiveness*.

2.6.1.1 Schema

Based on the temporal and spatial scope of the schema, large scale distributed systems can be classified as follows:

Static schema : Most of the file sharing P2P systems have been designed to share one or more specific types of files, *e.g.*, song, movie, software *etc.* For each type of file a specific set of properties is defined that remain unchanged throughout the life of the system. Essentially these systems have one or more static schemas that are globally known.

Quasi-static schema : Most of the service discovery systems fall into this category. Unlike file sharing P2P systems, service discovery systems allow dynamic creation of schema for describing services. Each service instance is advertised as a Service Description governed by a predefined *Service Schema* (or template). All schemas in a given service discovery system have to contain a minimal set of predefined *properties* to comply with the specific system under consideration. Though schema can be created dynamically, the rate of such events is very low and the number of available schemas in a given system is much lower than that in PDBS. Furthermore, it is assumed that all the existing schemas in the system are globally known.

Dynamic schema : Most of the PDBSs fall into this category. In these systems heterogeneous schemas exist. Temporal scope of a schema is often bounded by the lifespan of the peer advertising data with that schema. Spatial scope is local to the originating peer and its neighbors; no global knowledge is assumed. Automating the process of semantic mapping between similar schemas is a challenging problem in such systems, which may require additional functionality from the underlying routing mechanism.

2.6.1.2 Expressiveness

Query expressiveness refers to the capability of the query language in expressing information retrieval requirements. Exiting research works focus on a wide variety of query expressiveness ranging from simple keyword-based queries to complex queries, such as LDAP filter [77] and XPath [90]. Below is a non-exhaustive list of the different levels of query expressiveness commonly found in distributed search techniques.

Exact keyword match is the minimum level of query expressiveness supported by any search mechanism, and is present in most of the file sharing P2P systems, especially the ones based on DHT techniques. For this level of expressiveness, a globally known fixed schema (with a limited number of properties) is assumed.

Partial keyword match is supported by most of the unstructured techniques as well as some extensions of DHT techniques, such as Squid [127], Twine [27] and RDF-Peers [37]. Two major variants in this category can be found. Some extensions to DHT techniques support *partial prefix matching* and unstructured techniques support true *partial matching*.

Property-value list is used by many service discovery techniques. Service Descriptions are specified as a property-value list, and queries are specified as a subset of the advertised property-value list. Most service discovery techniques assume a flat list of property-value pairs and do not support wildcard-based partial matching in property names or values.

Complex queries involve logical and relational operators (*i.e.*, *range queries*), and hierarchical relations between properties. SLP offers LDAP filter-based complex queries. INS, Twine and most PDBSs use hierarchical data structures in advertisements and allow hierarchical query expressions. Many PDBSs allow formal query languages, such as XQuery [38], XPath [90] and SPARQL [114].

2.6.2 Translation

In most distributed systems the query expression specified by a user is not used *as is* by the underlying routing mechanism. Instead, the query expression goes through some kind of transformation before it is fed to the routing process. The translation function works as a bridge between user specified queries and the routing mechanism. The *domain* of the translation function is governed by the query semantics as discussed in the previous section. The *range* of the translation function, on the other hand, depends on the routing mechanism used by the underlying overlay. Based on the particular combination of query semantics and routing mechanism, this function can exhibit a wide variation. Based on their *range*, translation functions can be broadly classified into three categories:

Flat : This type of translation functions do a very little (*e.g.*, filtering) or no change to the query expression and associated semantic information. Such functions are usually used by unstructured [4] and semi-structured routing mechanisms [44], and most of the industrial approaches to service discovery, including SLP, Jini and UPnP.

Hash : Hashing is mostly used by structured and semi-structured search mechanisms. A wide variety of hashing techniques ([27], [37] and [58]) have been proposed for distributed search systems. However, the major problem with this type of translation functions is that they loose semantic information during the hash transformation process.

Hash-summary : This type of translation enables efficient query routing while preserving query semantics. Bloom filters are the most popular means of representing hash summaries. Hash summaries are used in a few semi-structured systems, including DPMS [16], SSDS [50] and NSS [99].

2.6.3 Routing

In overlay networks, routing refers to the process of forwarding a message from a source node to a destination node. The source and the destination nodes are usually at a number of hops away from each other on the overlay. Routing algorithms in overlay networks can be broadly classified into two categories: *uninformed* and *informed*. *Uninformed* routing algorithms do not use the knowledge of query semantics or target node's address in making routing decisions at each hop. *Flooding* [4][88], *Random walk* [103] and *Iterative deepening* [103][146] are the representative algorithms in this category. Such algorithms are very inefficient in terms of the generated volume of search traffic, but the robustness is very good in highly dynamic environments. Based on the nature of the information used for next hop selection, *Informed* routing algorithms can be classified into the following three categories:

Content-routing : Content-routing algorithms utilize the semantic information, embedded in user query, for making routing decisions at each hop. Hence, the associated translation function should be from the *flat* category. Examples of such routing strategy include, selective flooding ([44]) and hint based routing ([140], [146]). Content-routing allows partial match and complex queries, but the offered query routing efficiency is low. Moreover, there exists no guarantee on search completeness or the discovery of rare objects.

Address-routing : Address routing is adopted in DHT-based structured P2P overlays, such as Chord [132], CAN [118], Pastry [121] and Kademlia [105]. Different *hash* techniques are used to transform a query into a virtual address on the overlay, and this address is used to route the query to a responsible node. Routing algorithms in this category are very efficient in terms of query routing traffic, but they are not appropriate for semantic laden search (*e.g.*, partial matching and complex queries).

Signature-routing : A number of distributed search techniques, including [99] and [50], construct a signature (usually a Bloom filter) of the target object and routes queries based on this signature. These techniques strive to combine the merits of both content-routing and address-routing strategies. Signatures retain (part of or the whole) query semantics and allow aggregation for efficient indexing. However, search completeness and robustness are not as good as that in address-routing and content routing, respectively.

In Table 2.2, Table 2.3 and Table 2.4 we identify the components of search mechanisms from three application domains: P2P content sharing, service discovery and P2P databases, respectively.

Table 2.2: Components of selected search techniques in P2P content sharing

P2P content sharing				
Ref	Name	Query	Translation	Routing
[102]	Keyword fusion	Multi-keyword	Inverted index	Chord
[85]	Joung et al.	Multi-keyword	Query superset	Chord
[138]	pSearch	Full text, multi-keyword	VSM/LSI	CAN
[127]	Squid	Prefix match	Hilbert SFC	Chord
[82]	MKey	Subset match	Query superset	Chord+Flooding
[73]	SkipNet	Prefix match	Flat	Skip List
[39]	GIA	Partial keyword	Flat	Capacity bias
[140]	APS	Partial keyword	Flat	Result bias
[99]	NSS	Multi-keyword	Bloom filter (BF)	Controlled Flood
[119]	PLR	Multi-keyword	Attenuated BF	Hint bias

Table 2.3: Components of selected search techniques in Service discovery

Service discovery				
Ref	Name	Query	Translation	Routing
[50]	SSDS	Subset/PV-list	Bloom filter	Global hierarchy
[70]	SLP	LDAP filter	Flat	Flooding
[27]	Twine	Subtree match	Stranding + hash	Chord
[100]	PWSD	XML path prefix	Stranding + hash	Chord
[128]	Schmidt et al.	Prefix match	Hilbert SFC	Chord
[125]	Schlosser et al.	Semantic match	Ontology concept → d-coord.	2-tier Hypercube

Table 2.4: Components of selected search techniques in PDBS

P2P databases				
Ref	Name	Query	Translation	Routing
[112]	PeerDB	SQL	Synonym/flat	Flooding
[75]	Humboldt Discoverer	SPARQL/RDF	URI-hash+Flat	DHT+Controlled Flooding
[31]	XP2P	XPath(absolute)	Fingerprint	Chord
[58]	Galanis et al.	XPath(relative)	XML element hash	Chord
[37]	RDFPeers	Partial RDF triple	RDF element hash	Chord

2.6.4 Relation to Other Taxonomies

The taxonomy presented in this section is based on the *routing mechanism*. Two widely used taxonomies of search techniques in content-sharing P2P systems were presented in [22]: one based on *overlay architecture* and the other based on *indexing mechanism*.

These taxonomies have been already discussed in Section 2.3.1.1, we reproduce them here for the sake of clarity.

Based on *overlay architecture*, P2P systems can be classified [22] into the following three categories:

- **Centralized**: In centralized architectures there exists a central server in the network for indexing meta-information on advertised contents and the participating peers in the systems. Typical example of a centralized architecture is the Napster [8] network.

- **Partially Decentralized**: In partially decentralized case the overlay is composed of two types of peers: superpeers and regular peers. Superpeers are responsible for indexing and query routing in addition to the activities performed by the regular peers, *i.e.*, file transfer. Typical examples in this category include KaZaA [5] and Morpheus [7].

- **Pure Decentralized**: In this category all peers have equal responsibilities, *i.e.*, indexing, query routing and file transfer. Example in this category are Gnutella [4], GIA [39] *etc.*

Based on the *indexing mechanism* and placement of indexed information P2P systems can be classified [22] into the following three categories:

- **Unstructured** techniques do not build any index and use uninformed search mechanisms, like Flooding and Random walk. Examples in this category include Gnutella [4].

- **Semi-structured** techniques build index information but do not place any restriction on index placement. Indexed information contains hints on possible location of the content. Freenet [42] and DPMS fall into this category.

- **Structured** techniques rely on some index placement rule that allows one to pinpoint the peer(s) responsible for a given index. Each peer knows the exact location of the contents it has indexed. Examples in this category include DHT techniques (*e.g.*, Chord [132], CAN [118], Kademlia [105], *etc.*), Plexus and Skipnet [73].

Finally, in this work we have classified the search mechanisms based on *query routing mechanism* into three categories: *content routing*, *signature routing* and *address routing*. These three taxonomies are orthogonal to each other. In Table 2.5, Table 2.6 and Table 2.7 we categorize representative search mechanisms based on these three taxonomies.

2.7 Summary

Based on our study of distributed search techniques in large-scale distributed systems, we found that semantic-aware flexible search is required in all three application domains considered. Signature-routing can be a good candidate for fulfilling this requirement. Keywords for content-sharing P2P systems, Service Descriptions for service discovery systems and heterogeneous schema information for P2P database systems can be hashed into Bloom-filters (see Figure 1.3). This implies that we can map these problems into DPM. Hence, an efficient solution to the DPM problem will serve as a platform for solving a number of related problems.

To the best of our knowledge, none of the proposed approaches in any of the three application domains can provide a satisfactory solution to the DPM problem. *Unstructured techniques* can be used to solve the DPM problem, but the generated search traffic does not scale with network size. *Extensions to structured techniques* support partial prefix-matching only and require a number of individual DHT-lookups for resolving a single query. *Semi-structured techniques*, on the other hand, do not provide any guarantee on search completeness or any bound on the search traffic.

Table 2.5: Taxonomy comparison: Routing mechanism Vs. overlay architecture

	Content	*Signature*	*Address*
Centralized	Napster[8]		
Partially Decentralized	SkipNet[73], SkipGraph[24]	**Plexus**, **DPMS** SSDS[50]	Humboldt Discoverer[75]
Pure Decentralized	Gnutella[4]	NSS[99], GIA[39]	DHT, Freenet[42]

Table 2.6: Taxonomy comparison: Routing mechanism Vs. indexing mechanism

	Content	*Signature*	*Address*
Unstructured	Gnutella[4]		
Semi-structured	Associative[44]	NSS[99], SSDS[50], GIA[39], **DPMS**	FreeNet[42], JXTA[34]
Structured	SkipNet[73], SkipGraph[24]	**Plexus**	DHT

Table 2.7: Taxonomy Comparison: Indexing mechanism Vs. overlay architecture[22]

	Centralized	*Partial-decentralized*	*Pure-decentralized*
Unstructured	Napster[8]	KaZaA[5]	Gnutella[4]
Semi-structured		**DPMS**	FreeNet[42]
Structured		**Plexus**	DHT

In this thesis we are interested in finding efficient solutions to the DPM problem. Besides being efficient in terms of search traffic, we want such solutions to be robust (in the presence of churn in node population) and complete (in terms of search results).

Chapter 3

DPMS: Distributed Pattern Matching System

3.1 Introduction

This chapter presents the detailed design of DPMS as a solution to the DPM problem. DPMS can enable partial and multiple keyword search for P2P content sharing networks, service discovery using partial service capability description, and XML document mining using partial path prefix for PDBS. In DPMS, advertised patterns are replicated and aggregated by the peers, organized in a lattice-like hierarchy. Replication improves availability and resilience to peer failure, and aggregation reduces storage overhead. An advertised pattern can be discovered using any subset of its 1-bits. Search complexity in DPMS is *logarithmic* to the total number of peers in the system. Advertisement overhead and guarantee on search completeness is comparable to that of DHT-based systems. We have presented mathematical models and simulation results to demonstrate the effectiveness of the DPMS architecture.

3.2 Chapter Organization

We first give an overview of the system architecture in Section 3.3. The process of obtaining advertisements and queries, from keywords and service descriptions, is illustrated in Section 3.4. Desirable properties of aggregates and the aggregation process are described in Section 3.5 and Section 3.6, respectively. Index distribution, topology maintenance and query routing policies are presented in Section 3.7, Section 3.8 and Section 3.9, respectively. Section 3.10 and Section 3.11 describe peer join and failure recovery mechanisms, respectively. We provide mathematical analysis for routing efficiency, false match probability and advertisement overhead in Section 3.12. Finally, the experimental results are presented in Section 3.13.

3.3 Architectural Overview

This section presents an overview of the DPMS architecture. In this section we will use the terms *pattern* and *index* interchangeably, as patterns are used as indices for query routing.

In DPMS a peer can act as a leaf peer or indexing peer. A leaf peer is at the bottom layer of the indexing hierarchy and advertises its indices (created from the objects it is willing to share) to other peers in the system. An indexing peer, on the other hand, stores indices from other peers (leaf peers or indexing peers). A peer can join different levels of the indexing hierarchy and can simultaneously act in both roles.

Figure 3.1 presents two logical views (replication tree and aggregation tree) of a single overlay topology. Indexing peers get arranged into a lattice-like hierarchy and disseminate index information using repeated aggregation and replication.

Index (e.g. keywords or hash keys) replication is used by many unstructured P2P systems for improving reliability and availability. But replication incurs extra overhead

in terms of storage and network bandwidth. To improve efficiency, these systems adopt smart replication strategies [45].

DPMS uses replication trees (see Figure 3.1(a)) for disseminating patterns from a leaf peer to a large number of indexing peers. However such a replication strategy will generate large volume of advertisement traffic. To overcome this shortcoming, DPMS combines replication with lossy-aggregation to minimize the volume of traffic between peers in adjacent levels in the indexing hierarchy. As shown in Figure 3.1(b), advertisements from different peers are aggregated and propagated to peers in the next level along the aggregation tree. The amount of replication and aggregation is controlled by two system-wide parameters, namely replication factor R and branching factor B. In order to achieve constant volume of exchanged messages between adjacent levels, an aggregation ratio of $R : 1$ is required.

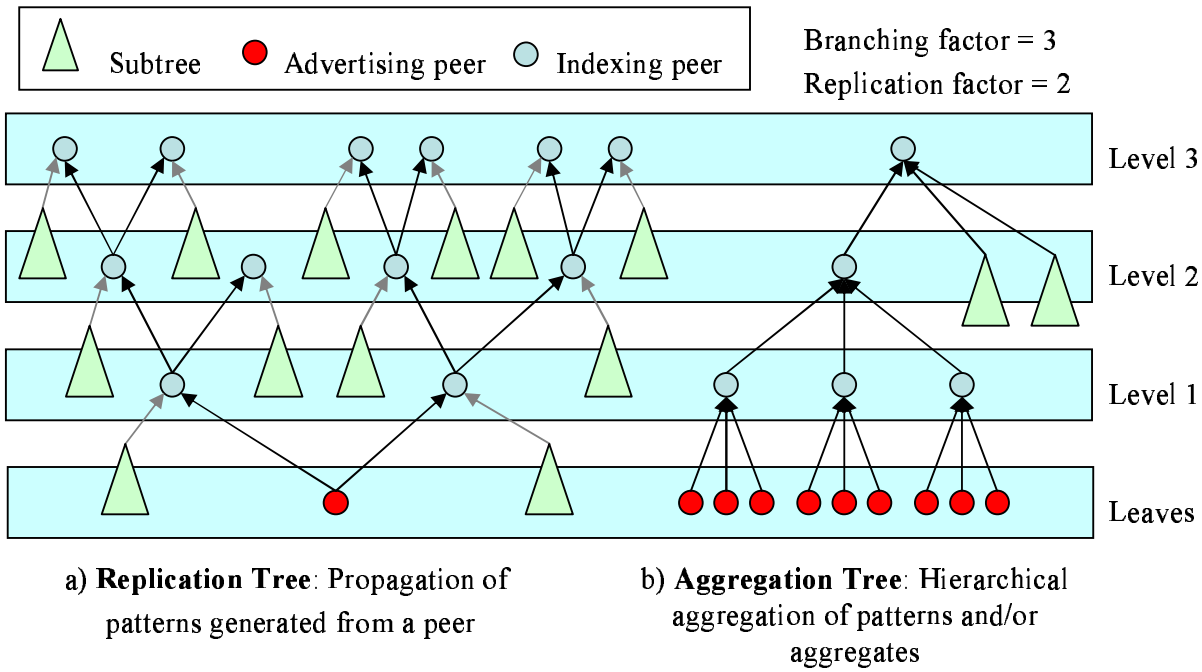


Figure 3.1: DPMS architecture overview

Patterns advertised by a leaf peer are propagated to R^l indexing peers at level l. On

the other hand, an indexing peer at level l contains patterns from B^l leaf peers. Due to repeated (lossy) aggregation, information content of the aggregates reduces as we move up along the indexing hierarchy.

The indexing hierarchy has three-fold impact on system performance. Firstly, the indexing hierarchy evenly distributes index information in the highest level indexing peers. This helps in load balancing the system and improves fault tolerance. Secondly, peers can route queries towards appropriate leaf peers without having any global knowledge of the overlay topology. Each peer needs to know the addresses of its children, replicas and one of its parents. Finally, the indexing hierarchy helps in minimizing query forwarding traffic. While forwarding a query from a root peer to multiple leaf peers in the same aggregation tree, shared path from the root peer to the common ancestor of the target leaf peers is utilized.

3.4 Index/Pattern Construction Process

In DHT-based systems an index is obtained by applying some system-wide known hash function to the keyword(s) related to a document. In DHT-techniques an index is used in two ways. First, to identify a document and second, to identify the peer responsible for that document (or a link to that document). A query consists of an index, created by hashing the search keyword. This warrants the search index to be identical to the advertised index. However, a peer can readily identify the responsible peer for a query, and route the query to that peer efficiently.

In unstructured systems, on the other hand, documents are identified using associated keywords. A query consists of one or more search keywords. Query routing is done based on Flooding or Random walk. A peer receiving a query can return a document partially matching the search keyword(s), in case an exact match was not found.

DPMS uses Bloom filters [30] as indices, to achieve the advantages of both structured

and unstructured P2P systems, *i.e.*, efficient routing and inexact matching. Bloom filters are used for set membership tests. Because of their space-efficiency, Bloom filters are used in many network applications for exchanging content summary between networked nodes [33]. However, this space-efficiency comes at the expense of a small possibility of false positives in the membership check operation.

Each document in a traditional file-sharing P2P system is associated with a set of keywords. In DPMS, all the keywords associated with a document are encoded in a single Bloom-filter. To facilitate inexact matching, each keyword is first fragmented into η-grams (usually trigrams). These η-grams are then inserted into the Bloom filter representing the document.

Query keywords are also fragmented into η-grams (see Figure 3.2) and encoded into a Bloom filter. The 1-bits on a query should be a subset of any pattern that it should match against. This kind of encoding allows us to retrieve documents, advertised with keywords *"invisible man"* and *"visible woman"* respectively, using a query containing partial keywords like *"*visi*man*"*.

For a P2P service discovery system, indices can be obtained in a similar fashion, using property-value pairs instead of keywords. Molecular fingerprint can be used as index for some envisioned distributed system storing molecular structure information.

3.5 Aggregates

DPMS relies on replication for disseminating pattern information along replication trees. Replication is necessary for load-balancing and for improving fault-tolerance. The replication strategy adopted in DPMS would significantly increase network and storage overhead. DPMS uses repeated aggregation at each level along the hierarchy to mitigate this problem.

DPMS index distribution and query routing architecture is independent of the un-

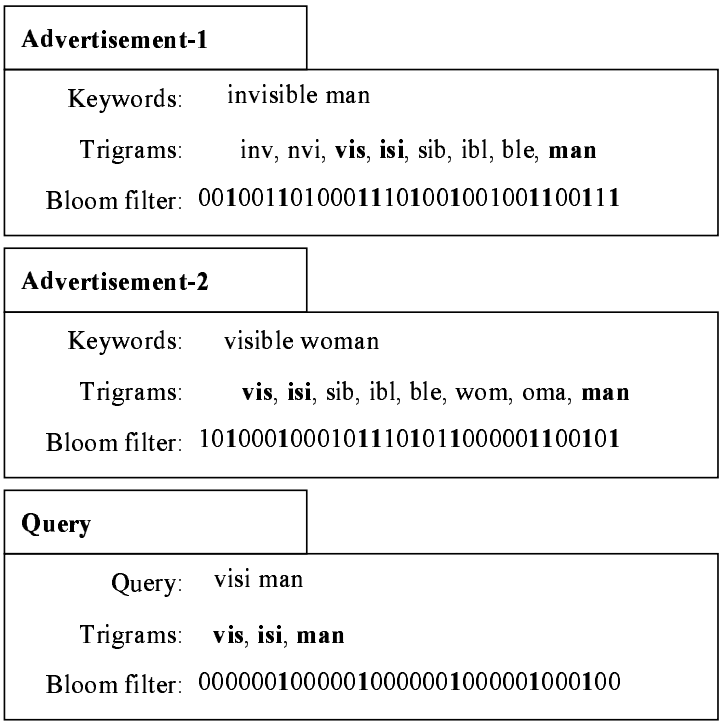


Figure 3.2: Bloom filter example of two advertisements and an inexact multi-keyword query. 1-bits in boldface corresponds to the matching trigrams.

derlying aggregation scheme. It is possible to plug-in different aggregation schemes with DPMS. However, all the peers in an instance of the system must use the same aggregation mechanism. An aggregation mechanism should have the following properties to be compatible with DPMS indexing hierarchy:

- The aggregation scheme should compress index information obtained from child peers. Lossy compression is allowed. It is preferable to have some parameter to control the level of aggregation.
- The aggregated form should retain original pattern information (to some extent), making it possible to perform pattern matching on the aggregates.
- Repeated aggregation should be possible, *i.e.*, it should be possible to perform

aggregation on aggregates without violating the pervious requirements.

A trivial way of pattern aggregation is to logically OR the bit sequences in two patterns as performed in [50] and [99]. But the information loss in this aggregation scheme is very high. More specifically, as the number of patterns contained in an aggregate increases, the number of 1-bits increases and at some point all the bits can become one, making the aggregate useless. Moreover, while matching a query with an aggregate, we cannot infer that some subset of the 1-bits in the aggregate was present in one single constituent pattern.

Considering the requirements, and the problems with bit-wise OR-based aggregation, we suggest a don't care based aggregation scheme. Don't cares (X-bits) are used to represent both 1 and 0 in the same bit position. An aggregate, say P, is defined as,

$$P = \{p_i |\ p_i \text{ is } i^{th} \text{ bit of } P \wedge p_i \in \{0, 1, X\}\}$$

We will use $\otimes$ to denote aggregation operation. The process of obtaining the bits of P, from two patterns (or aggregates), say A and B is as follows:

$$p_i = \begin{cases} a_i & \text{if } a_i = b_i \\ X & \text{otherwise, } i.e., \ a_i \neq b_i \text{ or } a_i = X \text{ or } b_i = X \end{cases}$$

This type of aggregates retains parts from the constituent patterns or aggregates. A 1-bit (or 0-bit) in such an aggregate indicates that all of the patterns contributing to this aggregate had 1 (or 0) at corresponding position. However incorporating this extra information (*i.e.*, X's) incurs some space overhead, which can be minimized (assuming X's are much higher in number than 0's or 1's) by compressing the aggregates using Huffman coding or run length encoding during transmission through the network.

A disadvantage of this aggregation scheme is that it does not allow deletion of one or more constituent patterns from an aggregate. It should also be noted that the binary

representation of three symbols (*i.e.*, 0, 1 and X) requires 2 bits. This results into wastage of the fourth state, which could be used for keeping track of the number of patterns covered by the aggregate or one of the constituent patterns *etc.* However this will decrease the compression ratio for the resulting aggregates. Compression ratio of a bit-vector can be increased by increasing the percentage of 1-bits it contains. To increase the number of 1-bits in the binary representation of an aggregate, we can encode the three symbols as '0'=01, '1'=10 and 'X'=11.

3.6 Aggregation Process

An indexing peer acts as a multiplexer in the indexing hierarchy. It gathers *in-list*s (lists of patterns or aggregates from the B child peers), aggregates them to another list (referred to as *out-list*) of aggregates, and sends this list to each of its parents.

Construction of out-list is not trivial. We want the aggregates in out-list to have the following two properties:

1. No aggregate in out-list should have more X's than a predefined limit.

2. Minimum number of X's should be introduced in the resulting aggregates.

In essence we are interested in partitioning the in-list into clusters such that the Hamming distance between the patterns within a cluster is minimal. Each partition is represented by an aggregate, which is obtained by applying the aggregation operation ($\otimes$) on the constituent patterns. Evidently the maximum Hamming distance between any two patterns in an aggregate is less than or equal to the number of X's in that aggregate.

The problem of constructing the out-list from a list of patterns is essentially the Hamming Distance Clustering (HDC) problem (or p-clustering problem). HDC can be stated as follows:

Given a positive integer p and a set $\mathscr{S}$ of θ binary strings $s_i \in \mathbb{F}_2^n$, where $i = 1 \ldots \theta$ and $\mathbb{F}_2^n$ is the binary vector space of length n, partition $\mathscr{S}$ into p disjoint subsets $s_1, s_2 \ldots s_p$ (called p-clusters of $\mathscr{S}$) so that the value of

$$\max_{1 \leq l \leq p} \max_{s_i, s_j \in \mathscr{S}_l} d(s_i, s_j) \tag{3.1}$$

is minimized.

Now we formally define the out-list construction process and explain its relation to the HDC problem.

We use $P^t (t \in \{0, 1, X\})$ to denote a mask on aggregate P. P^t has 1 in those bit positions where P has a bit equal to the value of t. *i.e.*,

$$P^t = \{p_i^t | \quad p_i^t \text{ corresponds to } i^{th} \text{ bit of } P \wedge p_i^t \in \{0, 1\} \\ \wedge p_i = t \Rightarrow p_i^t = 1\}$$

For example, if $P =$ `0X1X 101X`, then we can compute $P^1 =$ `0010 1010`, $P^0 =$ `1000 0100` and $P^X =$ `0101 0001`

It should be noted that,

$$\begin{aligned} |P^1| + |P^0| + |P^X| &= K \\ \text{where, } |P^t| &= \text{number of 1-bits in } |P^t| \\ \text{and } K &= \text{total number of bits in } P \end{aligned}$$

The formula for measuring similarity of two patterns/aggregates, say P and R is given in Equation (3.2).

$$h(P,R) = \frac{E + \alpha \times F + \beta \times G + \gamma \times H}{|P^1| + |P^0| + |P^X|} \tag{3.2}$$

where,

$$E = |P^0 \wedge R^0| + |P^1 \wedge R^1|$$

$$F = |P^X \wedge R^X|$$

$$G = |P^X \oplus R^X|$$

$$H = |P^0 \wedge R^1| + |P^1 \wedge R^0|$$

In this equation E and F define the number of positions in the aggregate that will remain the same as that of P (or R). While G and H give a measure of relative increase of X-bits in the resulting aggregate. Table 3.1 summarizes the roles of the different components in Equation (3.2).

Table 3.1: Significance of different components of Equation (3.2)

p_i	r_i	weight	component	term
0	0	1	E	$\|P^0 \wedge R^0\|$
1	1			$\|P^1 \wedge R^1\|$
X	X	α	F	$\|P^X \wedge R^X\|$
0	X			
1	X	β	G	$\|P^X \oplus R^X\|$
X	0			
X	1			
0	1	γ	H	$\|P^0 \wedge R^1\|+$
1	0			$\|P^1 \wedge R^0\|$

Values for the constants, α, β and γ depend on the nature of the patterns. If the

patterns are random without any correlation then the value of α, β and γ should be increased, which encourages the construction of aggregates with higher number of X's. On the other hand, if the patterns exhibit correlation or bias[1], then the values of these constants should be reduced as it would be possible to construct aggregates with lower number of X's.

To establish the relation between $h(P, R)$ and $d(P, R)$ we set $\alpha = 1$ and $\gamma = \beta$. The reason behind this can be deduced from Table 3.1. For aggregates P and R, the number of positions at which they agree is $(E+F)$ and at which they differ is $(G+H) = K-(E+F)$. By putting these values to Equation (3.2) we get

$$h(P, R) = \frac{1}{K}\left(K - (1-\beta)d(P, R)\right) \tag{3.3}$$

Hence the equivalent of Equation (3.1) would be to maximize

$$\min_{\mathcal{A} \in out-list} \min_{P,R \in \mathcal{A}} h(P, R) \tag{3.4}$$

A 2-approximation solution to the HDC problem can be found in [62]. However, we cannot adopt this algorithm *as is* because of two reasons. The algorithm in [62]

1. assumes binary patterns, whereas we are using a ternary system.

2. assumes patterns in the input list; on the contrary, the in-list in our case may contain aggregates (*i.e.*, clusters) and/or patterns.

We have used a modified version of that algorithm as presented in Algorithm 1. It takes three input parameters:

- **In-list (**$Pattern[\,]$**)** is an array of patterns or aggregates constructed from the B in-lists received from the B children.

[1]By correlation or bias of patterns we mean that the distribution of the patterns in hamming space is not uniform, instead patterns are clustered with non-uniform density distribution.

- **Minimum non-X bits (O)** is the minimum number of original (*i.e.*, non-X) bits an aggregate *must* retain after aggregation.

- **Target aggregation ratio (A)** is the ratio of the number of aggregates in the out-list to the number of patterns/aggregates in the in-lists. The aim is to achieve an aggregation ratio of A without violating the constraint imposed by O.

This algorithm starts with the initial set of patterns or aggregates as obtained from the in-list and tries to reduce the number of clusters (*i.e.*, aggregates) by one at each iteration until the target aggregation level (specified by A) is reached or the restriction on the minimum number of non-X bits imposed by O is violated. At each iteration the aim is to find the pair of patterns or aggregates which minimizes the similarity function $h(\cdot)$ in Equation (3.2) and replaces the pair with the newly computed aggregate.

Parameter O passed to Algorithm 1 controls the amplitude of loss incurred by the aggregation process. Instead of using a fixed value for O, we varied it linearly as a function of the level of the peer at which the algorithm is executed. A peer at level l executes the aggregation algorithm with $O = O_{sys}\frac{l}{L}$, where O_{sys} is the system wide lower limit for O and L is the maximum hight of the indexing hierarchy. The implication of this design choice is two fold:

- It promotes reduced aggregation loss at lower levels of the indexing hierarchy, though at an expense of increased size of out-list. This reduces the unwanted search traffic generated by false matches.

- Better quality aggregates (*i.e.*, covering larger number of patterns without introducing higher number of X's) can be produced in higher level peers.

Computing the runtime complexity of Algorithm 1 is straight forward. Let, $|inList| = \varrho$ and $A' = (1 - A)$. The loop in lines 10-17 will be executed $A'\varrho$ times. It is possible

Algorithm 1 Aggregate a list of Patterns

1: Input:
2: $inList : Pattern[\,]$
3: $O : Integer$
4: $A : Float$
5: Output:
6: $outList : Pattern[\,]$
7: Global:
8: $H(P, R)$ {see Equation (3.2)}
9: $outList \leftarrow inList$
10: **while** $|outList| > A \times |inList|$ **do**
11: find $P_r \in outList$ and $R_r \in outList$ such that
$(P_r \neq R_r) \wedge$
$(|(P_r \otimes R_r)^0| + |(P_r \otimes R_r)^1| \geq O) \wedge$
$(H(P_r, R_r) \geq H(P, R)\ \forall P, Q \in outList)$
12: **if** no such P_r and R_r exists **then**
13: break {failed to achieve target aggregation ratio}
14: **end if**
15: $P_{new} \leftarrow P_r \otimes R_r$
16: $outList \leftarrow \{outList - \{P_r, R_r\}\} \cup \{P_{new}\}$
17: **end while**
18: **return** $outList$

to execute line 11 in $(\varrho - i - 1)$ cost for the i^{th} iteration by keeping track of the closest aggregate for each element in out-list. Hence the complexity can be calculated as,

$$\sum_{i=1}^{A'\varrho}(\varrho - i - 1) = \varrho(A'\varrho + 1)(1 - \frac{A'}{2}) = O(\varrho^2).$$

3.7 Index Distribution

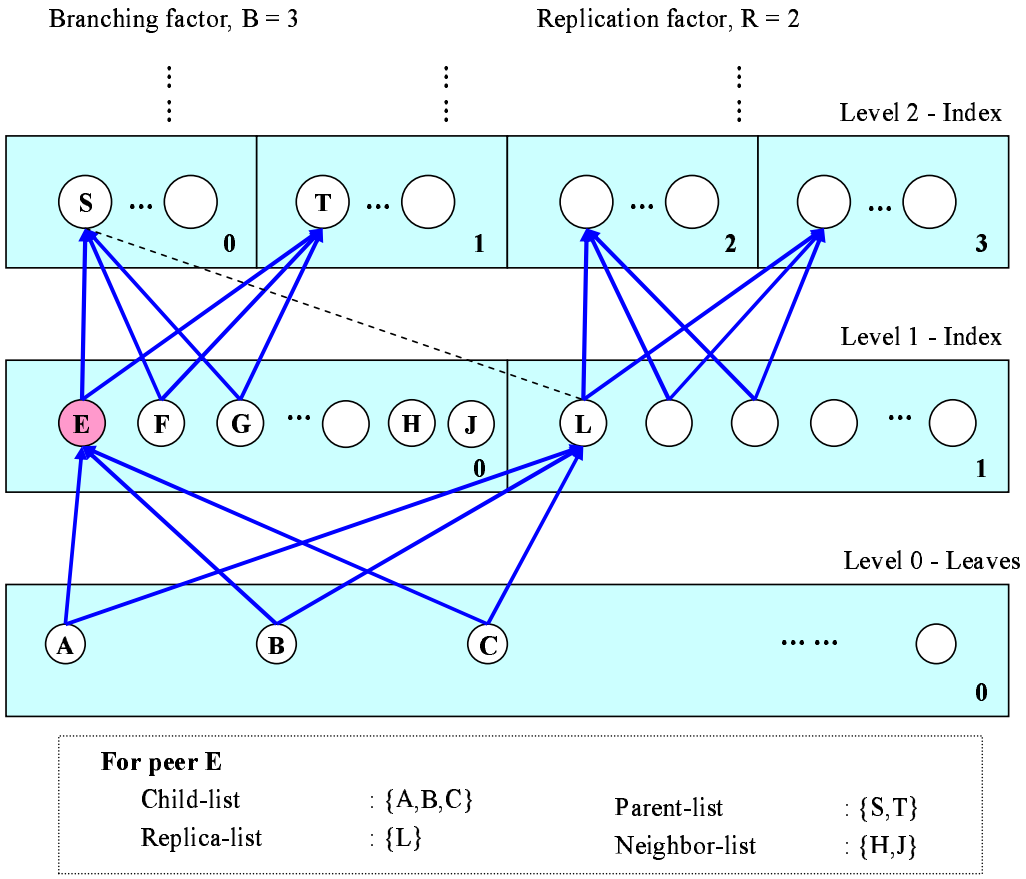


Figure 3.3: Index distribution architecture. All the peers interacting with peer E are labelled. Group number is printed at the bottom right corner of each box.

An indexing peer, participating in the DPMS architecture, belongs to two sets, vertical (*i.e.*, level) and horizontal (*i.e.*, group). According to the extent of aggregation, each

indexing peer belongs to a level (vertical set). Peers participating at higher index levels cover (*i.e.*, contain index information from) a higher number of leaf peers along the aggregation tree. But, due to increased loss from repeated aggregation the contained information gets vaguer at higher level peers.

Indexing peers in level l arrange into R^l groups (horizontal sets), numbered from 0 to $(R^l - 1)$ (see Figure 3.3). In the ideal case, all the indexing peers in a single group (at any level) should collectively cover all the leaf peers in the system.

A peer at level l and group g $(0 \leq g < R^l)$ is responsible for transmitting its aggregate information to R parents at level $(l + 1)$. Each parent belongs to a different group, in range $[g \times R, (g + 1) \times R)$, respectively.

Peers at level l and group g organize into subgroups (referred to as siblings) of size B to forward their aggregate information to the same set of parents. Thus each group in range $[g \times R, (g + 1) \times R)$ at level $(l + 1)$ will contain a peer replicating the same index information. This provides redundant routing paths for query forwarding and increases tolerance to peer failure.

3.8 Topology Maintenance

In the DPMS index distribution hierarchy, peers interact with each other in different roles, e.g. parent, child, neighbor etc. An indexing peer, say E, at level l and group g, maintains four separate lists for this purpose (see Figure 3.3).

1. **Replica-list** contains the list of peers in the adjacent groups that have common children as that of E. This list contains $(R - 1)$ peers, one from each group in range $[\lfloor g/R \rfloor \times R, (\lfloor g/R \rfloor + 1) \times R)$, excluding g.

2. **Parent-list** is the replica list obtained from one of E's parents. E uses this list to forward its aggregate information (out-list) to all of its parents along a replication

tree.

3. **Child-list** contains the list of all children and the replica list for each of them. A peer normally communicates with the child peers only. But in case of a failure of a child it can communicate with a replica of the failed child. This list contains B entries corresponding to the B children of E at level $(l-1)$ and group g/R.

4. **Neighbor-list** contains a fixed number of non-sibling peers that are in the same group (g) as peer E. This list is mostly used for maintaining connectivity in a group, during join operation, and for flooding queries horizontally within a group (mostly at the topmost level).

Out of these four lists a peer needs to keep track of three: child-list, parent-list and neighbor-list. The replica-list of a peer is the parent-list of any of its children. Peers use the Newscast protocol [143] for maintaining and updating these lists, *i.e.*, to detect peer failures and arrival of new peers. Flow of news packets is restricted to $2 \times R$ groups of peers. More specifically, the news-list of a peer at level l and group g will contain information about some peers from groups $[\lfloor g/R \rfloor \times R, (\lfloor g/R \rfloor + 1) \times R)$ at level l and groups $[g \times R, (g+1) \times R)$ at level $(l+1)$. That is, each peer sends news packets to its parents, neighbors and replicas, and receives news packets from its children, neighbors and replicas.

Unlike indexing peers, a leaf peer maintains only the neighbor-list and forwards this list to its parents. A leaf peer obtains its parent-list from one of its parents. It should be noted that leaf peers do not have any replica-list or child-list.

3.9 Query Routing

A query can be initiated by any peer in the system. The query life-cycle can be divided into three phases: ascending phase, blind search phase and descending phase.

During the *ascending phase*, an initiating peer, say I, first checks its local information to find a match. If a match is found, then the query is forwarded to the matching peer, otherwise it is forwarded to any of its parents. The parent peer performs the same operation. Thus the query either hits a peer with a match or it reaches a highest level peer, without any parents. In the first case, the query enters the descending phase, and in the later case it enters the blind search phase of query life cycle.

The *Blind search phase* is executed by a highest level peer, say E (without any parent node) upon receiving a query that does not match any aggregate in its routing table. E has no option but to blindly forward the message to some other peer in its group. As the query traverses a peer at level l, aggregates from R^l leaf peers are being checked. If a match is found, then the query if forwarded to the associated child peer and the query enters into the descending phase of its life cycle. If no peer in a group at the highest level has a match for the query pattern, then the system concludes that the search was for a non-existent advertisement pattern, and the search process terminates.

A query enters into the *descending phase* when it hits a peer that has some aggregate matching the query. The query is then forwarded to the child peer advertising the matching aggregate. This process recurs until the query reaches the leaf peer advertising the pattern.

Two types of complications may arise during the descending phase. Firstly, a peer may have multiple aggregates (from different child peers) matching a query. Secondly, a false match case may occur, *i.e.*, a peer may receive a query (from its parent) that does not match an aggregate in any of the in-lists. For the first case, the peer may forward the query to multiple peers, or any one peer based on some predefined policy and application-specific requirements. A possible scale for measuring the quality of a match is $\frac{|P^1 \wedge Q^1| + |P^0 \wedge Q^0|}{K}$, *i.e.*, the proportion of exact-match of a query Q with an aggregate P. For the second case, the search branch terminates.

Query forwarding can be done recursively, or iteratively by the initiator of the query. The iterative method is preferred for two reasons. First, the number of simultaneous searches can easily be controlled, especially in case of multiple matches during descending phase. Second, search termination criteria (like maximum number of hops, or total number of nodes probed, or total number of results etc.) can be handled more flexibly than with the recursive approach.

It is possible to achieve proximity sensitive routing in DPMS by arranging nearby peers in the network in the same aggregation tree. During the ascending phase of a query, it will be forwarded to the indexing peers common to the nearby (in network distance) leaf peers. This will automatically influence the query routing mechanism to select the nearest leaf peer with a matching pattern. Besides, the ascending phase of a query life cycle helps in balancing query load.

3.10 Node Join

A peer can join the system as a leaf peer or an indexing peer or both. To join as a leaf, a peer say C, has to find a level 1 indexing peer, say P, with an empty slot in its child-list. C joins the indexing hierarchy as a child of P. C constructs its parent list using the replica-list of P, and starts advertising its patterns to all of its parents. If C fails to find a level 1 peer with an empty slot, then it can either join in both level 1 and level 0 or select a level 1 peer with smaller number of children.

To join the indexing hierarchy as an indexing peer, a peer say E has to go through the following steps:

- **Choose level and group**: Peer E has to choose a level, say l, in the hierarchy. Selection of level can be based on the node's capacity, uptime distribution etc. Peers with higher capacity (storage and bandwidth) and longer life-time are expected to

join higher levels in the indexing hierarchy. Then peer E can choose a group g in random such that g is in $[0, R^l)$.

- **Construct child-list**: Joining Peer E has to contact a seed peer to get information about other peers in the system. Peer E can crawl the indexing hierarchy to reach a peer, say A, such that peer A is in level $(l-1)$ and in group $\lfloor g/R \rfloor$, and the parent-list of peer A contains less than R entries. Peer E can join as a parent of peer A. Peer E has to join the group in which peer A has no parents. Peer E has to obtain and update the replica-list of other parents of peer A. Peer E can obtain the child-list from a parent of peer A. Peer A can have an empty parent-list during the initialization phase of the system or after a failure of all of its parents. If peer A returns an empty parent-list, then peer E should look for other (up to B) peers, in the same group as that of peer A, with empty parent-list. If such peers exist then peer E should make them its children. Peer E may fail to find a suitable child (peer A) for two reasons (see line 7-13 in Algorithm 2): (a) if there is no peer in level $(l-1)$ and group $\lfloor g/R \rfloor$ then peer E should try to join level $(l-1)$, (b) otherwise group $\lfloor g/R \rfloor$ in level $(l-1)$ is saturated and peer E should join level $(l+1)$.

- **Construct parent-list**: To construct the parent-list peer E has to find a peer, say T, such that peer T is in level $(l+1)$ and in group $(g \times R)$, and T has an empty slot in its child-list. If such a peer (T) exists then peer E constructs its parent list using peer T and all the replicas of peer T. Otherwise, peer E will start with an empty parent-list, and will wait for more peers to join at level $(l+1)$.

The psuedocode for join process for an indexing peer is given in Algorithm 2.

Algorithm 2 Join process for indexing peer E

1: Input:
2: $l : Integer$ {Level at which to join}
3: Local:
4: $g : Integer$ {Group at which to join}
5: $g \leftarrow$ random number in range $[0, R^l)$
6: $A \leftarrow$ a peer at $< l-1, \lfloor g/R \rfloor >$ with less than R parents
7: **if** no such A exists **then**
8: **if** group $\lfloor g/R \rfloor$ is saturated **then**
9: join$(l+1)$
10: **else**
11: join$(l-1)$
12: **end if**
13: **else**
14: Join as a parent of A
$a)$ $E.replicaList = A.parentList$
$b)$ Add E to the $replicaList$ of each parent of A
$b)$ Add E to the $parentList$ of each sibling of A and A itself.
15: $T \leftarrow$ a peer at $< (l+1), (G \times R) >$ with less than B children
16: **if** no such T exists **then**
17: $E.parentList \leftarrow empty$
18: **else**
19: Join as child of T
$a)$ $E.parentList \leftarrow T.replicaList$
$b)$ Add E to the $childList$ of T and each peer in $T.replicaList$.
20: **end if**
21: **end if**

3.11 Node Leave or Failure

In DPMS, peer departure and failure are handled in the same manner, *i.e.*, a peer can leave the system without any notice. The absence of an indexing peer, say E, will affect the peers in its parent-list, child-list and replica-list. Parents and children of E can still communicate through any of the replicas of peer E. So query routing is not hampered until all of the replicas of a peer fail.

Failure or departure of a leaf peer has greater impact on the system. All the index information along the replication tree, rooted at the failed leaf peer, has to be updated. During this period (from the point of failure to the update of all indexing peers in the replication tree) a query directed towards the failed leaf peer will be evaluated as a false match. This will increase search overhead to some extent, as will be discussed in Section 3.13.3. Unlike tightly coupled distributed systems, we allow temporary inconsistency in routing tables due to peer failure. This relaxation will allow the system to efficiently deal with intermittent connectivity of peers.

To efficiently deal with frequent join and leave of a leaf peer, indexing peers should advertise their index information at constant intervals. Any advertisement from a child peer should be delayed until the end of the interval. The interval length can be used to tradeoff index update delay with network overhead due to frequent advertisements. By increasing this interval length we can reduce network overhead due to advertisement traffic but at the cost of increased update latency, and vice versa.

3.12 Analysis

3.12.1 Query Routing Efficiency

In this section we will provide an analytical bound on the levels of indexing hierarchy that will allow query routing in $O(\log N)$ hops, where N is the total number of peers in

the system. For this analysis we assume a fully grown indexing hierarchy with no peers failure. We will use B to denote branching factor, R for replication factor, and n_l as the number of peers at level l. Leaf peers reside at level 0 and the height (or maximum level) of the indexing hierarchy is h. Assuming these definitions we can calculate the maximum number of peers at level l as:

$$n_l = n_0 \left(\frac{R}{B}\right)^l = n_0 \alpha^l \tag{3.5}$$

and the total number of peers in the system as:

$$N = \sum_{l=0}^{h} n_l = n_0 \frac{1-\alpha^{h+1}}{1-\alpha} \tag{3.6}$$

Hence the total number of leaf peers in the system,

$$n_0 = N \frac{1-\alpha}{1-\alpha^{h+1}} \tag{3.7}$$

Now, the number of groups at level l is R^l. So, the average number of peers in a group at level l, say m_l, is:

$$m_l = \frac{n_l}{R^l} \tag{3.8}$$

Combining Equation (3.5), Equation (3.6) and Equation (3.8), and using $\alpha = \frac{R}{B}$ we obtain:

$$m_l = \frac{N(1-\alpha)}{B^l(1-\alpha^{h+1})} \tag{3.9}$$

For efficient query routing we expect the number of peers in a group at level h to be $f \times \log N$, where $0.5 \leq f \leq 2$. Replacing $m_h = f \times \log N$ in Equation (3.9) we obtain:

$$B^h(1-\alpha^{h+1}) = \frac{N(1-\alpha)}{f \log N} \tag{3.10}$$

For practical values of R and B, as discussed in Section 3.12.4, we can approximate $(1-\alpha^{h+1})$ with θ, where $0 < \theta < 1$. Replacing this value in Equation (3.10) we can approximate h (for $R \neq B$ and $R > 1$) as

$$h \approx \frac{1}{\log B} \times \log \frac{N(1-\alpha)}{\theta f \log N} = O\left(\log \frac{N}{\log N}\right). \tag{3.11}$$

Thus we claim that if we can build and maintain an indexing hierarchy of height $O\left(\log \frac{N}{\log N}\right)$ then we will be able to solve the DPM problem in

$$O\left(\log N + \xi\kappa \log \frac{N}{\log N}\right) \tag{3.12}$$

time. Here, $O(\log N)$ is the cost of flooding one of the R^h groups at level h, and $O(\xi\kappa \log \frac{N}{\log N})$ is the cost of reaching the κ matching leaf peers along the indexing hierarchy of height $O(\log \frac{N}{\log N})$. ξ accounts for the lossy aggregation scheme. For a system without any aggregation (*i.e.*, information loss) the value of ξ should equal one. Section 3.12.3 presents an estimate of ξ.

3.12.2 False Match Probability

In this section we present a mathematical model for predicting the false match probability for the don't care based aggregation scheme. In don't care based aggregation scheme, we consider an aggregate D to be a match for a query Q if the set of 1-bits in Q is a subset of the 1-bits in D, assuming the don't care positions in D to be 1s. This assumption leads to the possibility of *false match*, where an aggregate can match a query, although none of the constituent patterns is a match for the query. For measuring the false match probability we will assume that the patterns (advertised by the leaf peers) are Bloom-filters, with parameters m, ω, and $\hbar$ (see Section 2.3.2). Assuming the hash functions are perfectly random[2], the probability that a specific bit is 1, after all of the ω-elements have been inserted into the Bloom filter, is:

$$p = 1 - \left(1 - \frac{1}{m}\right)^{\hbar\omega} \tag{3.13}$$

[2]A hash function is perfectly random if the hashed value is uniformly distributed over the range, in this case $[0, m-1]$

The probability of false match depends on the amount of aggregation, which in turn increases as we move up along the indexing hierarchy. We can estimate the probability (ϕ_l) that an aggregate D (at level l) is a false match for a query Q as:

$$\phi_l = (1 - p^{\tau_l})^{y_l} \tag{3.14}$$

Here, y_l is the average number of patterns contained in D, and $\tau_l = |Q^1 \wedge D^X|$. Hence, p^{τ_l} is the probability that all of the τ_l 1-bits of Q are present in a pattern contained in D, and ϕ_l stands for the probability that none of the constituent patterns in D, has all of the τ_l 1-bits of Q.

We can estimate y_l as $y_l = \frac{1}{A^l}$ and τ_l as $\tau_l = \tau \left(\frac{m-O}{m}\right) \left(\frac{l}{h}\right)$. Here A and O are as defined in Section 3.6. $\tau = \chi pm$ is the expected number of ones in Q, and χ is the percentage of elements (*i.e.*, ω) from a pattern hashed into Q. As discussed in Section 3.6, $\left(\frac{m-O}{m}\right) \left(\frac{l}{h}\right)$ is the proportion of don't care bits in an aggregate at level l, considering that the number of don't care bits increases linearly as we move up along the indexing hierarchy.

For bitwise-OR based aggregation scheme, we can estimate false match probability as,

$$\psi_l = (1 - p^{\tau})^{y_l} \tag{3.15}$$

Comparing Equation (3.14) and Equation (3.15), we can infer that $\phi_l < \psi_l$ as $\tau_l < \tau$.

3.12.3 An Estimate of ξ

The don't care based lossy aggregation scheme introduces chances of "false matches" during the query routing process. As introduced in Equation (3.12), ξ represents the factor by which query routing overhead increases in presence of aggregation. This section presents an estimate of ξ. For a system without aggregation (*i.e.*, information loss) the value of ξ should equal one. In presence of aggregation the value of ξ will be greater than one. Hence an estimate of ξ can be used as a measure of the impact of aggregation on routing efficiency.

Let ν be the probability that a query will fail to match any aggregate at level l, though it matched an aggregate at level $(l+1)$. Then a query will fail to match an aggregate at level l with probability $(1-\nu)^{h-l}\nu$ and in that case $(h-l)$ hops will be wasted. Hence the expected number of probes in a complete descending phase (*i.e.*, from level h to level 0) is:

$$\xi h = h + \sum_{l=0}^{h-1}(h-l)(1-\nu)^{h-l}\nu$$

Applying equality $\sum_{t=0}^{n} tx^t = \frac{x-(n+1)x^{n+1}+nx^{n+2}}{(1-x)^2}$ yields:

$$\xi = 1 + \frac{1}{\nu h}\left[1-\nu-(h+1)(1-\nu)^{h+1}+h(1-\nu)^{h+2}\right] \tag{3.16}$$

An estimate of ν is, $\nu = \left(1-p^{\Delta\tau}\right)^{y}$. Here, $\Delta\tau = \tau_{l+1} - \tau_l = \frac{p\chi(m-O)}{h}$ and $y = \frac{1}{A}$. y is the average number of aggregates from level l that are fused to form an aggregate at level $(l+1)$.

3.12.4 Advertisement Overhead

In this section we compare the advertisement overhead in DPMS against that in DHT-based systems. For DPMS, number of advertisement messages in one refresh interval is $C_{DPMS} = \sum_{l=0}^{h-1} Rn_i = RN\frac{\alpha^h-1}{\alpha^{h+1}-1}$

Let $\Re$ be the total number of advertised patterns in the system. In DPMS $\Re$ can be computed as $\Re = Pn_0 = PN\frac{\alpha-1}{\alpha^{h+1}-1}$. Here, P is the average number of patterns advertised by a leaf peer.[3] Now if this same number of patterns (*i.e.*, $\Re$) are advertised in a DHT-based system, then we can estimate the number of advertisement messages as $C_{DHT} = \Re \lg N = PN\frac{\alpha-1}{\alpha^{h+1}-1}\lg N$. Hence, the ratio of message count in these two

[3]Unlike DHT techniques, in DPMS a peer can transmit all of its indices in a single message to a parent. Hence, the number of advertisement messages generated in DPMS does not depend on the number of pattern being advertised.

systems is:

$$MC = \frac{C_{DPMS}}{C_{DHT}} = \frac{R(\alpha^h - 1)}{P(\alpha - 1)\lg N} \tag{3.17}$$

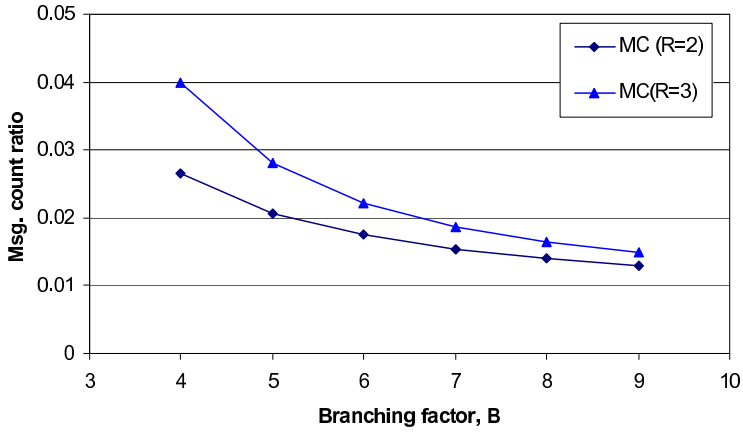
Assuming W to be the width of a pattern, we can estimate total advertisement volume for DHT-based systems as, $V_{DHT} = \Re W \lg N$. On the other hand, total advertisement volume in DPMS is, $V_{DPMS} = \sum_{l=0}^{h-1} PW(BA)^l n_l R = \Re W R \frac{(RA)^h - 1}{RA - 1}$. Hence, the ratio of message volume in these two systems is,

$$MV = \frac{V_{DPMS}}{V_{DHT}} = \frac{R\left((RA)^h - 1\right)}{(RA - 1)\lg N}. \tag{3.18}$$

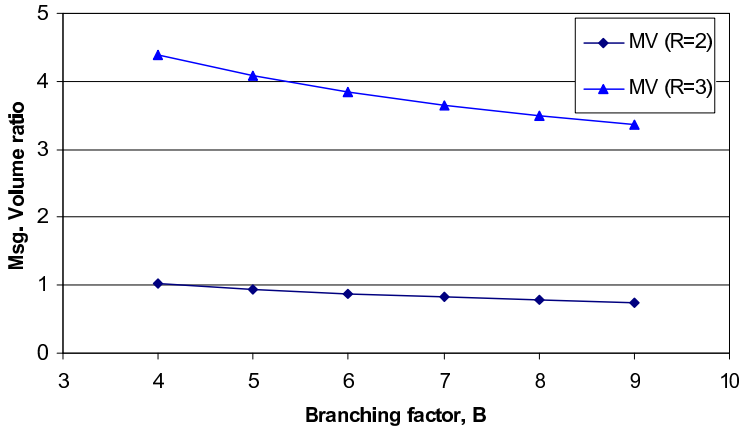
In Figure 3.4(a) and Figure 3.4(b) we produce plots of MC and MV, respectively, for varying branching factor B. For these plots we have used $A = 0.6$, $h = 5$, $P = 10$ and $R = 2, 3$. These are the parameter settings used in the experiments in Section 3.13 as well. To account for the varying number of peers in the system we have varied B from 4 to 9, which corresponds to a population of $24,000$ to $910,000$ peers for $R = 2$ and $40,000$ to $1,061,000$ thousand peers for $R = 3$. From Figure 3.4 we can infer that, advertisement message count in DPMS is much lower than that in DHT-based systems. Advertisement message volume, on the other hand, in DPMS is comparable to that of DHT-based systems for $R = 2$, though it is about 4 times higher for $R = 3$.

3.13 Experimental Evaluation

To measure the performance of the proposed system and to validate the concepts presented in this chapter, we have developed a prototype implementation of the DPMS protocol and run simulations with various parameter settings. This prototype implementation has been developed using Java language and is based on the PeerSim [10] simulator framework. PeerSim is an open source P2P simulator that allows cycle driven simulation. We shall explain various components of the PeerSim simulator in Section 5.3.2.



(a) Message count



(b) Message volume

Figure 3.4: Advertisement overhead

Existence of bias (or correlation) among the advertised patterns allows a higher level of aggregation (due to the similarity in patterns) without introducing significant number of don't cares. This results into better query routing performance at reduced storage overhead, than the case with randomly generated patterns. To justify this insight, we have conducted experiments for two cases: a) **random case**: patterns are randomly

generated bit strings, and b) **biased case**: patterns are Bloom-filters generated using 3-grams from <*song title, artist*> tuples. These tuples are chosen randomly from a database of song information extracted from *http://www.leoslyrics.com/*, which is an online database of more than 200,000 song lyrics. The bias is introduced from the non-uniform frequency distribution of different 3-grams that may occur in a keyword.

A query in the random case is created by randomly taking 33% of the 1-bits from a randomly chosen advertised pattern. While in the biased case, a query is created as a Bloom-filter constructed from 33% of the advertised 3-grams from a randomly chosen advertisement. Each data point presented in this chapter has been calculated as an average of the statistical values obtained from independent simulation runs and 3000 queries per simulation run.

In Section 3.13.1, we identify the system parameters and performance metrics, and investigate the impact of various system parameters on these performance metrics. Then we focus on the scaling behavior of the system with network size, in Section 3.13.2. Finally, in Section 3.13.3 we illustrate the impact of replication on the system's resilience to peer failure.

3.13.1 Parameter Tuning

The aggregation process introduces the possibility of trading off query routing efficiency with index storage size at peers. A higher level of aggregation results into a lower level of storage overhead, a higher level of information loss in the aggregates and a decrease in query routing efficiency, and vice versa. In this section we intend to find a balance between these two conflicting interests.

Table 3.2 summaries the system parameters and their value(s) used for parameter tuning. The first four of these parameters define the structure of the indexing hierarchy. Experiments in this section are dedicated to the analysis of the impact of different system

parameters on query routing performance and storage overhead. Hence, we have chosen R=1. This allows us to separate routing performance characteristics of the system from fault-tolerance behavior.

The last three parameters in Table 3.2 influence query routing efficiency. O and A are the parameters passed to the aggregation algorithm (see Algorithm 1). The impact of parameter W (pattern width) on query routing performance and level of aggregation is intuitive though not trivial. Experimental results demonstrate that, for a fixed value of O, increase in W increases query routing accuracy while decreasing index storage requirement at peers.

Table 3.2: System parameters and their values used for the parameter tuning experiments

Param.	**Value(s)**	**Description**
B	4	Branching factor
R	1	Replication factor
H	4	Maximum level
N	$\sim$ 4000	Number of peers in the system
P	10	Number of patterns advertised by a leaf peer.
A	0.6	Target aggregation ratio
O	10, 20...60	Min. number of non-X bits in an aggregate
W	80, 100...200	Pattern or aggregate width (m, if Bloom filter is used)

The performance metrics analyzed in this section are list below:

- **First-hit probes** is the number of peers that are probed before the first match is found.
- **Avg. probes/hit** is the average number of probes required for each hit. In cases

where multiple matches are present, we have traced up to 20 matches. Popular search engines, including *google*, return 10 results per page by default and a very little percentage of users browse beyond the first page of results. Thus we can assume that 20 matches in a search result will be sufficient for most cases.

- **Indexing overhead (IO)** is an indicator for the extra storage space requirement introduced by the indexing hierarchy. IO is measured as the ratio of the total number of aggregates in the system to the total number of patterns advertised by the leaf peers. Mathematically,

$$IO = \frac{\text{no. of aggregates (at the inexing peers)}}{\text{no. of patterns (at the leaf peers)}}. \tag{3.19}$$

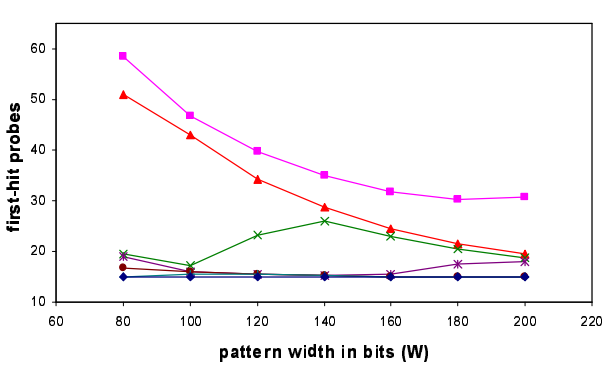
- **Effectiveness of aggregation (EA)** quantifies the amount of reduction in index storage requirement, achieved with the aggregation mechanism. The following equation is used to measure this quantity:

$$EA = 1 - \frac{\text{no. of aggregates with aggregation}}{\text{no. of aggregates without aggregation}} \tag{3.20}$$

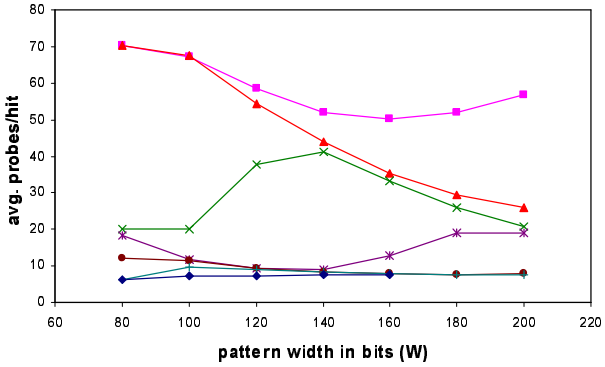
Figure 3.5(a) and Figure 3.5(b) show the impact of O and W on query routing accuracy. In turn, Figure 3.5(c) and Figure 3.5(d) show the impact of O and W on storage overhead. By analyzing the curves in these figures, we can infer the followings.

- Query routing accuracy increases with increase in the minimum number of non-X bits (O) in aggregates. This effect is intuitive. With higher number of original bits we have more information and lower probability of false matches.

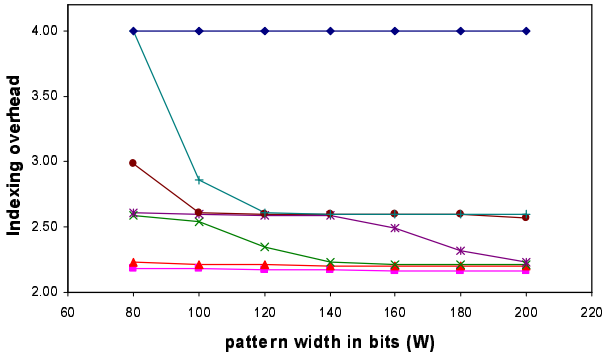
- Indexing overhead increases with an increase in O. An increase in O means lower number of don't care bits are allowed in the aggregates, which implies less space for aggregation. The result is a higher number of aggregates in the system. This justifies the previous observation as well.



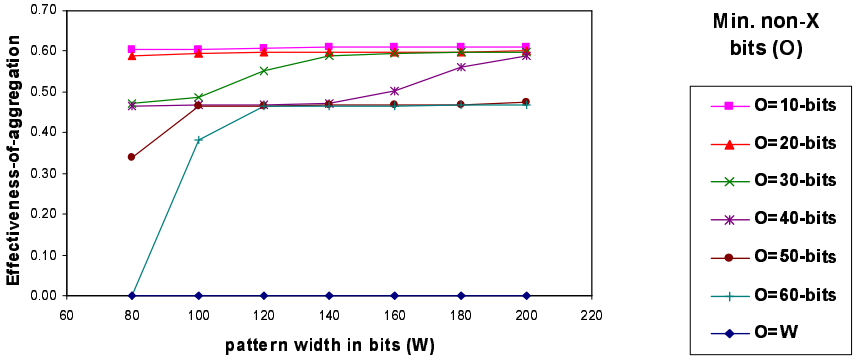
(a) First hit probes



(b) Avg. probes per hit

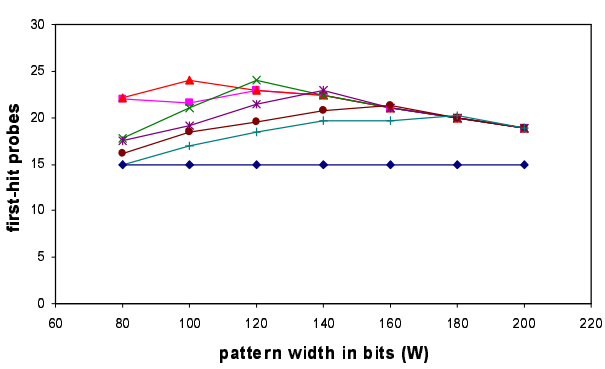


(c) Indexing overhead

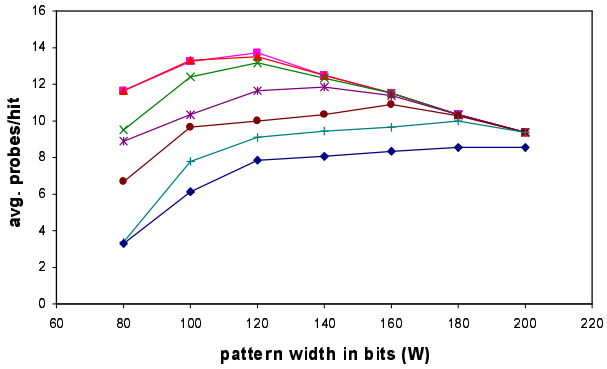


(d) Effectiveness of aggregation

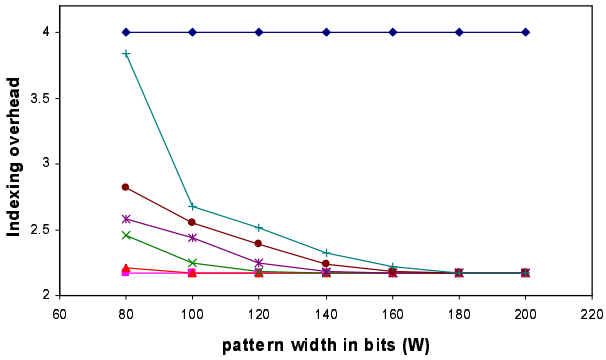
Figure 3.5: Impact of O and W on network and storage overhead (random case)



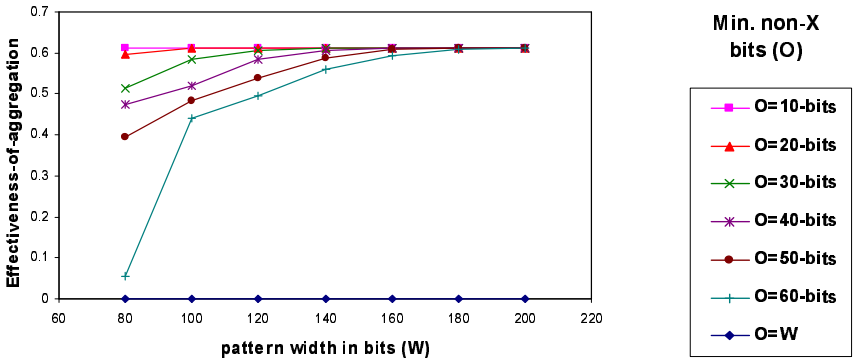
(a) First hit probes



(b) Avg. probes per hit



(c) Indexing overhead



(d) Effectiveness of aggregation

Figure 3.6: Impact of O and W on network and storage overhead (biased case)

- For $O = W$ case there is no aggregation in the system and the number of false matches is zero, which gives the best possible values for query routing metrics. For this case, the indexing overhead is 4, while effectiveness of aggregation is 0.

- With $O = 60$ and $W = 80$, at most 20 don't care bits are allowed in the aggregates. With this restriction almost no aggregation takes place. This justifies the high value of indexing overhead for $O = 60$ curve at $W = 80$ (see Figure 3.5(c)).

- Not all bits of a pattern are required for query routing with high accuracy. Query routing accuracy is almost the same for $O = 50$ and $O = W$ cases, whereas, indexing overhead for $O = 50$ case is only 65% of $O = W$ case.

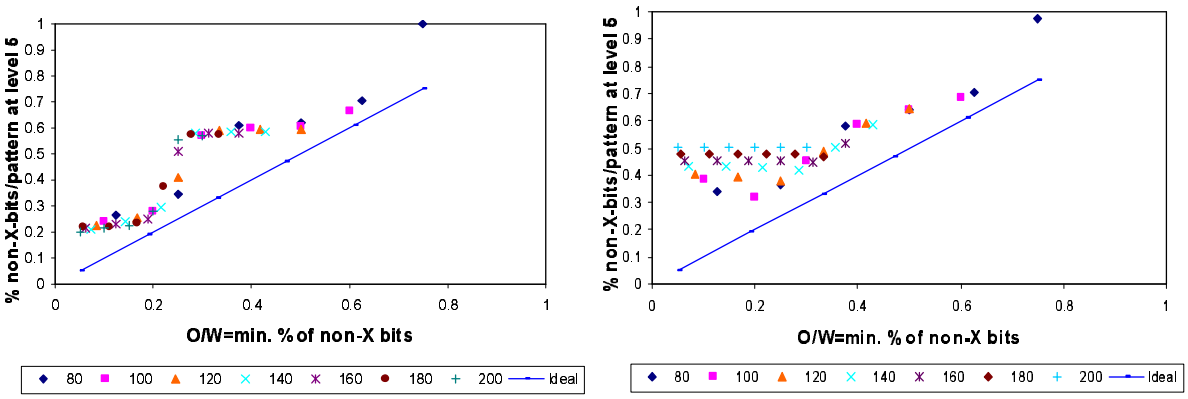
- For $O = 10$ and $O = 20$ curves query routing accuracy increases with increase in W. This is because we are using both positional value and content of each bit while matching a query to an aggregate. When W increases, number of possible positions of non-X bits increases, which reduces the probability of false matches. Hence, the decrease in the number of hops.

- For a fixed value of O indexing overhead decreases with increase in pattern-width (W). Given two random patterns, the probability that they will match on a given number of bits increases with W. This results into higher probability of aggregation and lower indexing overhead.

The observations presented for the random case (Figure 3.5) are equally applicable for the biased case (Figure 3.6). In addition, we can infer the followings by comparing these graphs.

- Query routing performance is in general better for the biased case, compared to the random case. Especially for $O = 10$ and $O = 20$ curves it is about 3 times better, while EA is almost identical for both cases.

- IO and EA are better for the biased case as well. For $O \geq 40$ curves, EA reaches its maximum upper limit [4] for the biased case (compare Figure 3.5(d) and Figure 3.6(d)). Which implies, higher level of aggregation is possible by reducing A, though at the expense of reduced routing efficiency.



(a) Random case (b) Biased case

Figure 3.7: Nature of aggregates

Based on these observations we can conclude that by controlling O (*i.e.*, the minimum number of non-X bits) we can control the amount of aggregation, but at the expense of increased query routing traffic. It is evident from the *first hit probes* and *average probes/hit* curves for random and biased cases (Figure 3.5(a), Figure 3.5(b), Figure 3.6(a) and Figure 3.6(b)), that there exists an oscillation *w.r.t.* the change in W. This effect can be explained by observing the curves in Figure 3.7, which plot the achieved level of aggregation (*i.e.*, % of non-X bits/pattern) against the allowed maximum (*i.e.*, $\frac{O}{W}$) level of aggregation. In these two diagrams the solid line is the plot of $\frac{O}{W}$. For random case there exists a shift at $\frac{O}{W} = 0.28$ and for the biased case the shift (though smaller)

[4] Upper limit for EA can be calculated as: $EA = 1 - \frac{\sum_{l=1}^{h} PA^{l-1}B^l n_l}{\sum_{l=1}^{h} PB^l n_l} = 1 - \frac{A(A^h-1)}{h(A-1)}$ (note: leaf peer do not aggregate)

is at $\frac{O}{W} = 0.34$. This implies that the achievable level of aggregation does not change continuously, rather there exists two discrete levels. The X-axis in Figure 3.7 indicates the inverse polarity of that in Figure 3.5 and Figure 3.6. Hence comparing these curves we can conclude that the routing efficiency increases with W once we are at the lower aggregation region *i.e.*, $\frac{O}{W} < 0.28$ for random case and $\frac{O}{W} < 0.34$ for biased case. Based on this observation we select $O = 50$ and $W = 180$ (*i.e.*, $\frac{O}{W} = 0.277$) for the subsequent experiments.

3.13.2 Scaling Behavior

In this section we analyze the scaling behavior of DPMS with growth in network size. Based on the observations presented in Section 3.13.1 we have chosen $W = 180$ and $O = 50$ for the experiment presented in this section. The number of peers (N) in the system has been varied from around $8,000$ to $21,000$ while keeping the number of peers per group at the highest level of indexing hierarchy in the range of $0.6 \log N$ and $1.5 \log N$. We have used $R = 1$ and $B = 4$ to remain compatible with the experiments presented in Section 3.13.1. The value of H was set to 5 to accommodate all the peers in the indexing hierarchy, without violating the above mentioned constraints.

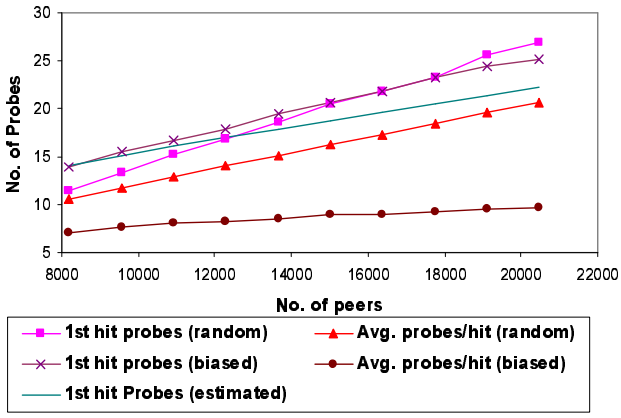


Figure 3.8: Scaling behavior

Figure 3.8 presents the curves for the first hit probes and the average probes/hit metrics according to the definitions presented in Section 3.13.1. The first hit probe includes the cost of flooding ($O(\log N)$) the peers in a group at the highest level of the indexing hierarchy. This justifies the gap between the first hit probes and the average probes/hit curves. Moreover, routing efficiency for biased case is much better than that of the random case, as already demonstrated in Section 3.13.1.

The *1st hit Probes(estimated)* curve has been computed based on the following equations:

$$probes = f \log_B N + \xi_{est} * h_{est} \tag{3.21}$$

$$h_{est} = \log_B \left(\frac{1}{B} + \frac{N(B-1)}{Bf \log_B N} \right) \tag{3.22}$$

The value of ξ_{est} has been derived from Equation (3.16) using the parameter values used in this experiment. In Equation (3.22), $f \log_B N$ represents the number of highest level peers; for this experiment f was varied from 0.6 to 1.5 based on the value of N. Equation (3.22) has been derived from Equation (3.10) for $R = 1$. Clearly, the estimated curve is close enough to the simulation results (see the curves in Figure 3.8).

This experiment also revealed that IO and EA do not depend on the number of peers in the system. These metrics are dependent on the maximum number of levels (H), and the replication factor (R). An increase in H (and/or R) increases IO (and decreases EA) For this experiment neither R nor H has been varied, hence IO and EA were constant as present in Table 3.3.

3.13.3 Fault tolerance

There exists two disadvantages of a tree-like, hierarchical indexing scheme: (a) the highest level peers become performance bottlenecks and (b) failure of an indexing peer results in

Table 3.3: Storage overhead in DPMS

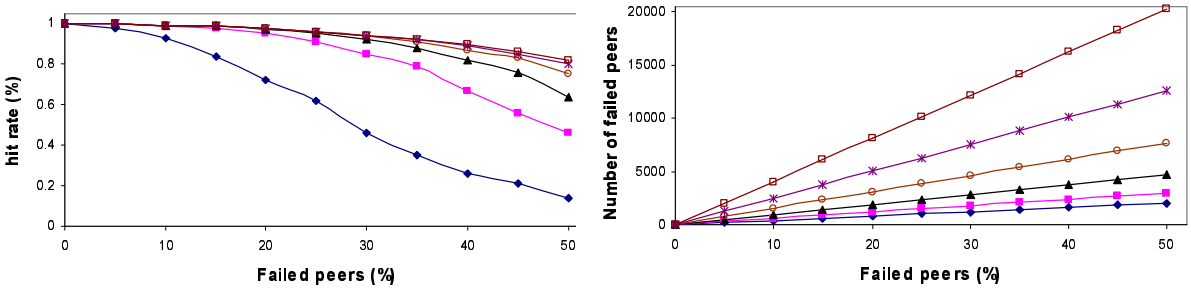
	IO	EA
Random	4.04	0.24
Biased	3.07	0.48

unreachable leaf peers. To overcome these problems we have incorporated the concept of replication in the DPMS indexing hierarchy. This section presents the impact of replication on routing performance in the presence of peer failures.

For the experiments in this section we have varied R from 1 to 6 while keeping all other parameters constant. We have used $H = 4$, $B = 4$, $O = 50$ and $W = 180$. For these settings the number of peers in the system was varied from about $4,000$ (for $R = 1$ case) to $40,500$ (for $R = 6$ case). For each value of R we have deactivated (*i.e.*, removed) up to 50% of the peers from the system in 5% steps, and have executed 3000 queries on the rest of the peers for each simulation run (*i.e.*, at each step). The impact of bias among patterns is orthogonal to fault-tolerance characteristics of the system, hence we have presented only the random case in this section.

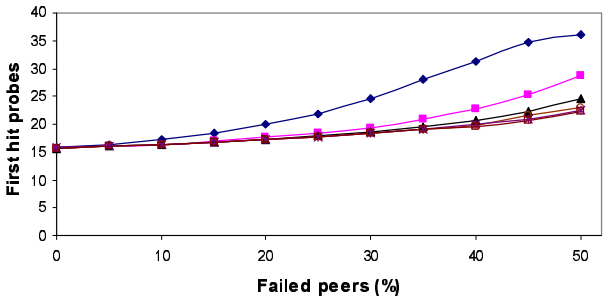
Like the previous experiments, we have used the *first hit probes* (Figure 3.9(c)) and the *average number of probes/hit* (Figure 3.9(d)) as the metrics for measuring query routing performance. However, for the previous experiments there were no peer failures, and so all the actual matches to a query could be discovered. But, for the experiments in this section this is not true anymore. Failure of indexing peers may result into unreachable leaf peers. To measure the impact of this phenomena we have defined *hit rate* (Figure 3.9(a)) as the average percentage of matches that are discovered by a query. The impact of replication on the overall storage overhead in the system is presented in Figure 3.10.

By analyzing the curves in Figure 3.9 and Figure 3.10 we can infer the followings:

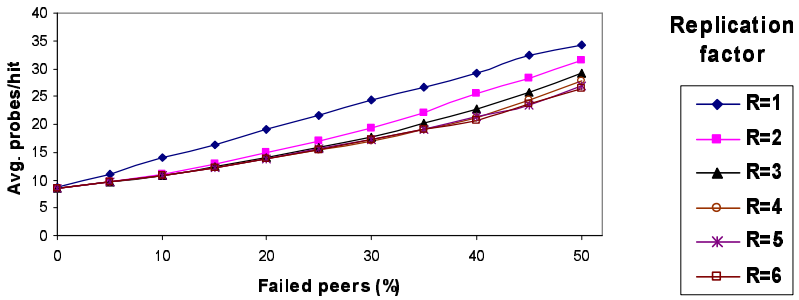


(a) Hit rate

(b) Number of failed peers



(c) First hit probes



(d) Avg. probes per hit

Figure 3.9: Impact of replication and peer failure on routing performance and hit rate

- Without any replication ($R = 1$ case) the hit rate reduces drastically with increase in the percentage of failed peers (see Figure 3.9(a)). Failure of an indexing peer, in this case, makes all of the leaf peers in its subtree unreachable. Routing efficiency is also low in this case, as many probes are wasted (*i.e.*, evaluated as false match at the parent of a failed indexing peer) while trying to route queries toward unreachable leaf peers.

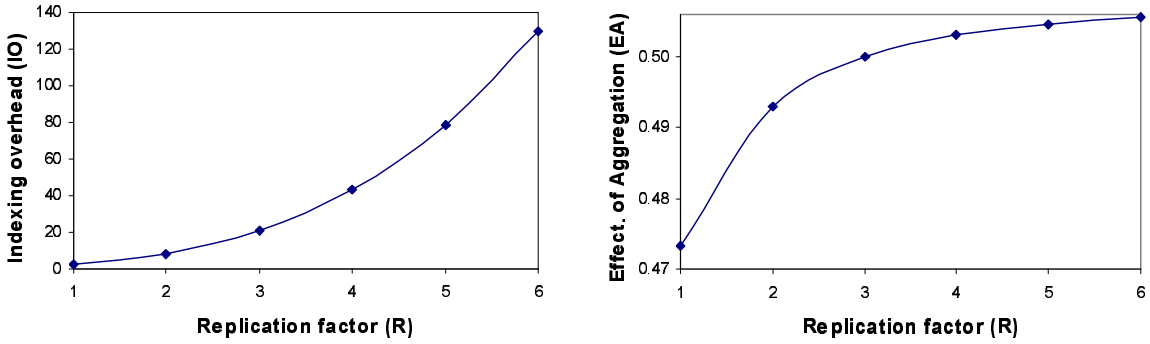
- Routing efficiency and hit rate increases with increase in R. This increase in hit-rate and routing efficiency (*i.e.*, a decrease in the number of probes) diminishes with higher values of R (e.g. $R = 5$ or $R = 6$).

- The downside of replication is the exponential increase in the indexing overhead (see Figure 3.10(a)). However effectiveness of aggregation increases with increases in R (see Figure 3.10(b)). Based on the equations in Section 3.12.1 and the definition of EA (see Equation (3.20))), it can be shown that the value of EA tends to A^{h-1} as R tends to infinity. This justifies the gradual decrease in EA in Figure 3.10(b).

In the light of the experiments presented in this section we can conclude that replication is necessary for improving reliability of the proposed system. A replication factor of 2 or 3 can satisfy the need of most applications, assuming the peer failure rate is less than 25%.

3.13.4 Indexing Hierarchy

In this experiment we focus on two aspects of the index distribution hierarchy: *peer distribution* and *index distribution*. We simulated a system of about 20,000 peers. In consistence with the experiment in Section 3.13.2 we set the parameter values to $H = 5$, $B = 4$, $A = 0.6$, $O = 50$ and $W = 180$.



(a) (b)

Figure 3.10: Impact of replication on storage overhead

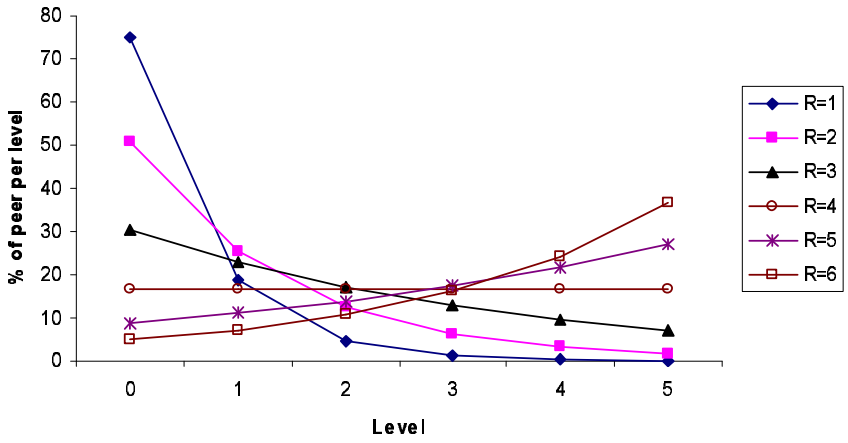


Figure 3.11: Distribution of peers along the indexing hierarchy

Distribution of peers along the indexing hierarchy mostly depends on replication factor R and branching factor B. For this experiment we varied R and recorded the percentage of peers at different levels along the indexing hierarchy. As demonstrated in Figure 3.11, percentage of indexing peers increases with R. We can utilize this behavior to accommodate the relative population of indexing peers and leaf peers in accordance with the target system.

Index size grows exponentially as we move up along the indexing hierarchy. In this

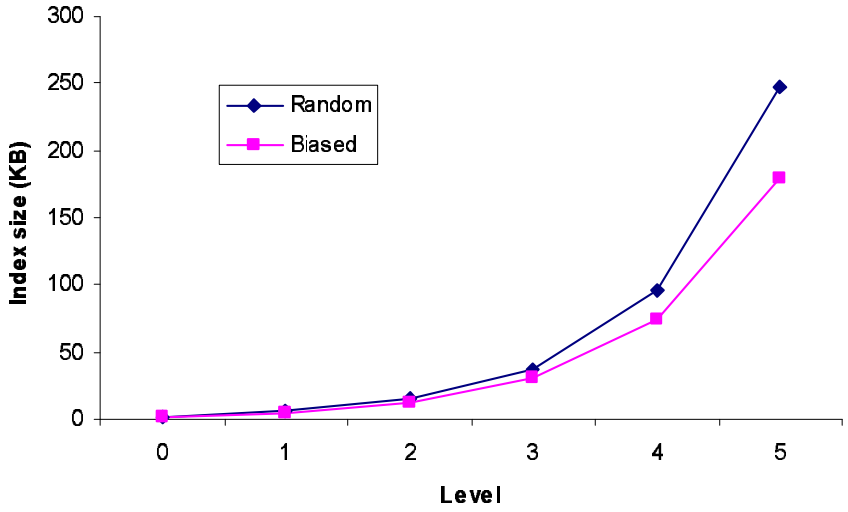


Figure 3.12: Variation in index size along the indexing hierarchy

experiment we try to provide an estimate of the expected index size in peers at different levels of the indexing hierarchy. Index size at an indexing peer grows linearly with the average number of advertisements per leaf peer. Another factor influencing index size is the achievable level of aggregation (*i.e.*, A) which in turn depends on the nature of bias in advertised information. Figure 3.12 presents the average index size in peers at different levels of the indexing hierarchy. Results for both random and bias data are presented. The investigation with Gnutella network carried out in [124], suggests that more than 75% of Gnutella peers advertise less than 100 files and more that 25% of them do not advertise at all. Based on this observation we set the average number of advertisements per leaf peer (*i.e.*, P) to 100. We can infer from Figure 3.12 that the average index size in higher level peers is lower for biased data. About 7% peers at the highest level of the indexing hierarchy reports high index size, which is on average 179KB and 247KB for biased and random cases, respectively. It will require about $2.9 \sim 4.1$ minutes (for biased case and random case respectively) to refresh the index at a peer using a 1KBps stream. Use of a compression technique can further reduce the refresh period.

3.14 Summary

In this chapter we have presented DPMS as a semi-structured solution to the DPM problem. DPMS supports flexible queries involving partial and multiple keywords. Query routing efficiency of DPMS is comparable to that of the structured P2P systems. For moderately stable networks, DPMS provides guarantee on search completeness and on the discovery of rare items. Peers in DPMS maintain constant number of links, in contrast to $O(\log N)$ links per peer required by most DHT-based systems. DPMS can easily be tailored to achieve context-sensitive (*e.g.*, network proximity, user interest *etc.*) query routing. Moreover, DPMS can exploit the heterogeneity in peer capabilities, and does not place any hard restriction on document or index placement.

The main drawback of the proposed system is the storage overhead introduced by hierarchical indexing and replication. Experimental results presented in this chapter demonstrate the worst possible values for the storage overhead (*i.e.*, for random case). For most applications, there exists some bias among the advertised patterns, which can enable higher levels of aggregation and hence lower levels of storage overhead, as demonstrated by the results for the biased case.

Another problem in DPMS stems from leave/join of leaf peers. Leave/join of indexing peers has local effect only. But, leave/join of a leaf peer results into cascaded updates along its replication tree. This problem can be mitigated by using periodic and differential updates of index information between adjacent indexing peers. This latency in update will not hamper the normal operation of the system other than degrading query routing performance to some extent.

Modification of data at leaf peers (*e.g.*, change in filename, insertion of new data/file or deletion of existing data/file) will invalidate the associated index information. This problem persists in any structured or semi-structured system, though the effect is higher in a hierarchical indexing system like DPMS. It has been demonstrated in [134] that in

modern 2-tier Gnutella network about 36% peers do not change their shared content over a one week interval. However, that number increases to 69%, 80% and 90% over one day, 6 hours and 2 hours intervals, respectively. Thus it is possible to reduce the effect of data dynamism using periodic index updates.

Chapter 4

Plexus

4.1 Introduction

In this chapter we introduce a *structured* [22] routing mechanism, named Plexus [15, 17], as a solution to the DPM problem. The routing algorithm in Plexus is based on the theory of *Error Correcting Codes (ECC)* [76]. The novelty of the proposed approach lies in the use of Hamming distance based routing, in contrast to the numeric distance based routing adopted in traditional DHT-approaches. This property makes subset matching capability intrinsic to the underlying routing mechanism. Plexus uses patterns (like Bloom filters [30]) to summarize the identifying properties associated with a shared object. It provides an efficient mechanism for advertising a binary pattern, and discovering it by using any subset of its 1-bits. Plexus has a *partially decentralized* architecture involving superpeers. The number of routing hops for resolving a query and the number of links maintained by each indexing peer scales logarithmically with the number of peers in the system. Plexus attains better resilience to peer failure using replication and redundant routing paths. The concepts presented in this chapter are supported with theoretical analysis, and simulation results obtained from the application of Plexus to partial keyword search utilizing the Extended Golay code[65].

4.2 Chapter Organization

The rest of this chapter is organized as follows. Preliminaries on coding theory are presented in Section 4.3. Section 4.4 explains the theoretical model of Plexus, while Section 4.5 presents the overlay topology construction and maintenance protocols. Simulation results, supporting our claims, are presented in Section 4.6. Finally, we summarize the concepts and the findings of this chapter in Section 4.7.

4.3 Preliminaries

In this section, we explain the properties of linear codes and the Extended Golay code. We highlight only those properties that will be required for the discussions in the subsequent sections.

4.3.1 Linear Covering Codes

Let $\mathbb{F}_2^n$ define the linear space of all n-tuples (or vectors) over the finite field $\mathbb{F}_2 = \{0, 1\}$. A linear binary code of length n is a *subspace*[1] $\mathcal{C} \subset \mathbb{F}_2^n$. Each element in $\mathcal{C}$ is called a *codeword*. A linear covering code is specified by using four parameters $(n, k, d)f$. Here, k is the dimension of the code. This indicates that there exists a total of 2^k codewords in the code. d is the minimum Hamming distance between any two codewords and n is the length of each codeword in bits. The *covering radius* f is the smallest integer such that every vector $P \in \mathbb{F}_2^n$ is covered by at least one $B_f(c_i)$. Here, $B_f(c_i) = \{P \in \mathbb{F}_2^n | d(P, c_i) \leq f\}$ is the *Hamming sphere* of radius f centered at codeword c_i.

From error correcting perspective, a good $(n, k, d)f$-code should have a small n (for fast transmission), a large k (for increased information content), and a large d (for cor-

[1] V is a vector subspace over $\mathbb{F}_2^n$ if $V \subset \mathbb{F}_2^n$ and for all $a, b \in V$, $a \oplus b \in V$. Here, $\oplus$ is the *addition* operation defined over $\mathbb{F}_2^n$

recting many errors).

Since the set of codewords in $\mathcal{C}$ is a *subspace* of $\mathbb{F}_2^n$, the XOR of any two codewords, u and v is also a codeword, *i.e.*, $\forall u, v \in \mathcal{C} \implies u \oplus v \in \mathcal{C}$. This property allows the entire set of codewords *i.e.*, $\mathcal{C}$ to be represented in terms of a minimal set of codewords, known as a *basis*, containing exactly k codewords. These k codewords, $g_1, g_2, \ldots, g_k$, are collated in the rows of a $k \times n$ matrix known as the *generator matrix*, $G_\mathcal{C}$, for code $\mathcal{C}$. The codewords of $\mathcal{C}$ can be generated by XORing any number of rows[2] of $G_\mathcal{C}$. The generator matrix for any linear code $\mathcal{C}$ can be expressed as,

$$G_\mathcal{C} = [I_k B] = [g_1 g_2 \ldots g_k]^T \tag{4.1}$$

where, I_k is the $k \times k$ identity matrix, and B is a $k \times (n-k)$ matrix. The dual code $\mathcal{C}^\perp$ of a linear code $\mathcal{C}$ is defined as,

$$\mathcal{C}^\perp = \{x \in \mathbb{F}_2^n | x \cdot c = 0 \; \forall c \in \mathcal{C}\}.$$

Here, $x \cdot c$ represents vector dot product over $\mathbb{F}_2$. A linear code is said to be *self-dual*, if $\mathcal{C}^\perp = \mathcal{C}$. For any codeword c of a self-dual linear code $\mathcal{C}$, $c \in \mathcal{C} \implies \bar{c} \in \mathcal{C}$, where $\bar{c}$ represents the bit-wise complement of c.

4.3.2 Extended Golay Code

The extended Golay code, $\mathcal{G}_{24}$, is a $(24, 12, 8)4$ self-dual linear binary code. It has $4096 (= 2^{12})$ codewords of length 24-bits each. The minimum distance between any two codewords is 8. The weight[3] distribution of this code is $0^1\, 8^{759}\; 12^{2576}\, 16^{759}\, 24^1$. In other words, $\mathcal{G}_{24}$ contains the all zero vector $\vec{0}$[4] and the all one vector $\vec{1}$. Exactly 759 codewords have

[2]Note that $\sum_{i=0}^{k} \binom{k}{i} = 2^k$.

[3]The number of 1-bits in a pattern (say P) is known as its weight $(= |P|)$

[4]Any linear code contains the all zero vector, $\vec{0}$

weight 8 (known as *special octads*), 2576 codewords have weight 12 (known as *umbral dodecads*), and 759 codewords have weight 16 (called *16-sets*).

Any vector in $\mathbb{F}_2^{24}$ can be categorized into 49 *orbits w.r.t.* $\mathcal{G}_{24}$ (see Figure 1. in [49]). These orbits are denoted as $S_w(0 \leq w \leq 24)$, $T_w(8 \leq w \leq 16)$, $U_w(6 \leq w \leq 18)$, P_{12} and X_{12}, where the subscript w denotes the weight of the vectors in that orbit. All vectors in a given orbit exhibit identical distance properties from the codewords in $\mathcal{G}_{24}$. Figure 4.1 (a portion of Figure 1 in [49])) depicts some of these orbits. An edge between orbits A and B indicates that a vector in orbit B can be obtained from some vector in orbit A (and vice versa) by complementing a single bit. The minimal hamming distance of a vector in orbit A from some vector in orbit B is essentially the length of the shortest path from node A to node B in the graph of Figure 4.1. Orbits S_8, U_{12} and S_{16} correspond to the *special octads*, *umbral dodecads* and *16-sets*, respectively.

4.4 Theoretical Model of Plexus

4.4.1 Core Concept

The originality of this work lies in the use of Hamming distance based clustering of pattern-space, as opposed to numeric distance utilized by DHT-techniques. Yet, similar to DHT-based systems, Plexus employs a three-party rendezvous mechanism for query resolution, *i.e.*, a third entity works as a mediator for query resolution between the advertising party and the searching party.

In Plexus, advertisements and queries are routed to two different sets of peers in such a way that the queried set of peers and the advertised set of peers have at least one peer in common, whenever a query pattern constitute a subset of the 1-bits, as present in an advertised pattern. As explained in Figure 4.2, we partition the entire pattern space $\mathbb{F}_2^n$ into clusters and select a unique representative (cluster head) for each cluster. Let $\mathcal{C}$ be

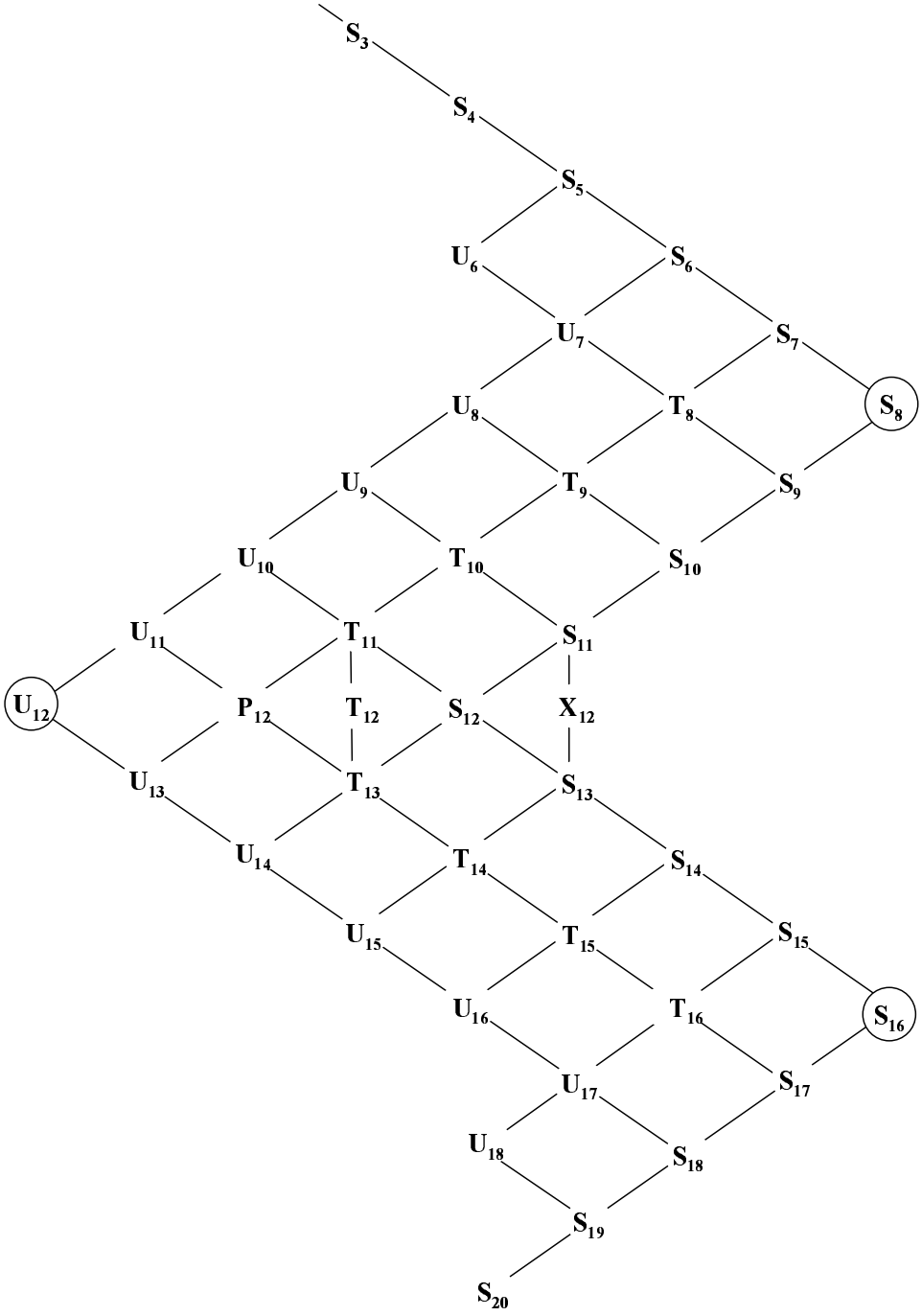


Figure 4.1: Relationships among the orbits of the vectors in $\mathbb{F}_2^n$ *w.r.t.* to the codewords in $\mathcal{G}_{24}$. Circled orbits correspond to the codewords of $\mathcal{G}_{24}$.

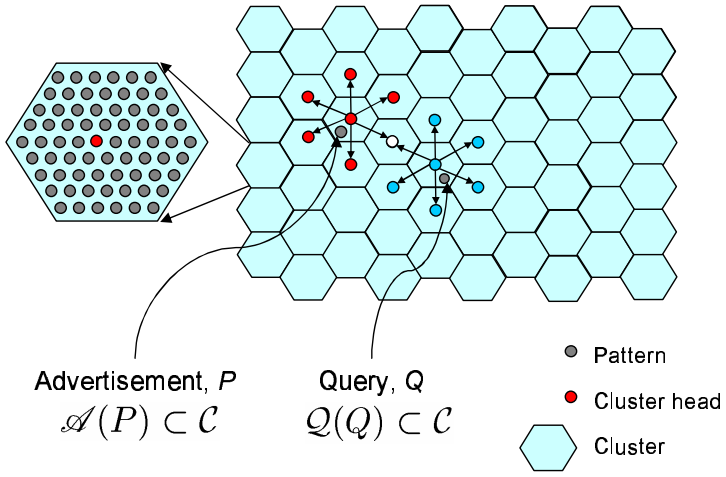


Figure 4.2: The core concept

the set of these cluster heads. Now the basic concept is to map a query pattern Q to a set of cluster heads ($\mathcal{Q}(Q) \subset \mathcal{C}$) and to map an advertised pattern P to another set of cluster heads ($\mathscr{A}(P) \subset \mathcal{C}$), such that $\mathcal{Q}(Q)$ and $\mathscr{A}(P)$ has *at least one* cluster head in common whenever the 1-bits of Q constitute a subset of the 1-bits in P. Mathematically,

$$Q \subseteq P \implies \mathcal{Q}(Q) \cap \mathscr{A}(P) \neq \emptyset \tag{4.2}$$

Clustering of pattern space has been extensively studied in Artificial Intelligence (AI) and Coding Theory [76] literature. AI-based clustering techniques [79] require priori knowledge of the pattern space (*e.g.*, pattern density distribution) and training phases.

Coding theory constructs, on the other hand, assume that all the patterns are equally likely. These techniques distinguish a set of patterns as cluster-heads (codewords), which cover the entire (or most of the) pattern space with no (or very little) overlaps. For a $(n, k, d)f$ code $\mathcal{C}$, a codeword $c_i \in \mathcal{C}$ represents all the patterns in $B_f(c_i)$, *i.e.*, all patterns within the Hamming sphere of radius f with center at c_i. Each peer is assigned one (or more, as explained later) codeword (c_i) and becomes responsible for all the patterns within its Hamming sphere $B_f(c_i)$.

Now the challenge is to compute $\mathscr{A}$(P) and $\mathcal{Q}$(Q), and to devise a mechanism for routing within the overlay. Coding theory literature does not provide any straight forward way to calculate $\mathscr{A}$(P) and $\mathcal{Q}$(Q), satisfying Equation (4.2). We present the algorithm for computing $\mathscr{A}$(P) and $\mathcal{Q}$(Q) in Section 4.4.2, and the routing algorithm is presented in Section 4.4.3

4.4.2 Computing $\mathscr{A}$(P) and $\mathcal{Q}$(Q)

This section presents the algorithms for computing $\mathcal{Q}$(Q) and $\mathscr{A}$(P). Given a code $\mathcal{C}$, a trivial way of computing $\mathscr{A}$ and $\mathcal{Q}$ is to use bounded distance decoding [135]; *i.e.*,

$$\begin{aligned} \mathscr{A}(P) &= B_s(P) \cap \mathcal{C} = \{Y | Y \in \mathcal{C} \wedge d(Y,P) \leq s)\} \quad and \\ \mathcal{Q}(Q) &= B_t(Q) \cap \mathcal{C} = \{Y | Y \in \mathcal{C} \wedge d(Y,Q) \leq t)\} \end{aligned} \tag{4.3}$$

for positive integers s and t. Now we want to compute the minimum $d(P,Q)$ without violating Equation (4.2). If the covering radius of $\mathcal{C}$ is f then the Hamming sphere of radius f around any arbitrary point should contain at least one codeword. Hence, according to Figure 4.3,

$$\begin{aligned} d(u,v) &= s + t - d(P,Q) \geq 2f \\ \Rightarrow d(P,Q) &\leq s + t - 2f. \end{aligned} \tag{4.4}$$

In other words, if we advertise to all the codewords in $B_s(P) \cap \mathcal{C}$ and search all the codewords in $B_t(Q) \cap \mathcal{C}$ then any subset Q of P within distance $d(P,Q) \leq s+t-2f$ can be discovered.

We define *query stretch* as the maximum value of $d(P,Q)$ without violating Equation (4.2). $|\mathcal{Q}|$ and $|\mathscr{A}|$ are proportional to query stretch. In practice we can achieve higher query stretch than Equation (4.4) while keeping $|\mathcal{Q}|$ and $|\mathscr{A}|$ within reasonable limits. In the rest of this section we describe a method for obtaining $\mathscr{A}$(P) and $\mathcal{Q}$(Q)

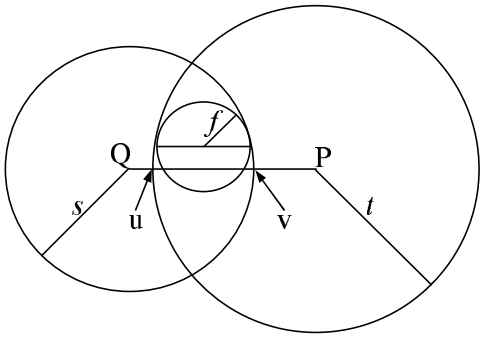


Figure 4.3: Bound on $d(P, Q)$

for $\mathcal{G}_{24}$, with a query stretch of 11, which will allow maximum weight of an advertised pattern ($|P|$) to be 14-bits and minimum weight of a query pattern ($|Q|$) to be 3-bits.

The discussion in the rest of this section is specific to $\mathcal{G}_{24}$. Special consideration is required to adopt these algorithms for other linear codes.

As explained in Section 4.3.2, any pattern P of length 24 belongs to one of the 49 orbits *w.r.t.* $\mathcal{G}_{24}$. Any vector in a given orbit has the same distance properties as listed in Table 4.1. In this table, the construct $d : x$ stands for distance (d) and number of codewords(x) at distance d. For vectors in a given orbit, we have listed only the number of *octads* and *dodecads* within distance 5 and 6, respectively.

The number of 1-bits in a query or advertisement is restricted to the range of 3 to 14. The reason behind this restriction can be justified by observing the following property of Bloom filter. As discussed in Section 2.3.2, $33 \sim 50\%$ bits of a well-designed Bloom filter are 1. Therefore, for a 24-bit chunk from a Bloom-filter $8 \sim 12$ bits are expected to be 1. Queries having fewer than three 1-bits are too generic, and are likely to match a large number of advertisements. Due to this restriction, we only need to consider the *octads* and *dodecads* in $\mathcal{Q}$ and $\mathscr{A}$ calculation.

Most queries (involving $3 - 8$ 1-bits) are closer to the octads than the dodecads or the 16-sets. This results in a bias for $\mathcal{Q}$ (and eventually $\mathscr{A}$) to have an octad:dodecad ratio higher than $2 : 7 (\approx 759 : 2576)$. To reduce the effect of this bias, the codewords

Table 4.1: Distance distribution of orbits from octads and dodecads

	S_8	U_{12}		S_8	U_{12}
S_3	5 : 21		S_{11}	3:1,5:2	5 : 16
S_4	4 : 5		T_{11}	5 : 5	3 : 1, 5 : 15
S_5	3:1,5:20		U_{11}		1 : 1
S_6	2 : 1	6 : 16	S_{12}	4 : 1	4 : 4, 6 : 48
U_6	4 : 6	6 : 18	T_{12}		4 : 6, 6 : 40
S_7	1 : 1		U_{12}		0 : 1
U_7	3:1,5:15	5 : 6	X_{12}	4 : 3	6 : 64
S_8	0 : 1		P_{12}		2 : 1, 6 : 55
T_8	2 : 1	6 : 42	S_{13}	5 : 3	5 : 16
U_8	4 : 4	4 : 2, 6 : 32	T_{13}	5 : 1	3 : 1, 5 : 15
S_9	1 : 1		U_{13}		1 : 1
T_9	3:1,5:7	5 : 14	S_{14}		6 : 56
U_9	5 : 12	3 : 1, 5 : 9	T_{14}		4 : 4, 6 : 42
S_{10}	2 : 1	6 : 56	U_{14}		2 : 1, 6 : 45
T_{10}	4 : 2	4 : 4, 6 : 42			
U_{10}		2 : 1, 6 : 45			

in S_{16} are adopted as replicas of the octads (S_8). This helps in reducing the volume of query traffic at the octads.

Pseudocodes for finding $\mathcal{Q}$ and $\mathscr{A}$ are presented in Algorithm 3 and Algorithm 4, respectively, in light of the preceding discussion. The algorithm for finding $\mathcal{Q}$(Q) starts with finding the set $\mathcal{Q}$ of the octads and dodecads that are within distance 5 and 6, respectively, from the query pattern Q. If $\mathcal{Q}$ contains fewer than τ codewords, then it is appended with the codewords that are reachable in one hop from the current members of $\mathcal{Q}$ and are within distance 7 (if $|Q|$ is odd) or 8 (if $|Q|$ is even) from Q.

$E(|\mathscr{A}|)$[5] is inversely proportional to $E(|\mathcal{Q}|)$, which in turn is proportional to τ. Hence, τ can be tuned to achieve a desirable ratio of search and advertisement traffic. For our experiments, $\tau = 5$ is used. A lower value of τ can be used if the anticipated volume of user queries is much higher than advertisements.

The algorithm for finding $\mathscr{A}$(P) has two stages. The first stage (lines 4-9) is to find set

[5] $E(X)$ is the expected or average value of variable X

Algorithm 3 find $\mathcal{Q}(Q)$

Input: $Q \in \mathbb{F}_2^{24}$

External: τ controls size of $\mathcal{Q}$ and $\mathscr{A}$

Returns: $\mathcal{Q} \subset \mathcal{G}_{24}$

$\mathcal{Q} \leftarrow \{Y | (Y \in S_8 \wedge d(Y,Q) \leq 5)$
$\bigvee (Y \in U_{12} \wedge d(Y,Q) \leq 6)\}$

if $|\mathcal{Q}| < \tau$ **then**

  $\imath \leftarrow (|Q| \text{ is odd})?\ 7 : 8$

  $\mathcal{Q}' \leftarrow \emptyset$

  for each $Y \in \mathcal{Q}$ **do**

    $\mathcal{Q}' \leftarrow \mathcal{Q}' \cup \{Z | Z \in neighbors(Y)$
    $\wedge d(Z,Q) \leq \imath \wedge Z \notin \mathcal{Q}\}$

  end for

  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \mathcal{Q}'$

end if

return $\mathcal{Q}$

$\mathscr{P}$ of $\mathcal{Q}(Q)$s that will be searched by all possible Q matching the advertised pattern P. Now the problem is to find a small (preferably minimum) set of codewords $\mathscr{A}$ such that $\mathscr{A}$ contains at least one element from each set in $\mathscr{P}$. This is essentially the *minimum hitting set* problem, which in turn, is equivalent to the *minimum set cover* problem. To find $\mathscr{A}$, we have applied the greedy algorithm (line 10-14) based on [41].

The time complexity of computing $\mathscr{A}$(P) is dominated by the loop in lines 5-8, where we compute $\mathscr{P}$. The upper bound for size of $\mathscr{P}$ is $|\mathscr{P}| \leq \sum_{i=3}^{|P|} \binom{|P|}{i} 2^k$. For $\mathcal{G}_{24}$ and smaller codes we can compute $\mathscr{A}$(P) in reasonable time. However for larger codes the time complexity will be an obstacle for the implementation. This is one of the main reasons behind selecting $\mathcal{G}_{24}$ for the implementation. With the ongoing research on list decoding techniques ([56],[145],[151]) we can hope to see efficient algorithms for dealing with larger codes.

Our implementation of computing $\mathscr{A}$(P) takes 4.27 seconds on average on a regular Pentium 4 1.7GHz machine. We can significantly reduce computation time by reducing the query stretch. This will eventually result into lower hit-rate when smaller percentage of η-grams is used for computing a query.

4.4.3 Routing

In Section 4.4.3.1 we explain the routing links for a peer and provide a bound on the maximum number of routing hops required for reaching any peer in the overlay. In Section 4.4.3.2 we illustrate the mechanism of applying the Generator matrix for next hop selection while routing a message. Section 4.4.3.3 presents the special considerations for routing using the Extended Golay code. Finally, the algorithm for multicast routing has been presented in Section 4.4.3.4.

By "peer X" we mean a peer responsible for codeword X. In this section, a peer is assumed to be associated with a single codeword. Section 4.5.2 presents a method for

Algorithm 4 find $\mathscr{A}(P)$

Input: $P \in \mathbb{F}_2^{24}$

Returns: $\mathscr{A}(P) \subset \mathcal{G}_{24}$

$\mathscr{A} \leftarrow \{Y | (Y \in S_8 \wedge d(Y,P) \leq 5)$

$\bigvee (Y \in U_{12} \wedge d(Y,P) \leq 6)\}$

$\mathscr{P} \leftarrow \emptyset$

for each Q s.t. $Q \wedge P = Q \wedge |Q| \geq 3 \wedge d(P,Q) > 3$ **do**

  if $\mathscr{A} \cap \mathcal{Q}(Q) = \emptyset$ **then**

    $\mathscr{P} \leftarrow \mathscr{P} \cup \{\mathcal{Q}(Q)\}$

  end if

end for

{*use greedy heuristic to find minimum hitting set*}

while $\mathscr{P}$ not empty **do**

  find Y s.t. Y is in maximum no. of sets $\mathscr{S} \in \mathscr{P}$

  $\mathscr{A} \leftarrow \mathscr{A} \cup Y$

  $\mathscr{P} \leftarrow \mathscr{P} - \{\mathscr{S} | \mathscr{S} \in \mathscr{P} \wedge Y \in \mathscr{S}\}$

end while

return $\mathscr{A}$

assigning multiple codewords to a peer.

4.4.3.1 Routing Table

As discussed in Section 4.3.1, the codewords of a linear code (C) form a vector subspace of $\mathbb{F}_2^n$. The basis vectors of this vector subspace are represented as the rows of the generator matrix for the code. Consider a (n, k, d) linear code (C) with generator matrix $G_{\mathcal{C}} = [g_1, g_2, \ldots, g_k]^T$. To route using this code, peer X has to maintain links to $(k+1)$ superpeers with IDs $X_1, X_2, \ldots, X_{k+1}$, computed as follows:

$$X_i = \begin{cases} X \oplus g_i & 1 \leq i \leq k \\ X \oplus g_1 \oplus g_2 \oplus \ldots \oplus g_k & i = k+1 \end{cases} \tag{4.5}$$

Theorem 4.4.1. *Suppose we are using a (n, k, d) linear code $\mathcal{C}$ and each superpeer is maintaining $(k+1)$ routing links as specified in Equation (4.5). In such an overlay, it is possible to route a query from any source to any destination codeword in less than or equal to $\frac{k}{2}$ routing hops.*

Proof. According to the definition of linear codes, $\vec{0} \in \mathcal{C}$, the rows of $G_{\mathcal{C}}$ (*i.e.*, $g_1, g_2 \ldots g_k$) form a basis for the subspace $\mathcal{C}$, and $\mathcal{C}$ is closed under XOR operation. This implies, for any permutation $(i_1, i_2, \ldots, i_k)$ of $(1, 2, \ldots, k)$,

$$X \in \mathcal{C} \implies Y = (X \oplus g_{i_1} \oplus g_{i_2} \oplus \ldots \oplus g_{i_t}) \in \mathcal{C} \tag{4.6}$$

for $1 \leq t \leq k$, *i.e.*, 2^k distinct codewords of $\mathcal{C}$ can be generated by XORing any combination of $1, 2, \ldots, k$ rows of $G_{\mathcal{C}}$ with X.

Suppose peer X (source) wants to route a message to peer Y (target) (see Equation (4.6)). Now, X can route the message to any of $X_j = X \oplus g_{i_j}$ in one hop by using its routing links (see Equation (4.5)). Suppose X routes to $X_1 = X \oplus g_{i_1}$. X_1 will evaluate Y as $Y = X_1 \oplus g_{i_2} \oplus \ldots g_{i_t}$. Note that Y is one hop closer to X_1 than X. X_1 can route

the message to any of $X_1 \oplus g_{i_2}$, $X_1 \oplus g_{i_3}$,...,$X_1 \oplus g_{i_t}$ peers in one hop. In this way, the query can be routed from X to Y in exactly t-hops.

If $t \leq \frac{k}{2}$, then our claim is justified. Now let $t > \frac{k}{2}$. For this case, we can write $Y = X_{k+1} \oplus g_{i_{t+1}} \oplus g_{i_{t+2}} \oplus \ldots \oplus g_{i_k}$, according to the definitions of X_{k+1} and Y in Equation (4.5) and Equation (4.6), respectively. Now using the $(k+1)$-th link, X can route the message to X_{k+1} in one hop, and X_{k+1} can route the message to Y in $(k-t-1)$ hops. Hence for $t > \frac{k}{2}$ we will need at most $(k-t-1+1) \leq \frac{k}{2}$ hops. □

4.4.3.2 Next Hop Selection

Given the above described routing protocol, peer X will need a way to find the rows of $G_{\mathcal{C}}$, satisfying Equation (4.6), in order to route a message to peer Y. To deal with this problem, the standard form of the generator matrix, $G_{\mathcal{C}} = [I_k|B]$ is used in conjunction with the following theorem.

Theorem 4.4.2. *Suppose peer X wants to route to peer Y and needs to find the g_{i_j}'s satisfying Equation (4.6). If $G_{\mathcal{C}}$ is in standard form, then the first k-bits of $X \oplus Y$ have 1-bits in exactly $\{i_1, i_2, \ldots, i_t\}$ positions.*

Proof. Let $\theta = X \oplus Y$. By using the definition of Y in Equation (4.6), we get $\theta = g_{i_1} \oplus g_{i_2} \oplus \ldots \oplus g_{i_t}$. Since $G_{\mathcal{C}}$ is in the standard form, only the i_j^{th} row of $G_{\mathcal{C}}$ (*i.e.*, g_{i_j}) has a 1-bit in i_j^{th} bit position for any $i_j \in \{1, 2, \ldots, k\}$. Therefore, row g_{i_j} has to be present in the linear combination of the rows of $G_{\mathcal{C}}$ producing θ. □

As discussed in the previous section, there exists a number of alternative paths between any source pattern X and any destination pattern Y. Let, $\Psi_t(X)$ return the first t bits of X. The number of alternate paths between X and Y, say $\Im(X, Y)$, can be calculated as follows.

$$\Im(X, Y) = \prod_{i=1}^{|\Psi_k(X \oplus Y)|} i = (|\Psi_k(X \oplus Y)|)! \tag{4.7}$$

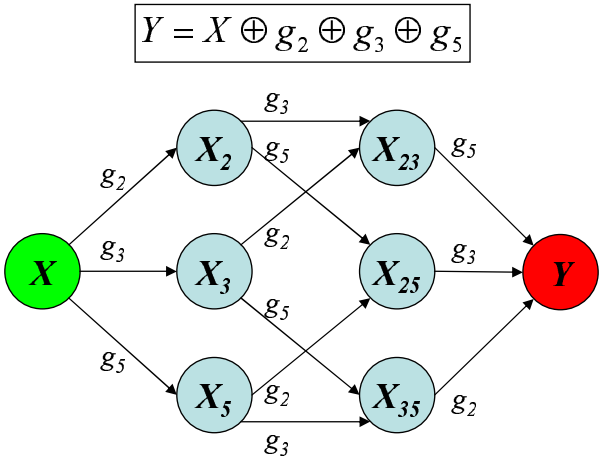


Figure 4.4: Bound on $d(P, Q)$

As an illustration of Equation (4.7) we present an example in Figure 4.4. In this example we assume that $X = Y \oplus g_2 \oplus g_3 \oplus g_5$, *i.e.*, the destination Y is 3 hops away from the source X. In this figure we compute $X_i = X \oplus g_i$ and $X_{ij} = X \oplus g_i \oplus g_j$. X can route to X_2, X_3 or X_5, *i.e.*, 3 possibilities. Suppose X routes to X_3. X_3 can route to X_{23} or X_{35}, *i.e.*, in 2 ways. Finally the message can reach Y through X_{23} or X_{35} in 1 way. The existence of multiple paths between any pair of codewords make the routing mechanism in Plexus resilient to failure of intermediate nodes. This phenomena is reflected in the experimental results presented in Section 4.6.4 and Section 5.6.5.

4.4.3.3 Routing with $\mathcal{G}_{24}$

The extended Golay code, $\mathcal{G}_{24}$, (like other linear codes) can be defined using different sets of basis vectors (*i.e.*, generator matrices). The generator matrix used in our implementation is $G_{24} = [I_{12}|B_{12}]$, where

$$B_{12} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{4.8}$$

This generator matrix has two desirable properties:

1. $g_1 \oplus g_2 \oplus \ldots \oplus g_k = \vec{1}$

2. $\forall X \in \mathcal{G}_{24}, \qquad d(X, X \oplus g_i) = |X \oplus X \oplus g_i| = 8$

The first property implies that $X_{k+1} = X \oplus g_1 \oplus g_2 \oplus \ldots \oplus g_k = X \oplus \vec{1} = \bar{X}$, and according to Equation (4.6) $X_{k+1} = \bar{X} \in \mathcal{G}_{24}$. This construct is possible because $\mathcal{G}_{24}$ is a self-dual code. The second property ensures the minimum distance of 8 between any peer and any of its first k-neighbors $(X_1, \ldots, X_k)$. These two properties influence the routing strategy based on $\mathcal{G}_{24}$ as follows.

In any 3-party rendezvous architecture the negotiating middle entity can become a performance bottleneck and a single point of failure unless appropriate measures are taken. In Plexus, replication is employed to mitigate the performance problem arising from the failure of superpeers. The information indexed at peer Y is replicated at peer $\bar{Y}$. The choice of $\bar{Y}$ as the replica for Y can be justified as follows.

- It can be proved that if $G_{\mathcal{C}}$ is in standard form then $Y = X \oplus g_{i_1} \oplus \ldots g_{i_t} \implies \bar{Y} = X \oplus g_{i_{t+1}} \oplus \ldots g_{i_k}$. Thus, paths from X to Y and X to $\bar{Y}$ are disjoint. This increases fault-resilience and influences uniform distribution of query traffic, especially in cases where peer Y is holding a popular index.

- Secondly, for ensuring at most $\frac{k}{2}$ hop routing, a link to peer X_{k+1} must be maintained. Since $X_{k+1} = \bar{X}$, the same link can be used for replication and routing purposes.

- Finally, as explained in Section 4.4.2, the 759 special octads in S_8 are likely to face a higher number of advertisements and queries than the 2576 umbral dodecads in U_{12}. On the other hand, $\mathscr{A}$ and $\mathcal{Q}$ do not contain any codeword from S_{16} (the 759 special 16-sets). Since, $X \in S_8 \implies \bar{X} \in S_{16}$, we can shed the extra load on $X \in S_8$ by replicating to $\bar{X} \in S_{16}$. Note that, $X \in U_{12} \implies \bar{X} \in U_{12}$

4.4.3.4 Algorithm for Multicast Routing

The routing algorithm will always route to a set of target peers instead of just one peer. A significant portion of routing hops can be reduced by utilizing the shared common paths to different targets as we will show later in Section 4.6.5. Algorithm 5 presents a pseudocode for multicasting a message from source peer X to a set of destination peers $\mathcal{Y} = \{Y_1, Y_2, \ldots, Y_u\}$.

The pseudocode presented in Algorithm 5 is a simplified version of the routing algorithm used in our simulator. It should be noted that the *msg* parameter contains a field named *msg.hops*, which is incremented at each hop. The routing of a message is suspended if *msg.hops* reaches a value of $\frac{k}{2} + 2$. Suppose, peer Y has failed, and a query targeted towards Y reaches one of its neighbors $Y_i (= Y \oplus g_i)$. Y_i can route the query to $\bar{Y}$ in two hops as $\bar{Y} = \bar{Y_i} \oplus g_i$, and $\bar{Y_i}$ is one hop away from Y_i. The maximum length of a path between any two peers is $\frac{k}{2}$. Hence, in the presence of failures, a maximum of

Algorithm 5 $X.route(msg, \mathcal{Y})$

Inputs:

msg: Message *e.g.*, search, advertise, join, *etc.*

$\mathcal{Y}$: $\{Y_1, Y_2, \ldots, Y_u\}$ set of target peers

Externals:

k: Dimension of the self-dual linear code

$X_1, \ldots, X_{k+1}$: $(k+1)$ neighbors of X {*see Equation (4.5)*}

{*update* $\mathcal{Y}$}

for each $Y_i \in \mathcal{Y}$ **do**

  if $X = Y_i \vee X = \bar{Y_i}$ **then**

    $\mathcal{Y} \leftarrow \mathcal{Y} - \{Y_i\}$

    process-message(msg)

  else if $d(X, Y_i) > \frac{k}{2} \bigvee$

      $(d(X, Y_i) = 1 \wedge !isAlive(Y_i))$ **then**

    $\mathcal{Y} \leftarrow (\mathcal{Y} - \{Y_i\}) \cup \{\bar{Y_i}\}$

  end if

end for

{*find suitability of each neighbor as next hop*}

$\mathscr{R} \leftarrow \{T_1, \ldots, T_{k+1} |\ \mathcal{T}_i \subseteq \mathcal{Y} \wedge$

    $Y \in \mathcal{T}_i \implies X_i$ is alive and on $X \rightsquigarrow Y\}$

{*do actual routing*}

while $\mathcal{Y}$ not empty **do**

  find s s.t. $\forall \mathcal{T}_i \in \mathscr{R}, |\mathcal{T}_s| \geq |\mathcal{T}_i|$

  if no such s exists **then**

    break {*remaining peers in* $\mathcal{Y}$ *are not reachable*}

  end if

  $\mathscr{R} \leftarrow \mathscr{R} - \{\mathcal{T}_s\}$

  $\mathcal{Y} \leftarrow \mathcal{Y} - \mathcal{T}_s$

  $X_s.route(msg, T_s)$

end while

$\frac{k}{2} + 2$ hops will be required to reach any peer or its replica.

4.5 Architecture of Plexus

In this section we present the architecture of Plexus and explain the protocols for topology maintenance.

4.5.1 Topology

From a functional point of view, peers in Plexus can be categorized as superpeers and leaf peers. In addition to the functionalities (mostly file transfer) carried out by the regular leaf peers, superpeers are responsible for indexing meta-information about the content published by the leaf peers and other superpeers, and for routing queries based on this information. Superpeers connect to a larger number of peers than the leaf peers, have higher capacity (bandwidth and processing power) and longer uptime. According to the classification presented in [22], Plexus has a *partially decentralized* architecture utilizing *structured search* in the superpeer network.

Partially decentralized systems, utilizing superpeers, are considered more practical solution ([144],[147]) due to their ability to accommodate peers with heterogeneous capabilities. These systems are less vulnerable to churn problem, as the oscillating population of leaf peers are kept at the edge of the logical overlay and the superpeers provide a relatively stable core. Superpeers population is much lower than the population of ordinary leaf peers. For these reasons we have chosen a two tier architecture and have considered Plexus to be the protocol for enabling structured and flexible routing within the superpeer network.

Figure 4.5 depicts the Plexus architecture. A superpeer indexes meta-information about the contents published by a number of other superpeers and leaf peers, and routes

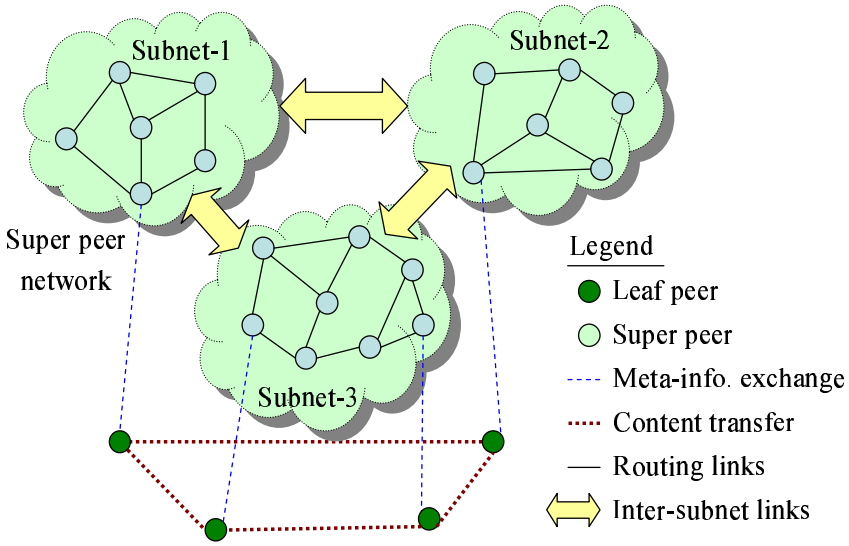


Figure 4.5: Architectural overview of Plexus.

queries based on this indexed information. Each superpeer is assigned one or more codewords. Due to the complexity of computing $\mathscr{A}(\mathrm{P})$ and $\mathcal{Q}(\mathrm{Q})$ we have to use small codes (*e.g.*, $\mathcal{G}_{24} \equiv (24, 12, 8)$). This restricts the number of superpeers to few thousands. On the other hand, we need about $100 \sim 200$-bits in a Bloom filter to encode about $20 \sim 30$ properties (*e.g.*, trigrams or property-value pairs) associated with a shared object. According to [133], the modern two-tier Gnutella network contains around a million peers of which around 18% are superpeers. To conform to these limitations and requirements we use a number of subnets or groups (around $6 \sim 10$) of superpeers. Each subnet uses different parts (bits) from an advertised or queried pattern and routes independently, as depicted in Figure 4.6 and explained in Section 4.4.3.

Figure 4.6 depicts the search/advertisement process in Plexus. In Figure 4.6, *step 1* and *step 2* are specific to the application under consideration, *e.g.*, keyword search, service discovery *etc.* The input to Plexus is a $(r \times n)$-bit pattern (in this example, a Bloom filter), representing an advertisement (or a query). Here, r is the number of subnets in the system and $n = 24$ as we have used $\mathcal{G}_{24}$. The input pattern is segregated

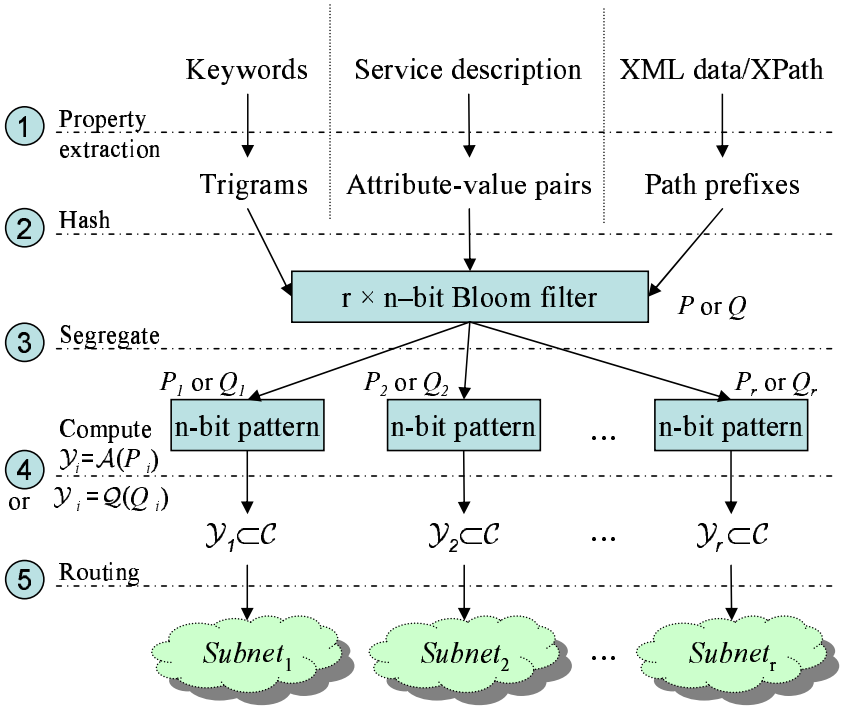


Figure 4.6: Search/advertisement process in Plexus.

into r chunks (*step 3*) P_i (or Q_i), each n bits long. In *step 4* an n-bit chunk is mapped to a set of codewords $\mathcal{Y}_i$ (*i.e.*, $\mathscr{A}(P_i)$ or $\mathcal{Q}(Q_i)$) and forwarded to any superpeer in subnet i. Finally in *step 5*, the superpeer routes the message in $O(\frac{1}{2}|\mathcal{Y}_i| \log_2 |C|)$ hops to the target superpeers in $\mathcal{Y}_i$ within subnet i. The process of mapping patterns to codewords (*Step 4*) is presented in Section 4.4.2 and the routing mechanism within a subnet (*Step 5*) is presented in Section 4.4.3.

If a query/advertisement message is propagated to all of the r-subnets, then the implied redundancy will be very high. Instead, we adopted the *Voting algorithm* [63]. In particular, an advertisement is propagated to $\lfloor \frac{r+1}{2} \rfloor$ subnets, and a query message is propagated to $\lfloor \frac{r}{2}+1 \rfloor$ subnets. This ensures that there exists at least one subnet receiving an advertisement, and any query matching that advertisement. The result of a query is computed as the union of the results obtained from each of the $\lfloor \frac{r}{2}+1 \rfloor$ subnets. $Subnet_i$ is selected for advertisement P (or query Q) if the weight of the i^{th} chunk *i.e.*, $|P_i|$ (or

$|Q_i|$) is within the query stretch as explained in Section 4.4.2.

Step 1 and *Step 2* in Figure 4.6 are always executed by a leaf peer. *Step 3* and *Step 4* can be executed either by a leaf peer or by a superpeer. We prefer the leaf peers to calculate the $\mathscr{A}_i$ and $\mathcal{Q}_i$, since these operations are CPU intensive. The leaf peer can then submit a message, containing the advertisement (P) or query pattern (Q) and the list of target codewords ($\{\mathcal{Y}_i\}$), to any known superpeer. It is the responsibility of a superpeer to maintain extra links to other superpeers outside its own subnet and to forward the message to appropriate subnets.

4.5.2 Mapping Codewords to Superpeers

So far we assumed that a superpeer is responsible for a unique codeword, which is not a practical assumption. In this section, we present a way of partitioning the codeword space, and dynamically assigning multiple codewords to a superpeer.

An (n, k, d) linear code $\mathcal{C}$ has k information bits and $(n - k)$ parity check (or redundant) bits. The k information bits correspond to the identity matrix (I_k) part of the generator matrix $G_{\mathcal{C}}$, and uniquely identifies each of the 2^k codewords present in $\mathcal{C}$ (consider $X = \vec{0}$ in Equation (4.6)). The codewords can be partitioned using a logical binary partitioning tree with height at most k. At i^{th} level of the tree, partitioning takes place based on the presence (or absence) of g_i (the i^{th} row of $G_{\mathcal{C}}$) in a codeword. Figure 4.7 presents an example. Each superpeer is assigned a leaf node in this logical tree and takes responsibility for all the codewords having that particular combination of g_is. The routing table entries at each superpeer are set to point to the appropriate superpeer responsible for the corresponding codeword. Figure 4.7 illustrates the routing table entries for superpeer $X = g_1 \oplus g_3 \oplus g_6 \oplus g_9$ with an equivalent prefix of $g_1\bar{g}_2g_3\bar{g}_4$. Here, $\bar{g}_i$ indicates the absence of the i^{th} row *i.e.*, g_i.

In order to incorporate the concept of partitioning codeword space in Algorithm 5,

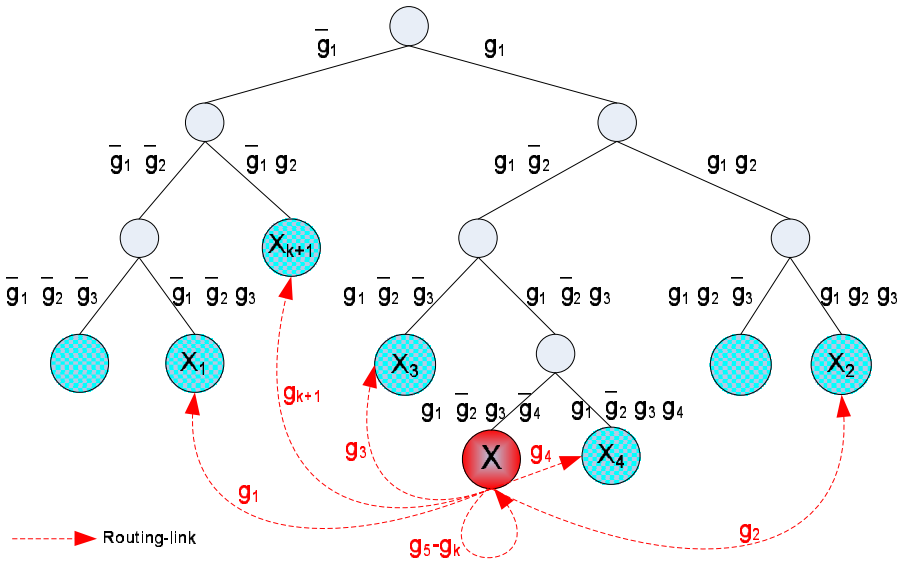


Figure 4.7: Logical binary partitioning tree for assigning codewords to superpeers. The routing table entries for peer X are also presented.

the comparison $(X = Y_i \vee X = \bar{Y_i})$ in line 4 should be replaced with $(Y_i \in \aleph \vee \bar{Y_i} \in \aleph)$, where $\aleph$ represents the set of codewords managed by peer X.

4.5.3 The Join Process

In this section we present the protocol that a superpeer has to follow in order to join a Plexus network. The first superpeer in the system begins with a random codeword, and all entries in its routing table point to itself. A new superpeer joins the system by taking over a part of the codeword space from an existing peer, say X. Assume that the string representation of $X = \rho_1 \cdot \rho_2 \ldots \rho_t \cdot \rho_{t+1} \ldots \rho_k$ and the prefix in peer X has t terms. Here, ρ_i is g_i or $\bar{g_i}$, based on the presence or absence of the i^{th} row in the formation of X. Peer X extends its prefix by one term and takes responsibility of all the codewords starting with prefix $\rho_1 \cdot \rho_2 \ldots \rho_t \cdot \rho_{t+1}$. The joining peer chooses a codeword, say Y, conforming to prefix $\rho_1 \cdot \rho_2 \ldots \rho_t \cdot \bar{\rho}_{t+1}$ and by selecting a random combination for the rest of the $(k - t - 1)$ rows from $G_{\mathcal{C}}$.

Routing table entries in peer X remain unchanged, except for adding Y as X_{t+1}. Peer Y can construct its routing table using the routing information from peer X. During this process, two situations can arise. First, the length of the prefix for peer X_i can be greater than t. In this case, Y has to lookup and contact the peer responsible for codeword $Y_i(= Y \oplus g_i)$. Peer Y requires at most 2 hops (see Theorem 4.5.1) to reach peer Y_i via peer X_i. For the second case, the length of the prefix for peer X_i is less than or equal to t. In this case, peer Y sets $Y_i = X_i$ and sends a join message to peer X_i. Peer X_i handles a join message by updating its routing table entry for link $X_t(= X_i \oplus g_t)$ with the address of peer X or peer Y depending on the presence of g_t in X_i.

Theorem 4.5.1. *If* $X(= \rho_1 \cdot \rho_2 \ldots \rho_t \cdot \rho_{t+1} \ldots \rho_k)$ *with* t *prefix bits is split to* $Y(= \rho_1 \cdot \rho_2 \ldots \rho_t \cdot \rho_{t+1} \ldots)$ *and* $Z(= \rho_1 \cdot \rho_2 \ldots \rho_t \cdot \bar{\rho}_{t+1} \ldots)$ *then all neighbors of* Y *and* Z *will be within* 2 *hops of* X.

Proof. Let, $\Psi_t(X)$ denotes the first t bits of X. Then $\Psi_{t+1}(Y)$ and $\Psi_{t+1}(Z)$ differ in exactly one bit. Again $\Psi_{t+1}(Y_i)$ and $\Psi_{t+1}(Z_i)$ differ from $\Psi_{t+1}(X_i)$ by at most 1 bit. Thus $\Psi_{t+1}(Y_i)$ and $\Psi_{t+1}(Z_i)$ will differ from $\Psi_{t+1}(X)$ by at most two bits, *i.e.*, at most 2 hops away.

□

To reduce the possibility of unbalanced partitioning of the codeword space, a joining peer should crawl the neighborhood of the seed peer, until a local minima is reached, and join the minima. By minima we refer to a peer having a prefix of length equal to or less than that of any of its neighbors.

4.5.4 Handling Peer Failure

The failure of a peer (say Y) does not hamper the routing process as long as its replica ($\bar{Y}$) is alive. This way temporary failures (or disconnections) of superpeers are automatically

handled. Measures adopted in Plexus to deal with permanent (long term) failures are discussed below.

Failure of peer Y will be detected by one of its neighbors, say Y_i. To avoid unbalanced partitioning of the codeword space, Y_i should crawl its neighborhood until a maxima, say Z, is reached. By maxima, we refer to a peer having a prefix of length equal to or greater than that of any of its neighbors. Clearly, if Z has t terms in its prefix, then $Z_t(= Z \oplus g_t)$ will be a neighbor of Z having a prefix of length t. Z will reassign its portion of codewords to Z_t; replace itself with Z_t from the routing tables of all of its neighbors; and finally rejoin the system as Y. Z_t has to reduce its prefix string by one, in order to accommodate the changes. In case Z_t has also failed then Z should start the recovery for Z_t first. To handle the failure of leaf peers, we adopt the hybrid (soft state/hard state) technique as presented in [27].

A leaf peer connects to a superpeer for publishing the meta-information about its shared content. A superpeer uses *soft-state* registration mechanism for tracking the failure of a leaf-peer, and explicitly removes (i.e., hard-state) the patterns, advertised by the failed leaf-peer, from the superpeer topology. This hybrid technique can handle churn problem in leaf peers and reduces traffic due to periodic re-advertisement in the superpeer network.

4.5.5 Analysis

In this section, we estimate the expected number of visited superpeers during an advertisement or a search process. Let r be the number of subnets in the system. Assume that an (n, k, d) linear code $\mathcal{C}$ is used. Let $|\mathscr{A}|$ and $|\mathcal{Q}|$ be the average size of $\mathscr{A}(\mathrm{P})$ and $\mathcal{Q}(\mathrm{Q})$, respectively. Let $\gamma_{\mathcal{Q}}$ and $\gamma_{\mathscr{A}}$ stand, respectively, for the fraction of routing hops reduced due to the presence of multicasting during search and advertisement. If N is the total number of superpeers in the system, then the expected number of superpeers in a

subnet is $2^k \approx \frac{N}{r}$. Now, according to Theorem 4.4.1 it will require at most $\frac{k}{2} \approx \frac{1}{2}\log_2 \frac{N}{r}$ hops to route a message within a subnet. Consequently, considering the use of the Voting algorithm as explained earlier, the expected number of hops required for routing an advertisement is computed to be

$$H_{\mathscr{A}} = \frac{1}{2}\left\lfloor \frac{r+1}{2} \right\rfloor (1-\gamma_{\mathscr{A}})\,|\mathscr{A}|\log_2 \frac{N}{r} \tag{4.9}$$

Similarly, the expected number of routing hops for routing a search message can be computed as

$$H_{\mathcal{Q}} = \frac{1}{2}\left\lfloor \frac{r}{2} + 1 \right\rfloor (1-\gamma_{\mathcal{Q}})\,|\mathcal{Q}|\log_2 \frac{N}{r} \tag{4.10}$$

For estimating $|\mathcal{Q}|$ we can adopt the extension of Johnson bound proposed by V. Guruswami and M. Sudan (Theorem 1 in [69]) as follows. Assume a $< n, k, d > f$ binary code (C). Let δ and γ ($0 < \delta, \gamma < 1$) be constants such that $d = \frac{1}{2}(1-\delta)n$, $t = \frac{1}{2}(1-\gamma)n$ and $\gamma > \sqrt{\delta}$, then it can be stated from Theorem 1 in [69] that

$$|\mathcal{Q}| = |B_t(Q) \cap \mathcal{C}| \leq min\{n, \frac{1-\delta}{\gamma^2 - \delta}\} \tag{4.11}$$

Estimating $|\mathscr{A}|$ as computed in Algorithm 4 is not a straight forward process. Rather we can compute an upper bound. Let $\mathcal{S}$ be the set of all subsets (Q) of an arbitrary pattern P such that, $|Q| \geq u$, where u is the minimum allowed weight of a query pattern. We compute $|\mathscr{A}|$ to be a subset of the codewords required to cover $\mathcal{S}$, with a covering radius f greater than the error correcting radius, say $e = \lfloor \frac{d}{2} - 1 \rfloor$, of the code. Hence, the number of codewords required to cover $\mathcal{S}$ with error correcting radius e will be an upper bound for $|\mathscr{A}|$.

Let $\mu = |P \wedge C|$ and $\eta = |P \wedge \overline{C}|$ for some codeword $C \in \mathcal{C}$. Evidently, $|P| = \eta + \mu$. We define query stretch s to be the difference between $|P|$ and the minimum weight of a query that we want to be discovered, *i.e.*, $s = \mu + \eta - u$. Now we can construct a query Q from P by taking i bits from the μ 1-bits of P (where C has 1-bits) and j bits from

the η 1-bits of P (where C has 0-bits) in $\binom{\mu}{i}\binom{\eta}{j}$ ways. Trivially, $d(P,Q) = i + j$ and $d(C,Q) = w + i - j$, where $w = d(P,C)$. In order for Q to be in $\mathcal{S}$, $d(P,Q) \leq s$ and $d(C,Q) \leq e$. Thus the total number of elements in $\mathcal{S}$ covered by C, say $\sigma(\mathcal{S},C)$ can be computed as

$$\sigma(\mathcal{S},C) = \sum_{i=0}^{\frac{1}{2}(e+s-w)} \sum_{j=0}^{\frac{1}{2}(s-e+w)} \binom{\mu}{i}\binom{\eta}{j}$$

We can compute $|\mathcal{S}| = \sum_{l=1}^{\mu+\eta-u} \binom{\mu+\eta}{l}$. Hence the upper bound on $|\mathscr{A}|$ can be computed as,

$$\begin{aligned} |\mathscr{A}| &\leq \frac{|\mathcal{S}|}{\sigma(\mathcal{S},C)} \\ \Rightarrow |\mathscr{A}| &\leq \frac{\sum_{l=1}^{\mu+\eta-u} \binom{\mu+\eta}{l}}{\sum_{i=0}^{\frac{1}{2}(e+s-w)} \sum_{j=0}^{\frac{1}{2}(s-e+w)} \binom{\mu}{i}\binom{\eta}{j}} \end{aligned} \tag{4.12}$$

For the experiments presented in this work we have used the Extended Golay code and have set the minimum query weight $u = 3$ and the maximum advertisement weight $v = max\{\mu + \eta\} = 14$. With this wide query stretch we obtained the average value of $|\mathscr{A}|$ and $|\mathcal{Q}|$ as 21.08 and 17.53 respectively for the dataset used in our experiments. Our experiments with the Extended Golay code reveal that the values of $\gamma_{\mathscr{A}}$ and $\gamma_{\mathcal{Q}}$ are proportional to $|\mathscr{A}|$ and $|\mathcal{Q}|$, respectively. For the average sizes of $|\mathscr{A}|$ and $|\mathcal{Q}|$, $\gamma_{\mathscr{A}}$ and $\gamma_{\mathcal{Q}}$ are 0.78 and 0.85, respectively (see Figure 4.12).

4.6 Experimental Evaluation

In this section we evaluate the effectiveness of Plexus protocol on three aspects of a music-sharing P2P system: routing efficiency, search completeness and fault-resilience.

4.6.1 Simulation Setup

We have simulated a growing network, where the overlay is gradually grown from an initial set of a few superpeers. The simulator goes through a growing phase and a

steady-state phase to achieve different network sizes. During the growing phase, arrival rate is higher than departure rate (up to five times). Once a target population size is reached the simulator turns to the steady state phase. While in steady state, arrival rate is approximately equal to departure rate and the network size does not vary a lot over time. During this period, advertisements and queries are performed and performance metrics like routing efficiency, search completeness *etc.* are measured.

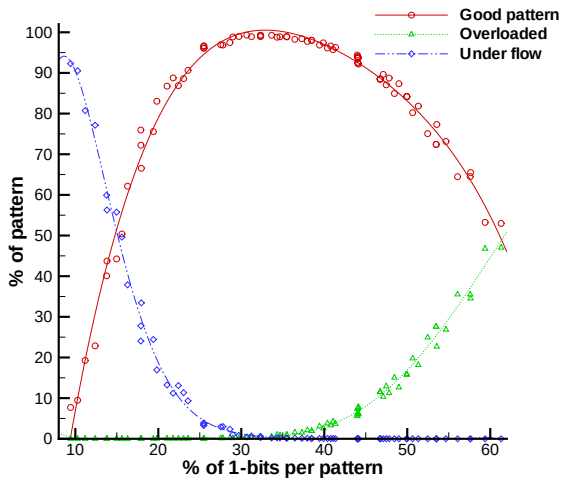
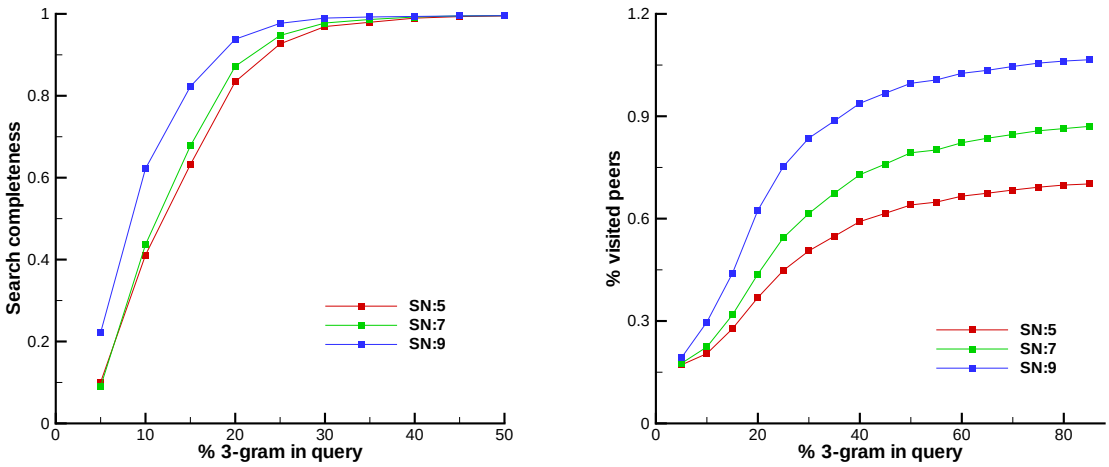


Figure 4.8: Fitness of patterns for advertisement

As explained in Figure 4.6, problems that can be mapped to distributed subset matching can be solved with Plexus. For this experiment we have applied Plexus to partial keyword search in music-sharing P2P system. The music information used in this simulation is based on the online database of more than 200,000 songs information available at *http://www.leoslyrics.com/*. For a <*song-title,artist*> pair, we constructed a Bloom-filter of length $m(= r * 24)$-bits. Here, r is the number of subnets in the network. A Bloom filter represents the set of trigrams extracted from a <*song-title,artist*> pair. We have experimented with three values of r: 5, 7 and 9; since m is dependent on r, the percentage of 1-bits in a pattern will vary if a fixed value for $\hbar$ (number of hash func-

tions) is used. To find the proper value of $\hbar$ we constructed Bloom filters for different values of r and $\hbar$. For each Bloom filter we constructed r 24-bit chunks and tested for fitness. A chunk is considered to be fit for advertisement if it contains $6 \sim 14$ 1-bits (see Section 4.4.2), and a $(r \times 24)$-bit pattern (*i.e.*, Bloom filter) is considered to be fit for advertisement if it contains at least $\lfloor \frac{r+1}{2} \rfloor$ fit chunks. Figure 4.8 plots the percentage of fit (or good) patterns as a function of the percentage of 1-bits. It also depicts the percentage of overflow patterns (majority of the chunks having more than 14-bits) and underflow patterns (majority of the chunks having less than 6-bits). Based on the peak values in the curve we have used $\hbar = 3, 4$ and 5 for $r = 5, 7$ and 9, respectively.

4.6.2 Impact of Query Content on Search Completeness



(a) Search completeness

(b) Impact on routing

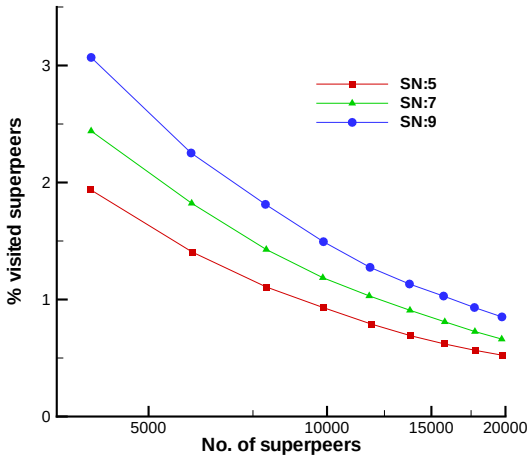
Figure 4.9: Effect of information content of a query on search completeness.

Search completeness is measured as the percentage of advertised patterns (matching the query pattern) that were discovered by the search. A query is formed as a Bloom

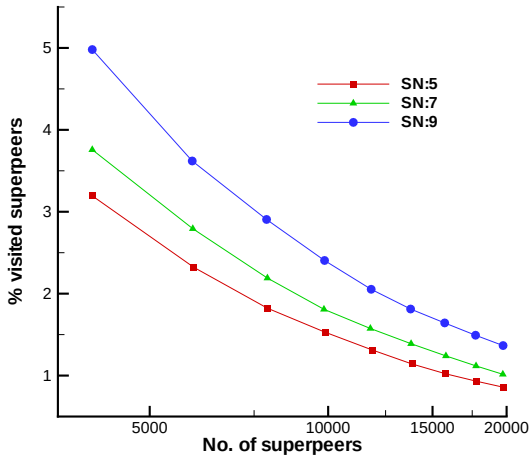
filter consisting of β% of trigrams (randomly chosen) from a <*song-title,artist*> pair. Figure 4.9(a) presents search completeness as a function of β, which is varied from $5\% - 50\%$ in 5% steps on networks of about 20,000 superpeers. For each step we performed 5000 queries. The number of 24-bit chunks having at least three 1-bits decreases as lower percentages of trigrams are taken from the original advertisement. Hence, read quorum for the *Voting algorithm* could not be met for lower values of β. This can also be observed from Figure 4.9(b), which is a plot of the percentage of visited peers against β. The sharp rise in the percentage of visited peers in Figure 4.9(b) justifies the increase in search completeness around $\beta = 30\%$ in Figure 4.9(a). It should also be noted that the percentage of visited peers is higher for higher values of r because the number of subnets to be searched is proportional to r. A completeness level of around 97% is achieved for $\beta = 33\%$. Only 2% increase in search completeness is achievable for $\beta > 33\%$, though at the expense of a higher percentage of visited peers. Therefore, we have used $\beta = 33\%$ in the subsequent experiments.

4.6.3 Scalability and Routing Efficiency

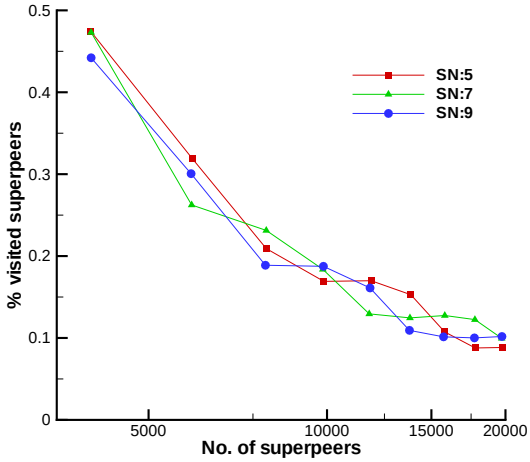
The impact of network size on routing efficiency and distribution of indexing load are considered in this section. Figure 4.10(a) and Figure 4.10(b) plot the average percentage of visited superpeers per search and advertisement, respectively, against the logarithm of the total number of superpeers in the network. The linear decrease in the curves confirms our assertion in Theorem 4.4.1, *i.e.*, number of visited peers per search and advertisement holds logarithmic relation with the total number of peers in the system. It should also be noted that the percentage of visited peers increases with the increase in the number of subnets (*i.e.*, r). For networks with fixed size (say N) and varying r (say r_1 and r_2), ratio of the percentage of visited peers can be calculated as $\frac{H_1}{H_2} \approx \frac{r_1 \log(N/r_1)}{r_2 \log N/r_2} \approx \frac{r_1}{r_2}$ (for small r_1 and r_2, see Equation (4.10)), *i.e.*, $H \propto r$. Similar results are obtained for advertisement



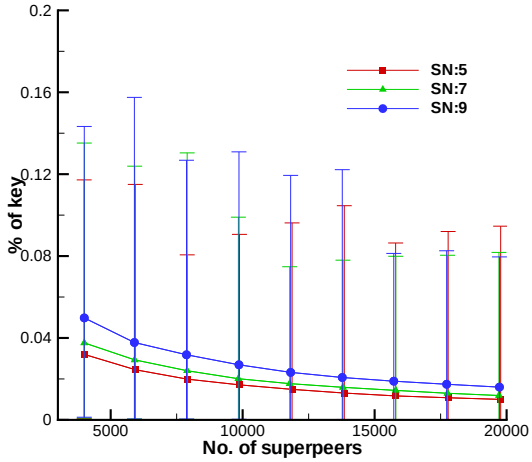
(a) Routing efficiency (search)



(b) Routing efficiency (advertisement)



(c) Join overhead



(d) Load distribution. average, 1^{st} and 99^{th} percentiles

Figure 4.10: Routing efficiency and scalability with network size.

traffic, as presented in Figure 4.10(b).

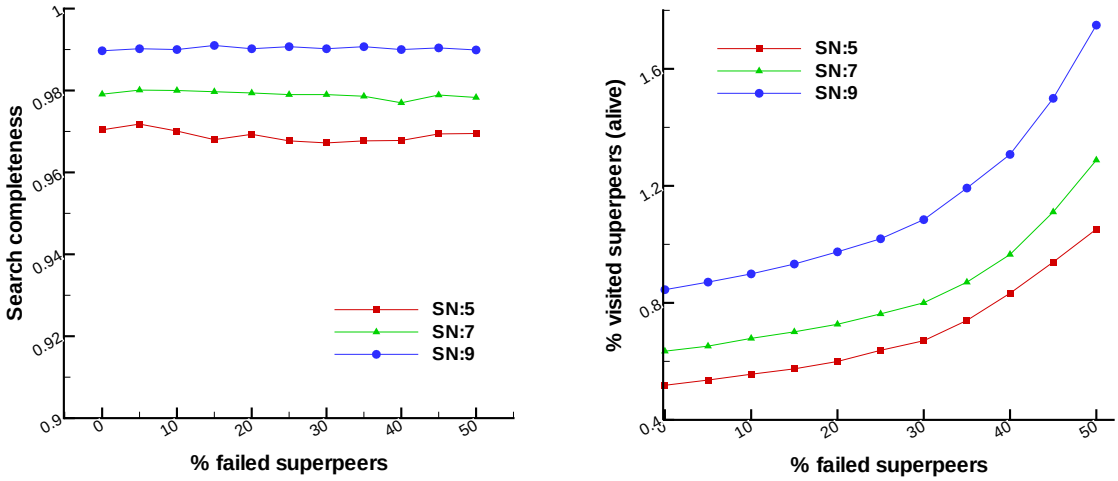
Figure 4.10(c) presents the percentage of visited peers for join operation as a function of network size. As discussed in Section 4.5.3 and reflected in Figure 4.10(c), the join process updates links only within a subnet and is thus independent of the number of subnets in the system. A join operation may require at most $O(\log_2 N)$ links (12 in this experiment since $\mathcal{G}_{24}$ is used) to be established. This justifies the decrease of the curves against the logarithm of network size.

Figure 4.10(d) presents the distribution of advertised patterns over the network. Average number of keys per pattern as well as 1^{st} and 99^{th} percentiles are presented. The skew in indexing load is higher for smaller networks and reduces gradually as the network size increases. Note that the average values of % *key per peer* in Figure 4.10(d) are very close to the expected values $= \frac{100}{N}$ (N is the network size) and the 99^{th}-percentiles are within reasonable limit.

4.6.4 Fault Tolerance

In this section, we analyze the robustness of Plexus in presence of simultaneous failures of a large number of superpeers. We start with a steady-state network of about 20,000 superpeers and cause each peer to fail with probability p. After the failures have occurred we perform 5000 queries and measure search completeness (Figure 4.11(a)) and the percentage of visited superpeers per query (Figure 4.11(b)). There were no rearrangement in topology to redistribute the responsibility of failed peers to an existing peer. Only the immediate neighbors of a failed peer have the knowledge of the failure. This setup suppresses the effect of recovery mechanism and allows us to observe the effectiveness of replication and multi-path routing in presence of simultaneous peer failures.

The number of replicas of an advertised pattern is proportional to r, thus much better search completeness is achieved for higher values of r. Failure of a peer cannot be detected



(a) Search completeness

(b) Query routing traffic

Figure 4.11: Fault resilience

until reaching a neighbor of the failed peer. The percentage of visited peers increases with *% failed peers* as many hops are wasted in trying to reach a failed peer and its replica, which may also have failed. However, the good thing is that in such cases two extra hops are required to reach the replica, as discussed in Section 4.4.3.

It can be observed from Figure 4.11(a) that search completeness is almost identical regardless of the percentage of failed peers (up to 50%). It should be noted that the query patterns where formed using 33% of the trigrams from existing patterns only; lost patterns due to the failure of a superpeer where not considered. The high levels of search completeness indicate that the superpeers remain reachable even in the presence of a large number of failures. This is possible because of the existence of multiple paths connecting any two superpeers within a subnet. However, this resilience to failure comes at an expense of increased routing overhead as observed in Figure 4.11(b).

4.6.5 Effectiveness of Multicast-routing

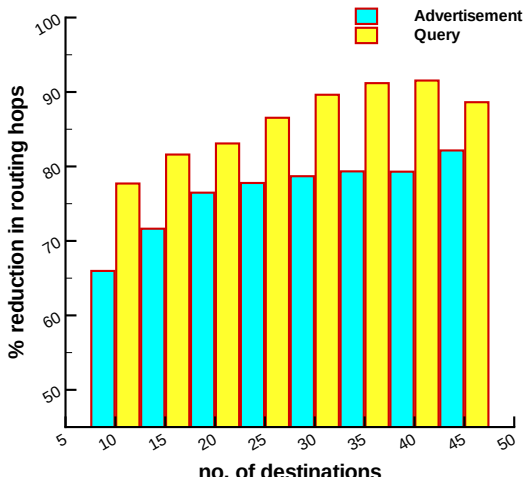


Figure 4.12: Effectiveness of simultaneous routing to multiple targets: reduction in routing hops as a function of the number of targets.

Network distance of the codewords in $\mathscr{A}(\mathrm{P})$ and $\mathcal{Q}(\mathrm{Q})$ have significant impact on routing efficiency. The routing algorithm described in Section 4.4.3 routes a message to multiple targets simultaneously. This design choice saves a portion of the routing hops that might have occurred if we had used pair-wise routing. Figure 4.12 shows the reduction in routing hops (γ) calculated as

$$\gamma = \left(1 - \frac{\text{no. of hops with multicast routing}}{\text{no. of hops for pair-wise routing}}\right) \times 100 \tag{4.13}$$

As defined in Section 4.5.5, Equation (4.13) provides an estimate of the reduction in routing hops for a query ($\gamma_{\mathcal{Q}}$) or an advertisement ($\gamma_{\mathscr{A}}$). The reduction in routing hops takes place within a subnet and hence does not depend on the number of subnets present in the system. The bar chart in Figure 4.12 displays the average γ for groups of 5 targets, *i.e.*, $6-10$, $11-15$, *etc.* The denominator for Equation (4.13) has been calculated as: $\sum_{Y \in \mathcal{Y}_i} d(\Psi(X), \Psi(Y))$, where X is the source peer, $\Psi(X)$ returns the k-bits of a codeword corresponding to the I_k part of $G_{\mathcal{C}}$, and $\mathcal{Y}_i$ is the set of target peers

calculated as $\mathscr{A}(P_i)$ or $\mathcal{Q}(Q_i)$ (see Figure 4.6). By observing the high values of γ, it can be inferred that the codewords close in Hamming distance are assigned to superpeers in close vicinity within the overlay.

4.7 Summary

Plexus has a *partially decentralized* architecture utilizing *structured search*. To our knowledge, the use of coding theoretic constructs in P2P routing is novel. As demonstrated by the simulation results, for a network of about 20,000 superpeers, Plexus needs to visit only $0.7\% \sim 1\%$ of the superpeers to resolve a query and can discover about $97\% \sim 99\%$ of the advertisements matching the query. For achieving this level of completeness, the query needs to contain only 33% of the trigrams from an advertisement that it should match against. Plexus delivers a high level of fault-resilience by using replication and redundant routing paths. Even with 50% failed superpeers, Plexus can attain a high level of search completeness (about $97\% \sim 99\%$) by visiting only $1.4\% \sim 2\%$ of the superpeers. Plexus can route queries and advertisements to target peers in $O(logN)$ hops and by using $O(logN)$ links.

The originality of our approach lies in the application of coding theoretic construct for solving the subset matching problem in distributed systems. We believe that this concept will aid in solving a number of other problems pertaining to P2P networking research, including P2P databases, P2P semantic search and P2P information retrieval.

Chapter 5

Comparative Evaluation

5.1 Introduction

In Section 3.13 and Section 4.6 we presented experimental results for assessing the performance of DPMS and Plexus, respectively. In this section we will compare the performance of DPMS and Plexus against each other and three dominant P2P search techniques. As representatives for *content routing*, *unstructured* search techniques we will use *Flooding* and *Random-walk*. We will use a generic *inverted indexing* mechanism on top of *Chord* as a representative for *address routing*, *structured* techniques. We have chosen the performance metrics to reflect the ability of each of these techniques in fulfilling the search requirements in LSDS as explained in Section 2.5.

5.2 Chapter Organization

In Section 5.3 we investigate existing P2P simulators and explain the components within the PeerSim simulator. In that section we also highlight the experimental dataset. The search techniques used for this experiment are presented in Section 5.4. The performance metrics used for the comparison have been explained in Section 5.5. Finally in Section 5.6

we present the performance results and discuss our findings.

5.3 Simulation Setup

In this section we explain the simulation environment. First we focus on the available P2P simulators in Section 5.3.1. Then we present the simulator (*i.e.*, PeerSim) that we have used for the simulations in Section 5.3.2. Finally in Section 5.3.3 we present the characteristics of the dataset used in these experiments.

5.3.1 P2P Simulators

Peer-to-peer technology has been on the scene for a while. But the P2P research community has not come to a consensus on standardizing a simulation platform for simulating the research projects from numerous working groups. It can be observed from the survey presented in [110] that more than 90% of P2P research was tested in non-standard or custom-made simulation environments. Yet there are a number of freely available P2P-simulators on the Internet. Most of these simulators are still at the early stages of implementation; it will take a while for these simulators to achieve maturity.

In this section we will focus on few P2P simulators. A comprehensive survey on P2P simulators can be found in [110]. NS2 [9] is the defacto standard for simulating network protocols. But an overlay simulator focuses on the application layer whereas NS2 and other network simulators concentrate on packet-level performance metrics. Moreover, an overlay simulator needs to simulate much larger networks compared to traditional network simulators. The majority of overlay protocols ignore the physical network topology. In these cases, simulating the entire TCP/IP protocol stack is not required. For this reason most overlay simulators bypass the simulation of the underlying TCP/IP protocol stack.

Over the last few years many P2P simulators have been proposed. But most of them

are aimed at simulating a specific type of P2P architecture, rather than being generic. For example, P2PRealm [95] is focused on simulating neural network based P2P protocols, PLP2P [74] is a packet-level simulator on top of NS2, and SimP^2 [89] is designed for simulating ad-hoc P2P networks. For our experiments we have used PeerSim, which is described in detail in the next section. In the rest of this section we present few P2P simulators that we considered as possible alternatives for PeerSim. We also present their relative merits and demerits.

PlanetSim [60] is a discrete-event simulator for overlay networks, written in Java. Architecturally PlanetSim is composed of three layers:

- *Network layer* models the overlay topology and network characteristics *e.g.*, proximity, failure *etc.*
- *Overlay layer* provides abstract classes that can be extended for defining overlay routing protocols like Chord, CAN *etc.*
- *Application layer* provides API (Application Programming Interface) for implementing services like file-sharing, keyword search *etc.* on top of the overlay routing mechanism.

Each of these layers expose *Common API* [51] routines, proposed for standardizing the structured P2P networks. For our implementation this simulator has two major drawbacks:

- PlanetSim is focused on structured P2P networks. The source package is shipped with Chord and Symphony [104] implementations only.
- Mechanisms for gathering statistics from the simulation runs are limited and premature in PlanetSim.

GPS [149] is a discrete event driven, message-level, P2P simulator. It has a built-in implementation of the BitTorrent [43] protocol and it has provision for implementing structured and unstructured search techniques. Unlike other simulators GPS models file transfer mechanism, which is the main focus of BitTorrent architecture. The main disadvantages with this simulator are as follows:

- The simulator offers limited functionality for implementing protocols other than BitTorrent.
- The API documentation is not comprehensive.

P2PSim [64] is multi-threaded P2P simulator developed at MIT (Massachusetts Institute of Technology). P2PSim has been designed to simulate structured P2P networks only. A number of structured routing protocols including Chord, Accordion [98], Koorde [86], Kelips [68], Tapestry [150], and Kademlia [105] have been implemented in this simulator. Yet the lack of support for simulating unstructured or semi-structured P2P routing protocols, makes it unusable for our purpose.

3LS [139] is 3 layer P2P simulator. Similar to PlanetSim, it has *Network layer*, *Overlay layer* and *User layer*. The network layer maintains a two dimensional matrix for storing the distance between any two node. The overlay layer is responsible for the protocol being simulated, while the user layer defines an input interface for the user. The major drawback of 3LS is scalability: memory overhead incurred by the Network layer and the GUI (Graphical User Interface) is too high, which restricts the simulated network's size to a couple of thousand peers on a regular machine.

Query-Cycle Simulator [126] focuses on unstructured routing mechanisms. A simulation run is designed to be a collection of *Query-Cycle*s. A Query-Cycle comprises the following steps: (a) a peer initiates a query, (b) the overlay network routes the query to

the peer(s) that can respond to the query, (c) the target peer(s) returns results to the peer that initiated the query, and (d) the actual file-transfer is completed. Multiple queries can be executed in parallel. The simulator has built-in models for content-distribution, peer behavior, and network topology based on the studies conducted on Gnutella network. The drawback of this simulator lies in its inability to incorporate structured P2P networks.

NeuroGrid [84] is a discrete-event P2P simulator primarily designed for evaluating unstructured search techniques. The distribution is packaged with Freenet [42], Gnutella [4] and Neurogrid [83] protocols. Only the overlay layer can be simulated with this simulator. It assumes a graph topology as input to the simulator. The current implementation does not provide support for simulating churn. Statistics information is available for a set of predefined variables and additional coding is required to incorporate new statistics.

5.3.2 PeerSim

We have used PeerSim for our experiments. PeerSim is an open source, general purpose P2P simulator, written in Java. It offers both cycle-driven simulation and discrete-event simulation using separate simulation engines. It has been designed to be scalable and dynamic for simulating large P2P networks. Both structured and unstructured P2P networks can be simulated using PeerSim. Like other P2P simulators it does not consider the underlying network communication stack for monitoring network layer performance, rather the focus is on the overlay layer. It is possible to extend the simulator to incorporate network layer characteristics.

Implemented Protocols

Initially PeerSim was developed as a part of the BISON project [2]. There exists a number of protocols implemented in PeerSim, including:

- **Overstat** [81]: an aggregation tool for providing statistical information, such as average load in a distributed system.
- **SG-1** [109]: a protocol for constructing and maintaining superpeer based topologies.
- **T-Man** [80]: a network topology generator that allows a wide range of network topologies to be generated.
- **SCALER** [71]: a protocol for generating artificial social networks over P2P overlay.

In addition a number of recent research works including [66, 25, 48, 67] have used PeerSim as their simulation platform.

Advantages of Using PeerSim

The most attractive properties of PeerSim are given below:

- **Simplicity**: It defines a handful of Java interfaces through which user defined java-components can be plugged into the simulator.
- **Extendability**: These java-components can be used for defining and monitoring overlay topology, routing mechanism, join/leave protocol, replication strategy *etc.* The use of Java component based architecture makes it possible to replace most of the predefined simulator components with user-defined components.
- **Configurability**: The simulator dynamically loads java-components, specified in a configuration file, at startup. This allows different types of simulations to be

performed without recompiling the code. The configuration file is a simple ASCII file containing key-value pairs.

- **Support**: The distribution is accompanied with a number of reusable components for topology generation, modelling churn and network growth, *etc.* The number of available components is growing rapidly with the contributions from various research groups. These contributions are regularly posted at the PeerSim homepage.

PeerSim Architecture

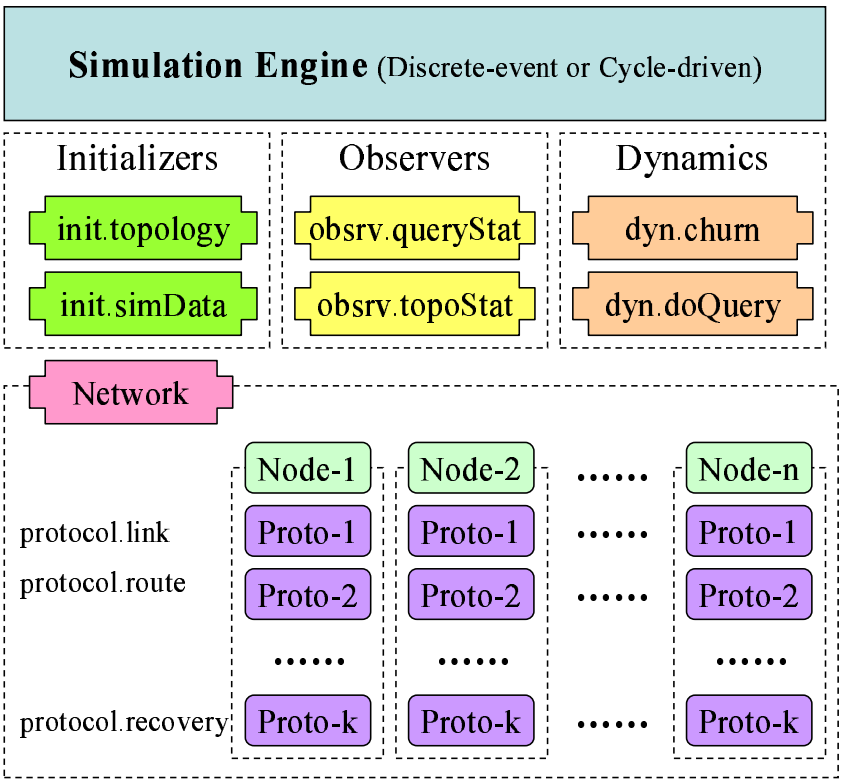


Figure 5.1: Architectural components of PeerSim

Figure 5.1 presents the components in PeerSim architecture and logical implementation of these components in our simulation study. Each of these components is specified using a configuration file, and loaded and configured in runtime by the simulation engine.

We have used the cycle-driven simulation engine. Functionality and purpose of each of these components and the example implementations are given below.

- **Simulation engine** wraps the core scheduling mechanism and initiates the loading of the configuration file and the components specified therein. Separate engines are provided for event-driven and cycle-driven simulations.

- **Observer** components have a global view of the network and are designed to gather statistical information regarding system performance. Any number of *Observer* objects can be defined. In Figure 5.1 two observer objects have been defined: *obsrv.queryStat* is for gathering statistics on query routing performance and *obsrv.topoStat* is defined for gathering topology specific statistics like average node degree, percentage of failed peers *etc.*

- **Dynamics** components, like Observers, have a global view of the network and are used for introducing dynamism in overlay topology. There is no limit on the number of *Dynamics* object that can be used in a simulation. In the example of Figure 5.1, two *Dynamics* objects have been defined: *dyn.churn* is used for simulating random arrival and failure of peers at a certain rate, and *dyn.doQuery* is used to initiate queries during the simulation process.

- **Initializer** is a special type of *Dynamics* component that is executed only at the beginning of the simulation. We have used two initializer objects for the simulation process: **(a)** *init.topology* has been used for initializing overlay topology and **(b)** *init.simData* has been used for initializing advertisement and query data like the source peer, start time, keyword list *etc.*

- **Network** is one of the core components of the simulator and is responsible for managing the list of all *Node*s in the system. The *Network* object may have any number of *Node* objects.

- **Node** is the wrapper for a peer functionality and its interaction with other peers in the network. Each node contains a set of protocols that are invoked by the simulation engine at each cycle. A *Node* instance may contain any number of *Protocol* instances. In addition, our implementation of the *Node* interface contains incoming messages queue, advertisement storage and seen messages filter for blocking recursive loops.

- **Protocol** interface defines the point of invocation for the simulation engine. A developer have to write code for this interface for simulating various routing or application level functionality. Each protocol instance uses the local knowledge (or view) available to the peer. As shown in Figure 5.1, we used *protocol.link* as a container for the overlay connectivity (*i.e.*, neighborhood view). *protocol.route* is the routing mechanism specific to the search technique being examined (*e.g.*, Chord, Flooding *etc.*). *protocol.recovery* is for the implementation of a recovery mechanism local to the peer.

5.3.3 Experimental Dataset

The dataset used for this experiment is the same as the one used in Section 3.13 and Section 4.6. Here we present detailed statistics regarding the dataset.

It has been measured in [133] and [134] that at least 66-70% of the content in the modern Gnutella network are music files. It has been identified in [107] and [131] that majority of P2P queries are for audio files. Hence we have used the database of song information for this experiment. We compiled this dataset by crawling the webpages at *http://www.leoslyrics.com/* [6], which is an online database of over 200,000 song lyrics. We created the crawler using simple AWK-scripts and the GNU WGET utility [11]. We extracted the *song-title* and *artist-name* information from the crawled pages and created the database as a list of tuples. *Album* and *genre* information were deliberately ignored

because of two reasons: (a) in many cases these two pieces of information are missing and (b) the possible values of these two fields are much smaller than the *song title* or *author* fields. The later would introduce additional bias to the resulting Bloom filters. The effect would be better aggregation level for DPMS and improved routing performance.

For enabling subset matching on DPMS, Plexus and Chord we have used 3-grams collected from each <*songtitle, artist*> tuple. Performance of these systems are related to the average number of 3-grams generated from the <*songtitle, artist*> tuples. The average number of 3-grams per pair was found to be 29.37 for the experimental dataset.

5.4 Compared Search Techniques

The search techniques considered for this comparative evaluation are as follows:

1. Flooding
2. Random-walk
3. II/DHT (Inverted Indexing over Chord routing)
4. DPMS
5. Plexus

Table 5.1 enumerates the system specific parameters for each of these search techniques and the values used in this experiment. In the following sections we explain each of these parameters along with the associated search technique.

5.4.1 Flooding and Random-walk

Unstructured search techniques like *Flooding* and *Random-walk* do not place any restriction on the underlying network topology. For these two cases we have adopted the

Table 5.1: Simulation parameters for comparative evaluation

	Param	Value	Description
Flooding	α	15	Topology parameter controlling node density
	Δ	12	Topology parameter for average node degree
	R_F	120	Average replica per advertisement
	TTL_F	4	Time to live
Random-walk	α	15	Topology parameter controlling node density
	Δ	12	Topology parameter for average node degree
	R_R	120	Average replica per advertisement
	TTL_R	10	Time to live
	W_R	15	Number of walkers
II/DHT	R_C	4	Replica per key
	X	Chord	DHT routing protocol
DPMS	H	5	Height of indexing hierarchy
	R_D	2	Recursive replication factor
	B	4-6	Branching factor
	A	0.6	Aggregation ratio
	O	50	Minimum number of non-X bits
	W	180	Pattern width
	$\hbar_D$	3	No. of hash functions in Bloom filter
Plexus	r	7	Number of subnets
	$\hbar_P$	3	No. of hash functions in Bloom filter
	u	3	Minimum query weight per chunk
	v	14	Maximum advertisement weight per chunk

network model proposed in [53], which is used for constructing power-law networks [54], mimicking the characteristics of the contemporary P2P topologies on the Internet. This topology generator comes with the PeerSim distribution package. Parameters α and Δ in Table 5.1 are used by the topology generator routine. We have chosen the values of these two parameters based on the observations in [53] and [54].

Search completeness in unstructured techniques depends heavily on the popularity of the content being searched. For these techniques the probability of discovering a popular (hence highly replicated) object is higher than an unpopular item. In [131], it has been shown that object popularity follows a Zipf-like distribution in Gnutella networks. But in [45] it has been shown that both uniform and proportional (to query popularity) replication strategies are equivalent in terms of search completeness. So we have used uniform replication for Flooding and Random-walk techniques. Based on the observations in [29] we have used an average replication factor (R_F and R_R) of 120 per advertisement.

The Time-to-live parameter (TTL_F and TTL_R) can be used to trade of search traffic with search completeness. Search completeness increases with the increase in TTL but at the expense of increased search traffic, and vice versa. However, the effect of TTL_F is much higher in Flooding than that in Random-walk (*i.e.*, TTL_R).

5.4.2 II/DHT

The DPM problem can be solved using content routing and signature routing techniques. Plexus, on the other hand, is the only known address routing technique that can solve the DPM problem. Thus, it is not possible to compare Plexus with other structured techniques if we try to solve the DPM problem. Instead, we have compared the performance of different search techniques *w.r.t.* an application of the DPM problem; here partial keyword matching.

We have used Chord as a representative for the DHT routing protocols. To enable partial keyword matching on top of Chord we adopted the simple inverted indexing strategy as used in [72] (for partial keyword matching), [102] (for multiple keyword matching) and [27] (for partial service description matching). In brief, each keyword is broken into 3-grams, which are then hashed and routed to the set of responsible peers. The generated volume of advertisement traffic is pretty high for the simple inverted indexing mechanism. There exists few proposals for reducing advertisement traffic by adopting specialized hash functions like Locality Preserving Hashing (LPH) as used in Squid [127] and Fingerprint function as used in [31]. These optimizations support prefix matching only but we are interested here in partial (or infix) matching.

For improving fault resilience the advertisement is replicated at (R_C) peers along the Chord ring following the responsible peer. The value of R_C has been chosen based on the suggestions in [27].

5.4.3 DPMS

As presented in Chapter 3, DPMS [16] uses a hierarchical topology and indexing structure. Lossy aggregation is used for controlling the index volume at higher level peers. Recursive replication along the indexing hierarchy is adopted for increasing fault-resilience and load distribution.

In this experiment we have an indexing hierarchy of 5 indexing levels. Based on the discussion in Section 3.13.3 we have set the recursive replication factor (R_D) to 2. The allowed range for branching factor (B) has been set to $4 - 6$. The values of aggregation ratio A, pattern width (W) and number of hash function $\hbar_D$ are according to the findings from our previous experimentation with DPMS, as presented in Section 3.13.

5.4.4 Plexus

Plexus is the only structured search technique that uses signature routing and can efficiently solve the DPM problem. For this experiment we have selected the parameter values based on our previous experiments with Plexus in Section 4.6. For this experiment we have used a Plexus network with $r = 7$ subnets and $\hbar_P = 3$ hash functions based on the curve in Figure 4.8. The query stretch $s = v - u$ has been set according to the discussion in Section 4.4.2.

5.5 Performance Metrics

Topology Maintenance Overhead

Topology maintenance overhead largely determines the level of peer dynamism that a search technique can bear. In general this overhead is larger in structured techniques than unstructured or semi-structured cases. This causes structured techniques to be more sensitive to churn in network.

As a measure of topology maintenance overhead we will look into two metrics:

- **Average degree** is the average number of links maintained by a peer. Topology maintenance overhead is directly proportional to this metric, especially for structured techniques that require a peer to connect to distinguished peers on the overlay.

- **Join overhead** is measured as the percentage of peers that a joining peer has to communicate for setting up its own routing table and the routing tables of its neighbors on the overlay. This metric should be small, otherwise the system will become unstable in presence of churn.

Advertisement Efficiency

In order to handle content dynamism, the advertisement or index registration process should be efficient in terms of storage and network usage. We can have an estimate of storage and network overhead using the following two metrics, respectively.

- **Replication factor** is defined to be the percentage of peers containing a given advertisement (*i.e.*, index or content). Update propagation traffic is proportional to average replication factor.

- **Advertisement traffic** is measured as the percentage of peers that had to be visited for registering an advertisement. We used percentage of visited peer instead of number of hops or number of messages, because the later two metrics are implementation dependent, *i.e.*, can be optimized using smart implementation techniques.

Search Efficiency

Search efficiency deals with two questions: (a) what percentage of matches are discovered by a search? and (b) how much bandwidth is consumed by the search? In most systems search efficiency is more important than advertisement efficiency, as more search is performed than advertisements. As a measure of search efficiency we will investigate the following two quantities:

- **Search completeness** answers the first question: what percentage of matches are discovered by a search? We measure this quantity as the expected ratio of discovered, distinct advertisements matching the query to the total number of distinct matching advertisements available in the system. We do not consider the replicas of an advertisement in this measure and this is the norm adopted for unstructured

techniques. Mathematically,

$$Search\,completeness = E\left(\frac{no.\,of\,discovered\,distinct\,matches}{available\,matches\,(distinct)} \times 100\right)$$

- **Search traffic** answers the second question: how much bandwidth is consumed by the search? Similar to the measurement of advertisement traffic, search traffic is measured as the percentage of peers that had to be visited for satisfying the search.

Evidently, a good search mechanism should incur low overhead on the network and should discover all of the matching advertisements.

Search Flexibility

One of the major targets of DPMS and Plexus is to discover advertisements using only partial information. Search flexibility refers to the ability of a search system to discover matching advertisements using partial information. Similar to the measure of search efficiency, we use search completeness and search traffic to measure search flexibility. However, for search efficiency we measured these two quantities for various network sizes. On the contrary, for assessing search flexibility we take a network of a fixed size and measure these two quantities for different levels of information content in the query string.

Fault Resilience

Churn or peer dynamism is common in any LSDS. The ability of a search technique to cope up with churn largely depends on the recovery mechanism, which in turn is implementation dependent, *e.g.*, keep alive message exchange rate, keeping tract of alternate neighbors *etc.* But any search technique in LSDS is expected to perform smoothly in presence of peer failures. As new peers join the system routing tables of the existing

peers are updated overtime by the recovery mechanism. But we want to measure search efficiency in presence of failure and before any recovery has taken place. For this purpose we simultaneously make the peers to fail with certain probability and measure the search efficiency. Even in the presence of failures, a good search technique is expected to return complete search results (*i.e.*, discover all matching advertisements available in the system), while the search traffic should not increase significantly.

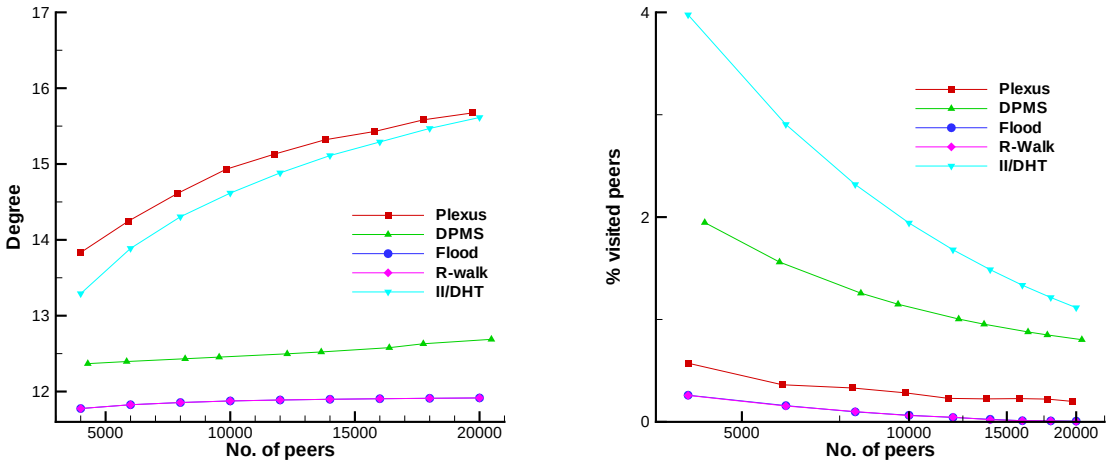
5.6 Simulation results

The rest of the section presents relative performance of the five search techniques *w.r.t.* the performance metrics presented in Section 5.5.

5.6.1 Topology Maintenance Overhead

Topology maintenance overhead is dependent on the average degree of the peers. Topology maintenance overhead is usually lesser for a smaller value of average node degree. Order of node degree in Plexus is $O(\lg \frac{N}{r} + r)$ and for Chord it is $O(\lg N)$. For Flooding and Random walk we have used the internet topology generation model, in which the average degree is governed by topology parameter Δ. Finally, in DPMS node degree is $(B + R_D + \vartheta)$, where ϑ is the cardinality of neighborhood-list used by the Newscast protocol [143]. As reflected in Figure 5.2(a), the average degree depends on network size for Plexus and DHT-technique, whereas it is constant for unstructured or semi-structured techniques.

Minimizing peer join overhead is crucial for handling peer dynamism in large scale distributed systems. It is mostly governed by the protocol used to define the logical overlay topology. Join overhead is minimum for unstructured systems, as in this case the only rule for join protocol is to keep the degree below a maximum value. In Plexus, a



(a) Average degree

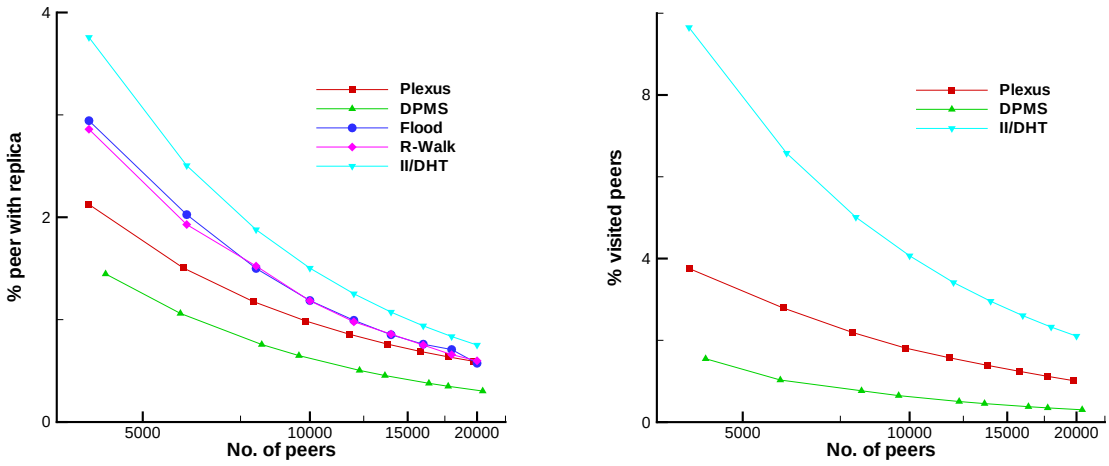
(b) Join overhead

Figure 5.2: Topology maintenance overhead.

joining peer has to locate $O(\lg \frac{N}{r})$ neighbors, all of which are located within two hops of the seed peer (see Theorem 4.5.1), while the r external links can be obtained from the seed and the neighbors. In essence, the effect of joining is localized in Plexus and therefore the low overhead. The higher overhead of peer join in DPMS results from the strict rules of connectivity within the indexing hierarchy. Chord, and in general DHT-techniques, exhibit higher join overhead because of the global impact of initialized routing links. In Chord, the joining peer becomes a predecessor of the seed peer. The joining peer cannot benefit from the seed peer's routing table because of the unidirectional nature of Chord routing (see [87]). In summary, the average degree in Plexus is close to that in DHT-techniques though the join overhead is much less. DPMS has low average degree and join overhead is moderate.

5.6.2 Advertisement Efficiency

Replication is used by search techniques for different purposes. Unstructured and semi-structured techniques use content replication for improving availability, structured techniques use index replication for improving fault-resilience, and extensions on DHT-techniques (like Twine [27] and Squid [127]) use index replication for achieving search completeness.



(a) Replication factor

(b) Advertisement traffic

Figure 5.3: Advertisement efficiency.

Figure 5.3(a) presents the average replication factor. In Plexus, the expected number of replica for some advertisement, say P, can be calculated as $|\mathscr{A}(P)|\lfloor\frac{r+1}{2}\rfloor$. Here, $|\mathscr{A}(P)|$ depends on the Error Correcting Code (ECC) in use and query stretch. In DPMS, the replication factor depends on H and R_D (see Table 5.1), and not on network size. It can be calculated as $\sum_{i=1}^{H-1} R_D^i$. For Flooding and Random-walk we have used uniform replication with an average of 120. In Chord, replication overhead depends on R_C and the average number of 3-grams per advertisement, which is 29.37 for the experimental dataset.

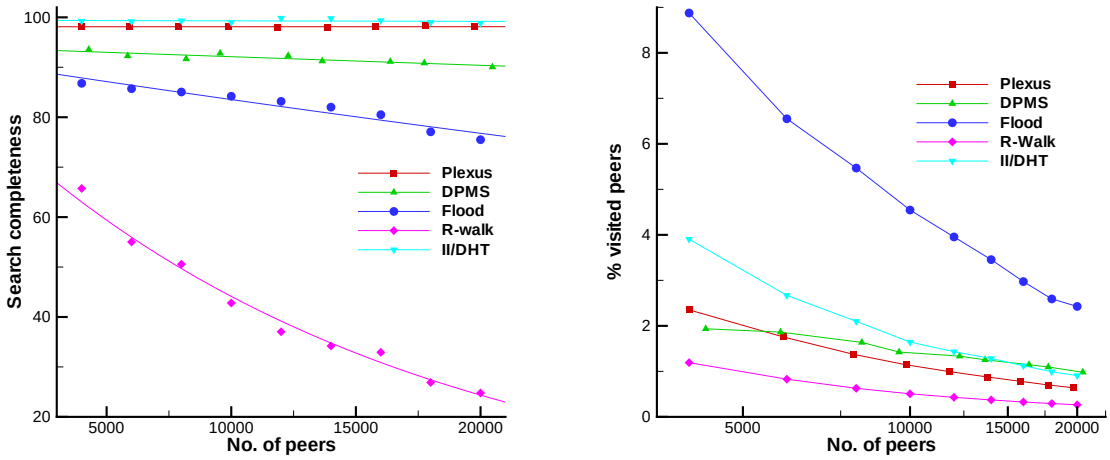
Though DPMS exhibits lower level of replication factor than Plexus, the index update cost is higher in DPMS because the aggregation algorithm does not allow incremental updates.

Figure 5.3(b) plots the average advertisement traffic. Advertisement traffic for Flooding and Random walk have not been presented, as in these two cases there exists no explicit advertisements; rather, content information is propagated as a result of the search process. For these two cases, we assume that the content information is well replicated before the search process begins. Advertisement traffic is low in DPMS because it uses bulk advertisement. II/DHT requires many independent DHT lookups to register the 3-grams with the Chord ring. Plexus, on the contrary, exploits multicast routing and the presence of closeness (in hamming distance) within the target codewords to reduce advertisement traffic.

5.6.3 Search Efficiency

In this experiment we simulated networks of different sizes and measured two aspects of search efficiency : search completeness (Figure 5.4(a)) and generated traffic (Figure 5.4(b)). A query has been constructed using a random 35% 3-gram from a randomly chosen advertisement. Flooding and Random-walk yields the two extremes in terms of search traffic. In spite of visiting a large percentage of peers, search completeness in Flooding is low. On the other extreme, Random-walk generates the least traffic and lowest level of search completeness.

Unlike structured techniques DPMS does not assign indexes to peers. Advertisements matching a given query cannot be found in any predefined (or computable) set of peers, rather a user has to search additional peers for discovering each match. In the experiment with DPMS we have stopped searching after 20 matches were found. Better search completeness is achievable by increasing this termination criteria (above 20), though at



(a) Search completeness

(b) Query traffic

Figure 5.4: Search efficiency.

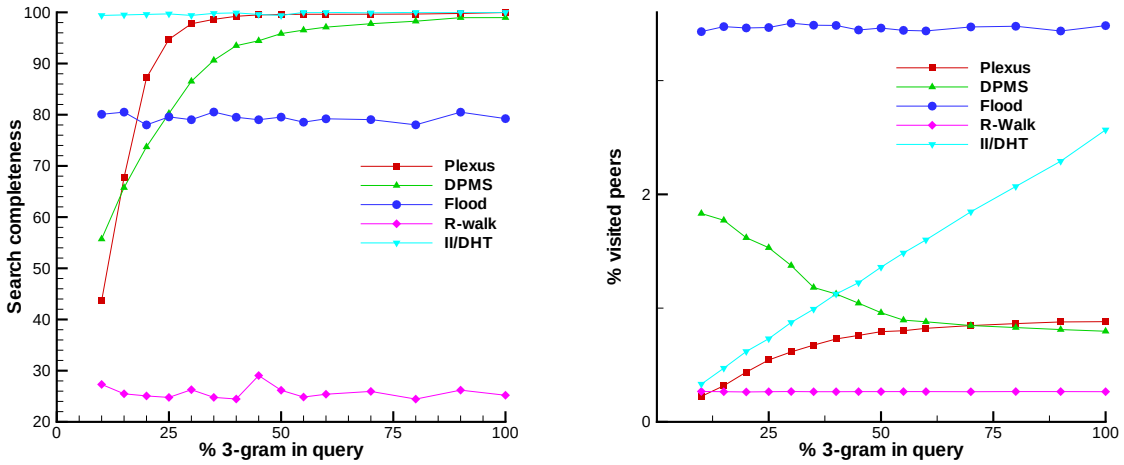
the expense of additional search traffic.

Search traffic is almost half in Plexus than that in II/DHT; yet search completeness in these two systems is almost identical. On the other hand, Plexus and DPMS generate similar search traffic, yet Plexus provides higher level of search completeness.

5.6.4 Search Flexibility

For this experiment we consider overlays of about 20,000 peers and vary the percentage of 3-grams (PNG) from the advertisements used for constructing the queries.

For Flooding and Random-walk, search traffic and completeness are invariant to the change in PNG. In the case of DPMS, we used an iterative search method and stopped after 20 matches were found. However at low PNG, the number of matching advertisements is more than 20, which results in low completeness levels. On the other hand, false-match probability is higher at low PNG levels, which explains the higher search



(a) Search completeness

(b) Search traffic

Figure 5.5: Search flexibility.

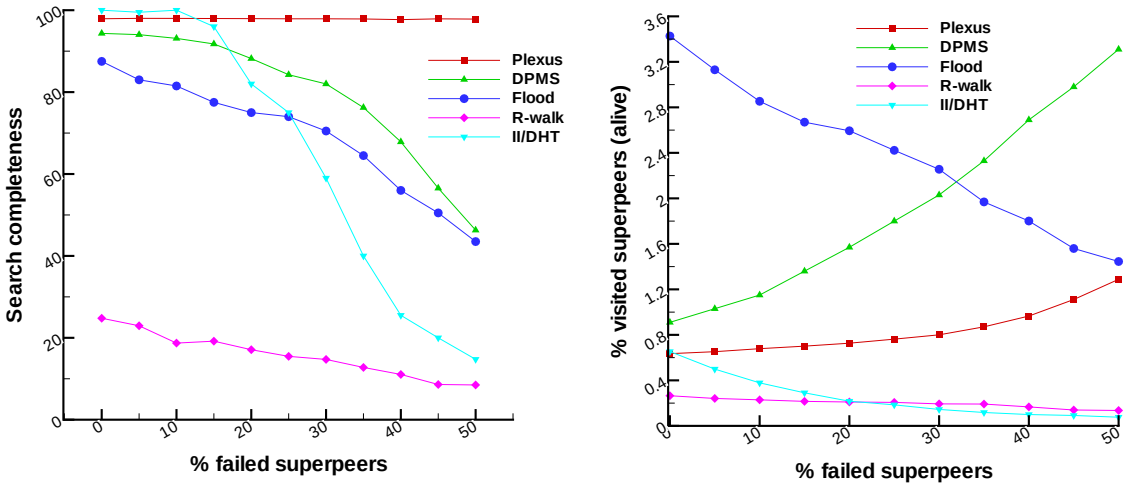
traffic in DPMS for low PNG levels.

In Plexus, many read quorum could not be satisfied at low PNG as there were too few bits per chunk. This results into lower search completeness and search traffic in Plexus for $PNG < 25\%$. On the contrary, II/DHT produces almost complete (99.8%) search results for all PNG levels, though search traffic increases linearly with PNG. In contrast, search traffic for Plexus is almost constant for $PNG > 40\%$.

5.6.5 Fault Resilience

For this experiment $PNG = 35\%$ is used for constructing queries from the available advertisements. We started with overlays of about 20,000 peers and gradually caused randomly chosen peers to fail in 5% steps.

In all techniques, except for Plexus, search completeness falls with increase in the percentage of failed peers (PFP) (Figure 5.6(a)). The fall is sharpest for II/DHT because



(a) Search completeness

(b) Search traffic

Figure 5.6: Fault resilience.

of its unidirectional routing table and lack of alternate routing paths. On the contrary, the use of multi-path routing and replication allows Plexus to achieve an almost constant level of search completeness regardless of the failure rate, though at the expense of higher search traffic.

For Plexus and DPMS, search traffic increases with PFP (Figure 5.6(b)), because these two techniques adopt alternate routing paths to reach the target peer(s) in presence of failures. On the other hand, search traffic decreases with the increase in PFP for II/DHT and Flooding because in these two cases the effective search tree gets pruned as more peers fail. This results into decreased search completeness. Finally for Random-walk, search traffic is independent of PFP as a walker is not aborted before TTL expires unless all the neighbors of an intermediate peer along the walker have failed.

5.7 Summary

In this section we have presented a comparative evaluation of Plexus and DPMS, against each other and representative search mechanisms from structured and unstructured paradigms. Based on the experimental results presented in this section, we can infer the followings:

- Unstructured techniques have low topology maintenance overhead but advertisement and search efficiencies are not good.
- Average degree in Plexus is close to II/DHT, yet the join overhead in Plexus is much lower and close to the unstructured cases. DPMS has low average degree and join overhead is moderate in DPMS.
- II/DHT exhibits the highest level of search completeness and flexibility. Plexus, on the other hand, exhibits almost identical level of completeness and flexibility as II/DHT when more than 30% of advertised information is present in the query. However, advertisement and query traffic in Plexus and DPMS is about 50% of that in II/DHT.
- Plexus proves to be the most resilient solution to peer failures. Unlike other techniques, it can maintain a high level of search completeness even in the presence of a large number of simultaneous failures. DPMS proves to be the second candidate in this regard.

Besides evaluating the performance of the studied systems, this section presented a brief survey of existing P2P simulators and discussed the rational behind using PeerSim for our experiments.

Chapter 6

Conclusion and Future Research

6.1 Conclusion

This thesis has focused on the search problem in Large Scale Distribute Systems, including P2P content-sharing, service discovery and P2P databases. A search mechanism in such environments needs to be flexible in query expressiveness, efficient on bandwidth and storage usage, and resilient to failures.

Instead of addressing the search problem in the above three domains independently, we have defined a common ground: the Distributed Pattern Matching (DPM) problem. Search problem in these three representative application domains can be mapped to the DPM framework. Therefore we argue that if the DPM problem can be solved efficiently then the search problem in these three important application domains can be solved efficiently.

As a solution to the DPM problem we presented two distributed search mechanisms: DPMS (Distributed Pattern Matching System) and Plexus. Both of these techniques solve the DPM problem, *i.e.*, allow a user to discover an advertised bit-vector using some subset of its 1-bits. In spite of solving the same problem, DPMS and Plexus utilized completely different indexing and routing mechanisms - each having its own strengths

and weaknesses.

For DPMS we adopted a *semi-structured* routing mechanism. The core idea behind DPMS architecture was the following observation: an indexing hierarchy of height $O\left(\log \frac{N}{\log N}\right)$ has $O(logN)$ elements in the topmost level. In DPMS we organized the indexing peers in a lattice-like hierarchy and used restricted flooding within $O(logN)$ peers at the topmost level. Here, N is the total number of peers in the system. To avoid hot spots and to ensure load balancing, we used recursive replication of indexing peers along the indexing hierarchy. In addition, we utilized a don't care based aggregation scheme to reduce the volume of indexed information at each peer. Query routing efficiency of DPMS was found to be comparable to that of structured P2P systems. For moderately stable networks, we were able to discover all the matching advertisements (up to a target maximum) regardless of the replication level.

For Plexus we have adopted a *structured* routing mechanism based on Coding Theory construct. To our knowledge, the use of Coding Theoretic constructs in P2P routing is novel. In contrast to the numeric distance based routing adopted in traditional DHT-approaches, we used Hamming distance based routing in Plexus. This property makes subset matching capability intrinsic to the underlying routing mechanism. The number of routing hops and the number of links maintained by each indexing peer were found to scale logarithmically with the number of peers in the network. As demonstrated by the simulation results on a network of about 20,000 superpeers, Plexus needed to visit only $0.7\% \sim 1\%$ of the superpeers to resolve a query and discovered about $97\% \sim 99\%$ of the advertised patterns matching the query. For achieving this level of completeness, the query needed to contain only 33% of the trigrams from an advertisement that it should match against. Plexus was able to deliver a high level of fault-resilience by utilizing replication and redundant routing paths. Even with 50% failed superpeers, Plexus attained a high level of search completeness (about $97\% \sim 99\%$) by visiting only $1.4\% \sim 2\%$ of

the superpeers.

We have also presented a proof-of-concept, using prototypes of the DPMS and Plexus protocols, and compared them against representative search techniques from unstructured and structured domains. Experimental results, presented in Chapter 5, reflects the effectiveness of DPMS and Plexus in solving the DPM problem without compromising advertisement or search efficiency. The comparative merits and demerits of DPMS and Plexus can be summarized as follows.

- DPMS requires $O\left(\log \frac{N}{\log N}\right)$ additional hops for finding each match, after a group of $O(\log N)$ peers at the topmost level has been flooded. On the contrary, Plexus can discover all the matches by searching a limited number of superpeers, scaling logarithmically with network size.

- Plexus requires that the percentage of 1-bits in a pattern to be within a certain limit as determined by the *query stretch*. For DPMS there exists no such bound. Yet this bound on Plexus seems to be practical for most cases. An advertisement with too many 1-bits will match almost all queries, while a query with too few 1-bits will match a large number of advertisements.

- Use of alternate routing paths is common to both DPMS and Plexus. However the number of alternate routing paths is much higher in Plexus than DPMS, which makes Plexus more resilient to failures than DPMS.

- Though DPMS exhibits lower level of replication factor than Plexus, the index update cost is higher in DPMS because the aggregation algorithm does not allow incremental updates.

- Peers in DPMS maintain a constant number of links, in contrast to $O(\log N)$ links per peer required by Plexus.

In summary, DPMS is appropriate for systems with relatively slow content update rate and requiring a predefined fixed number of search results. On the other hand, Plexus is suitable for the scenarios where search results should be complete, and peer and content dynamism are high.

Another contribution of this thesis is the survey and the taxonomy presented in Chapter 2. We have identified the main components involved in existing distributed search mechanisms from the three important application domains and have classified these search techniques based on the characteristics of these components. The taxonomy presented in this work is simple and generic, and encompasses the majority of search techniques in large scale distributed systems.

Taken collectively, our contributions increase our understanding of the search issues in large scale distributed systems and provide a common ground for future research on distributed search in three important application domains: P2P content-sharing, service discovery and P2P databases.

6.2 Future Research

There are several possible avenues for future research. In the following, we will discuss some of the most important ones:

Similarity Search

Although we have focused on subset matching in this work, DPMS and Plexus can easily be tailored to support similarity (or edit distance) matching. Possible applications of similarity search include phonetic search and semantic-laden search (as in Web Service discovery). In order to enable similarity search using DPMS we need to change the function that compares a query to an advertisement during the search process as follows: instead of forwarding queries based on strict subset match, we need to forward queries

based on some predefined threshold on hamming distance. While in the case of Plexus, we need to adjust the $\mathscr{A}(\cdot)$ and $\mathcal{Q}(\cdot)$ computation algorithms so that additional codewords are included to cover the supersets of advertised and queried patterns.

AI clustering

Clustering of pattern space has been extensively studied in Artificial Intelligence (AI) and Coding Theory literature. AI-based clustering techniques [79] require a priori knowledge of the pattern space (e.g., pattern density distribution) and training phases. Coding theory constructs, on the other hand, assume that all the patterns are equally likely, *i.e.*, uniformly distributed over the pattern space. A possible extension of Plexus is to investigate AI-based clustering techniques instead of using Coding Theoretic construct. Indeed, if the pattern density distribution is non-uniform over the pattern space, AI-based clustering techniques may yield a better routing performance.

Coding Theoretic Extensions

A major reason behind selecting the Extended Golay code for our implementation of Plexus is the code size. The proposed algorithm for finding $\mathcal{Q}(\mathrm{Q})$ requires to find all codewords within a specified distance from a given 24-bit vector. We had to use linear search for this step, as no efficient algorithm for this task is known. Use of linear search is not feasible with larger codes with hundreds of thousands of codewords. Similarly, finding $\mathscr{A}(\mathrm{P})$ for some advertised bit-vector P requires to compute $\mathcal{Q}(\cdot)$ for each subset of P. It may be possible to avoid groups of subsets by using the intersection property of a codeword and a subset, as explained in Equation (4.4). This is clearly an interesting issue that we will further investigate.

Alternate For Voting Algorithm

Use of the Voting algorithm in Plexus is suitable for systems with a small number of subnets. With a larger number of subnets in the system, the replication overhead will be high. Through careful observation, it can be realized that selecting appropriate subnets for advertisements and queries is essentially the same problem as that of finding $\mathscr{A}(\mathrm{P})$ and $\mathcal{Q}(\mathrm{Q})$ satisfying (4.2). Hence, with larger number of subnets we can reduce the number of selected subnets by using some linear code with large distance, *e.g.*, 1^{st} order Reed-Muller codes. We intend to investigate this possibility as a future extension of this work.

Self-tuning in DPMS

The experimental results in Section 3.13.1 suggests that system specific tuning is required for the following three parameters: pattern width W, minimum number of non-X bits O and aggregation ratio A. The characteristic function presented in Figure 3.7 provides a guideline for selecting the ratio of $\frac{O}{W}$ for a given value of A. Routing and indexing efficiency in DPMS can be tuned by dynamically adjusting the values of O and A. Further investigation is required for finding self-tuning capabilities in DPMS. Another possible optimization in DPMS is to introduce a collaboration mechanism among the leaf peers based on common interest. Patterns advertised by peers in same interest group are expected to have smaller hamming distance. This will increase aggregation rate without degrading the quality of aggregates. Investigating these self-tuning and self-optimization aspects of DPMS seems to be an interesting avenue for future research.

Applications of DPM

In this thesis we have presented two generic search techniques: DPMS and Plexus. In the experiments we have investigated the applicability of these two search techniques to

partial keyword search in content-sharing P2P networks. As discussed previously, these techniques can be applied to service discovery and P2P database domains. Detailed investigation and experimentation are however needed. As a future extension of this work we intend to apply these search techniques in other application domains than content-sharing P2P networks and to perform detailed performance studies.

The originality of this work lies in the formulation of the DPM construct and development of the DPMS and Plexus routing mechanisms for solving the subset matching problem in distributed environments. We believe that these concepts will aid in solving a number of other problems pertaining to P2P networking research, as well as, distributed databases, ontology-based semantic search and distributed information retrieval.

Bibliography

[1] Alta Vista website, http://www.altavista.digital.com/cgi-bin/query?pg=h. 44

[2] The BISON Project homepage, http://www.cs.unibo.it/bison/. 141

[3] The FastTrack Peer-to-Peer technology, http://www.fasttrack.nu/. 35

[4] The Gnutella website, http://www.gnutella.com. 8, 20, 35, 50, 51, 54, 56, 140

[5] The KaZaA website, http://www.kazaa.com/. 20, 54, 56

[6] Leo's lyrics database: http://www.leoslyrics.com/. 144

[7] The Morpheus website, http://morpheus.com/. 54

[8] The Napster website, http://www.napster.com/. 19, 54, 56

[9] The network simulator ns2. http://www.isi.edu/nsnam/ns/. 137

[10] Peersim Peer-to-Peer simulator, http://peersim.sourceforge.net/. 83

[11] Wget's website: http://www.leoslyrics.com/. 144

[12] Yahoo website, http://www.yahoo.com/docs/info/faq.html. 44

[13] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Symposium on Operating Systems Principles*, pages 186–201, 1999. 22

[14] R. Ahmed and R. Boutaba. Distributed pattern matching for P2P systems. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2006. 9

[15] R. Ahmed and R. Boutaba. A scalable Peer-to-Peer protocol enabling efficient and flexible search. Technical Report CS-2006-05, David R. Cheriton School of Computer Science, University of Waterloo, May 2006. 12, 101

[16] R. Ahmed and R. Boutaba. Distributed pattern matching: A key to flexible and efficient P2P search. *IEEE Journal on Selected Areas in Communications (JSAC)*, 25(1):73–83, January 2007. 10, 50, 148

[17] R. Ahmed and R. Boutaba. Plexus: A scalable Peer-to-Peer protocol enabling efficient subset search. Manuscript submitted to IEEE/ACM Transaction on Networking (TON), February 2007. 12, 101

[18] R. Ahmed, R. Boutaba, F. Cuervo, Y. Iraqi, T. Li, N. Limam, J. Xiao, and J. Ziembicki. Service discovery protocols: A comparative study. In *Proceeding of the IFIP/IEEE International Symposium on Integrated Network Management (IM) Application Sessions*, Nice, France, May 2005. 22, 37

[19] R. Ahmed, R. Boutaba, F. Cuervo, Y. Iraqi, T. Li, N. Limam, J. Xiao, and J. Ziembicki. Service naming in large-scale and multi-domain networks. *IEEE Communications Surveys & Tutorials*, 7(3):38–54, Third Quarter 2005. 37

[20] J. Allard, V. Chinta, S. Gundala, and G. G. Richard-III. Jini meets UPnP: An architecture for Jini/UPnP interoperability. In *Proceedings of Symposium on Applications and the Internet (SAINT)*, page 268, Washington, DC, USA, 2003. IEEE Computer Society. 37

[21] A. Amir, E. Porat, and M. Lewenstein. Approximate subset matching with don't cares. In *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 305–306, 2001. 2, 6

[22] S. Androutsellis-Theotokis and D. Spinellis. A survey of Peer-to-Peer content distribution technologies. *ACM Computing Surveys*, 45(2):195–205, December 2004. 18, 19, 20, 53, 54, 56, 101, 119

[23] G. Antoniou and F. van Harmelen. Web Ontology Language: OWL. *Handbook on Ontologies in Information Systems*, pages 76–92, 2003. 40

[24] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, 2003. 34, 56

[25] O. Babaoglu, G. Canright, A. Deutsch, G. A. D. Caro, F. Ducatelle, L. M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems*, 1(1):26–66, September 2006. 141

[26] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 71–80, 1993. 2

[27] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A scalable Peer-to-Peer architecture for intentional resource discovery. In *Proceedings of International Conference on Pervasive Computing*, pages 195–210. Springer-Verlag, 2002. 23, 30, 37, 38, 39, 41, 49, 50, 53, 125, 148, 154

[28] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003. 44

[29] R. Bhagwan, D. Moore, S. Savage, and G. Voelker. Replication strategies for highly available Peer-to-Peer storage. In *Proceedings of Future directions in Distributed Computing (FuDiCo)*, June 2002. 147

[30] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970. 3, 16, 61, 101

[31] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in P2P networks. In *Proceedings of the ACM international workshop on Web information and data management (WIDM)*, pages 48–55, New York, NY, USA, 2004. ACM Press. 40, 42, 53, 148

[32] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Service Architecture, 2004. [Online]. Available. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. 38, 39

[33] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003. 28, 62

[34] D. Brookshier, D. Govoni, and N. Krishnan. *JXTA: Java P2P Programming*. SAMS, 2002. 20, 42, 56

[35] J. Buford, A. Brown, and M. Kolberg. Meta service discovery. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2006. 37

[36] M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. DAML-S: Web Service description for the semantic web. In *Proceedings of International Semantic Web Conference on The Semantic Web (ISWC)*, pages 348–363, London, UK, 2002. Springer-Verlag. 39, 40

[37] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured Peer-to-Peer network. In *International World Wide Web Conference (WWW)*, 2004. 40, 42, 49, 50, 53

[38] D. Chamberlin, J. Siméon, S. Boag, D. Florescu, M. F. Fernández, and J. Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/. 49

[39] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proceedings of ACM SIGCOMM*, pages 407–418, 2003. 36, 52, 54, 56

[40] D. Choon-Hoong, S. Nutanong, and R. Buyya. *Peer-to-Peer Computing: Evolution of a Disruptive Technology*, chapter 2–Peer-to-Peer Networks for Content Sharing, pages 28–65. Idea Group Inc., 2005. 18

[41] V. Chvátal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979. 111

[42] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science (LNCS)*, 2009:46–66, 2001. 20, 54, 56, 140

[43] B. Cohen. Bittorrent protocol specification. http://www.bitconjurer.org/BitTorrent/protocol.html, February 2005. 139

[44] E. Cohen, A. Fiat, and H. Kaplan. Associative search in Peer-to-Peer networks: Harnessing latent semantics. In *Proceedings of IEEE INFOCOM*, 2003. 9, 36, 50, 51, 56

[45] E. Cohen and S. Shenker. Replication strategies in unstructured Peer-to-Peer networks. In *Proceedings of ACM SIGCOMM*, pages 177–190, 2002. 60, 147

[46] R. Cole and R. Harihan. Tree pattern matching and subset matching in randomized $o(n \log^3 m)$ time. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 66–75, 1997. 2, 6

[47] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *Proceedings of Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 245–254, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. 6

[48] C. Comito, S. Patarin, and D. Talia. A semantic overlay network for P2P schema-based data integration. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 88–94, Los Alamitos, CA, USA, 2006. IEEE Computer Society. 141

[49] J. Conway and N. Sloane. Orbit and coset analysis of the Golay and related codes. *IEEE Transactions on Information Theory*, 36(5):1038–1050, September 1990. 104

[50] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of International Conference on Mobile Computing and Networking (MOBICOM)*, pages 24–35, 1999. 22, 37, 50, 52, 53, 56, 64

[51] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured Peer-to-Peer overlays. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 58, Berkeley, CA, February 2003. 138

[52] S. Decker, M. Schlosser, M. Sintek, and W. Nejdl. Hypercup - hypercubes, ontologies and efficient search on P2P networks. In *International Workshop on Agents and Peer-to-Peer Computing*, July 2002. 39

[53] A. Fabrikant, E. Koutsoupias, and C. H. Papadimitriou. Heuristically optimized trade-offs: A new paradigm for power laws in the Internet. In *Proceedings of*

International Colloquium on Automata, Languages and Programming (ICLAP), pages 110–122, London, UK, 2002. Springer-Verlag. 147

[54] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of ACM SIGCOMM*, pages 251–262, New York, NY, USA, 1999. ACM Press. 147

[55] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. The coDB robust Peer-to-Peer database system. In *Proceedings of Workshop on Semantics in Peer-to-Peer and Grid Computing at the International World Wide Web Conference (WWW)*, May 2004. 42

[56] J. Freudenberger and V. Zyablov. On the complexity of suboptimal decoding for list and decision feedback schemes. *Discrete Applied Mathematics*, 154(2):294–304, 2006. 111

[57] M. Fuchs, P. Wadler, J. Robie, and A. Brown. XML schema: Formal description. W3C working draft, W3C, Sept. 2001. http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/. 26

[58] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *Proceedings of the VLDB Conference,*, 2003. 40, 41, 42, 50, 53

[59] P. Ganesan, Q. Sun, and H. Garcia-Molina. Adlib: A self-tuning index for dynamic Peer-to-Peer systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 256–257, Los Alamitos, CA, USA, 2005. IEEE Computer Society. 31

[60] P. Garcia, C. Pairot, R. Mondejar, J. Pujol, H. Tejedor, and R. Rallo. PlanetSim: A new overlay network simulation framework. *Lecture Notes in Computer Science (LNCS) Software Engineering and Middleware*, 3437:123–136, 2005. 138

[61] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis. Web service discovery mechanisms: Looking for a needle in a haystack? In *International Workshop on Web Engineering*, 2004. 38

[62] L. Gasieniec, J. Jansson, and A. Lingas. Approximation algorithms for hamming clustering problems. *Journal of Discrete Algorithms*, 2(2):289–301, 2004. 68

[63] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh Symposium on Operating Systems Principles*, pages 150–162, 1979. 121

[64] T. M. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. P2Psim version 0.3 (MIT IRIS project). http://pdos.csail.mit.edu/p2psim/, April 2005. 139

[65] M. J. E. Golay. Notes on digital coding. In *Proceedings of the IEEE*, volume 37, 1949. 13, 101

[66] A. Grizard, L. Vercouter, T. Stratulat, and G. Muller. A Peer-to-Peer normative system to achieve social order. In *AAMAS06 Workshop on Coordination, Organization, Institutions and Norms in agent systems (COIN)*, Hakodate, Japan, May 2006. 141

[67] D. Guo, H. Chen, C. Xie, H. Lei, T. Chen, and X. Luo. Decentralized grid resource locating protocol based on grid resource space model. In *Lecture Notes in Computer Science (LNCS)*, volume 3795, pages 621–632,. Springer-Verlag, 2005. 141

[68] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and back-

ground overhead. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 160–169, 2003. 139

[69] V. Guruswami and M. Sudan. Extensions to the johnson bound. Manuscript, 2001. 126

[70] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol (SLP), version 2. Technical report, IETF, RFC2608, http://www.ietf.org/rfc/rfc2608.txt, June 1999. 22, 37, 53

[71] D. Hales and S. Arteconi. SCALER: A self-organizing protocol for coordination in Peer-to-Peer networks. *IEEE Intelligent Systems*, 21(2):29–35, 2006. 141

[72] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based Peer-to-Peer networks. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 242–259, 2002. 31, 148

[73] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003. 34, 52, 55, 56

[74] Q. He, M. Ammar, G. Riley, H. Raj, and R. Fujimoto. Mapping peer behavior to packet-level details: a framework for packet-level simulation of Peer-to-Peer systems. In *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 71–78, October 2003. 138

[75] S. Herschel and R. Heese. Humboldt Discoverer: A semantic P2P index for PDMS. In *Proceedings of the International Workshop Data Integration and the Semantic Web (DISWeb'05)*, June 2005. 42, 53, 56

[76] D. G. Hoffman, Wal, D. A. Leonard, C. C. Lidner, K. T. Phelps, and C. A. Rodger. *Coding Theory: The Essentials*. Marcel Dekker, Inc., New York, NY, USA, 1991. 12, 101, 106

[77] T. Howes. The string representation of ldap search filters, 1997. 49

[78] H. Hu and A. Seneviratne. Autonomic Peer-to-Peer service directory. *IEICE/IEEE Joint Special Section on Autonomous Decentralized Systems*, E88-D(12):2630–2639, December 2005. 39

[79] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999. 106, 164

[80] M. Jelasity and O. Babaoglu. T-Man: Fast gossip-based construction of large-scale overlay topologies. Technical Report UBLCS-2004-7, University of Bologna, Department of Computer Science, Bologna, Italy, May 2004. http://www.cs.unibo.it/techreports/2004/2004-07.pdf. 141

[81] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, pages 102–109, Tokyo, Japan, March 2004. IEEE Computer Society. 141

[82] X. Jin, W.-P. K. Yiu, and S.-H. Chan. Supporting multiple-keyword search in a hybrid structured Peer-to-Peer network. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 42–47, Istanbul, June 2006. 31, 34, 52

[83] S. Joseph. Neurogrid: Semantically routing queries in Peer-to-Peer networks. In *Lecture Notes in Computer Science (LNCS): Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 202–214, London, UK, 2002. Springer-Verlag. 140

[84] S. Joseph. An extendible open source P2P simulator. *P2PJournal*, pages 1–15, November 2003. 140

[85] Y. Joung, L. Yang, and C. Fang. Keyword search in DHT-based Peer-to-Peer networks. *IEEE Journal on Selected Areas in Communications (JSAC)*, 25(1):46–61, January 2007. 31, 32, 52

[86] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 98–107, 2003. 139

[87] X. Kaiping, H. Peilin, and L. Jinsheng. FS-Chord: A new P2P model with fractional steps joining. In *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW)*, 2006. 153

[88] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for Peer-to-Peer networks. In *Conference on Information and Knowledge Management (CIKM)*, 2002. 51

[89] K. Kant and R. Iyer. Modeling and simulation of Adhoc/P2P resource sharing networks. In *Proceedings of TOOLS*, November 2003. 138

[90] M. Kay, M. F. Fernández, S. Boag, D. Chamberlin, A. Berglund, J. Siméon, and J. Robie. XML path language (XPath) 2.0. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/. 5, 41, 49

[91] J. Kim and G. Fox. A hybrid keyword search across Peer-to-Peer federated databases. In *Proceedings of East-European Conference on Advances in Databases and Information Systems (ADBIS)*, September 2004. 42

[92] G. Koloniari and E. Pitoura. Peer-to-Peer management of XML data: issues and research challenges. *ACM SIGMOD Record*, 34(2):6–17, 2005. 25

[93] T. Koponen and T. Virtanen. A service discovery: a service broker approach. In *Proceedings of Hawaii International Conference on System Sciences*, January 2004. 37

[94] S. R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 178–183, 1989. 2

[95] N. Kotilainen, M. Vapa, T. Keltanen, A. Auvinen, and J. Vuori. P2PRealm Peer-to-Peer Network Simulator. In *Proceedings of Intenational Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks*, pages 93–99, June 2006. 138

[96] A. Kumar, J. Xu, and E. W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, 2005. 36

[97] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. supersed work, W3C, Feb. 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222. 42

[98] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, May 2005. 139

[99] M. Li, W. Lee, and A. Sivasubramaniam. Neighborhood signatures for searching P2P networks. In *Proceedings of Seventh International Database Engineering and Applications Symposium (IDEAS)*, pages 149–159, 2003. 9, 36, 50, 52, 56, 64

[100] Y. Li, F. Zou, Z. Wu, and F. Ma. PWSD: A scalable web service discovery architecture based on Peer-to-Peer overlay network. In *Proceedings APWeb, Lecture Notes Computer Science (LNCS)*, volume 3007, 2004. 39, 53

[101] N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, D. T. Li, R. Boutaba, and F. Cuervo. OSDA: Open service discovery architecture for efficient cross-domain service provisioning. *International Computer Communications Journal (COMCOM) Special Issue on Emerging Middleware for Next Generation Networks*, 30(3):546–563, February 2007. 37

[102] L. Liu, K. D. Ryu, and K. Lee. Supporting efficient keyword-based file search in Peer-to-Peer file sharing systems. In *Proceedings of GLOBECOM*, 2004. 30, 31, 32, 52, 148

[103] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured Peer-to-Peer networks. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2002. 8, 36, 51

[104] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 127–140, March 2003. 138

[105] P. Maymounkov and D. Mazireres. Kademlia: A Peer-to-Peer information system based on the XOR metric. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65. Springer-Verlag, March 2002. 6, 30, 51, 55, 139

[106] B. A. Miller, T. Nixon, C. Tai, and M. D. Wood. Home networking with Universal Plug and Play. *IEEE Communications Magazine*, pages 104–109, December 2001. 22, 37

[107] J. Miller. Characterization of data on the Gnutella Peer-to-Peer network. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 489– 494, Las Vegas Nevada, USA, January 2004. 144

[108] M. Montebello and C. Abela. DAML enabled web service and agents in semantic web. In *Workshop on Web, Web Services and Database Systems, Lecture Notes Ccomputer Science (LNCS)*, 2003. 39

[109] A. Montresor. A robust protocol for building superpeer overlay topologies. In *Proceedings of International Conference on Peer-to-Peer Computing (P2P)*, Zurich, Switzerland, August 2004. 141

[110] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A survey of Peer-to-Peer network simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium (PGNET)*, Liverpool, UK, June 2006. 137

[111] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Loser. Super-peer-based routing strategies for RDF-based Peer-to-Peer networks. *Journal of Web Semantics*, 1(2):177–186, February 2004. 40

[112] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 633–644, 2003. 42, 53

[113] M. T. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. 25

[114] E. Prud'Hommeaux and A. Seaborne. SPARQL query language for RDF. Working Draft WD-rdf-sparql-query-20061004, World Wide Web Consortium (W3C), October 2006. 43, 49

[115] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. 34

[116] M. Rabin. Fingerprinting by random polynomials. Technical report, CRCT TR-15-81, Harvard University, 1981. 41

[117] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Professional, 2002. 25

[118] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, 2001. 6, 21, 30, 31, 40, 51, 55

[119] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of IEEE INFOCOM*, 2002. 36, 52

[120] P. Rompothong and T. Senivongse. A query federation of UDDI registries. In *Proceedings of International Symposium on Information and Communication Technologies (ISICT)*, pages 578–583, 2003. 39

[121] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale Peer-to-Peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001. 30, 51

[122] H. Sagan. *Space-filling curves*. Springer-Verlag, 1994. 33

[123] Salutation Consortium. Salutation architecture specification version 2.0c. http://www.salutation.org, June 1999. 37

[124] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of Peer-to-Peer file sharing systems. In *Multimedia Computing and Networking (MMCN)*, 2002. 98

[125] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. A scalable and ontology-based P2P infrastructure for semantic web services. In *Proceedings of International Conference on Peer-to-Peer Computing (P2P)*, September 2002. 39, 40, 53

[126] M. T. Schlosser, T. E. Condie, and S. D. Kamvar. Simulating a file-sharing P2P network. In *Workshop on Semantics in P2P and Grid Computing*, December 2002. 139

[127] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 8(3):19–26, June 2004. 30, 31, 33, 39, 49, 52, 148, 154

[128] C. Schmidt and M. Parashar. Peer-to-Peer approach to web service discovery. In *WWW: Internet and web information systems*, volume 7, pages 211–229, 2004. 39, 53

[129] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making Peer-to-Peer keyword searching feasible using multi-level partitioning. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 151–161. Springer, 2004. 35

[130] C. M. Sperberg-McQueen, T. Bray, E. Maler, J. Paoli, and F. Yergeau. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816. 25

[131] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. OReillys www.openp2p.com, February 2001. 144, 147

[132] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable Peer-to-Peer lookup protocol for Internet applications. *IEEE/ACM Transaction on Networking (TON)*, 11(1):17–32, 2003. 6, 20, 30, 31, 40, 51, 55

[133] D. Stutzbach and R. Rejaie. Characterizing unstructured overlay topologies in modern P2P file-sharing systems. In *Internet Measurement Conference*, October 2005. 120, 144

[134] D. Stutzbach, S. Zhao, and R. Rejaie. Characterizing files in the modern Gnutella network. *Multimedia Systems Journal (To appear)*, 2007. Technical Report CIS-TR-06-09, University of Oregon, http://mirage.cs.uoregon.edu/pub/stutzbach-2006-msj.pdf. 99, 144

[135] M. Sudan. List decoding: Algorithms and applications. *Lecture Notes in Computer Science (LNCS)on the Proceedings of the International Conference IFIP Theoretical Computer Science (TCS)*, 1872:25–41, August 2000. 107

[136] Sun Microsystems. Jini Technology Core Platform Specification, October 2000. [Online]. Available. http://www.sun.com/jini/specs/. 22, 37

[137] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient Peer-to-Peer information retrieval. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, June 2004. 31

[138] C. Tang, Z. Xu, and M. Mahalingam. pSearch: information retrieval in structured overlays. *ACM SIGCOMM Computer Communication Review*, 33(1):89–94, 2003. 30, 31, 32, 52

[139] N. S. Ting and R. Deters. 3LS - a Peer-to-Peer network simulator. In *Proceedings of International Conference on Peer-to-Peer Computing (P2P)*, pages 212–213, September 2003. 139

[140] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search for Peer-to-Peer networks. In *Proceedings of International Conference on Peer-to-Peer Computing (P2P)*, 2003. 36, 51, 52

[141] UDDI Consortium. UDDI Technical White Paper, 2002. [Online]. Available. http://www.uddi.org/ pubs/Iru_UDDI_Technical_White_Paper.pdf. 22, 38, 39

[142] M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. *Knowledge Sharing and Review*, 11(2), 1996. 39

[143] S. Voulgaris, M. Jelasity, and M. van Steen. A robust and scalable Peer-to-Peer gossiping protocol. In *Proceedings of International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, 2003. 73, 152

[144] L. Xiao, Z. Zhuang, and Y. Liu. Dynamic layer management in superpeer architectures. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1078–1091, November 2005. 119

[145] H. Yagi, T. Matsushima, and S. Hirasawa. A heuristic search method with the reduced list of test error patterns for maximum likelihood decoding. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E88-A(10):2721–2733, 2005. 111

[146] B. Yang and H. Garcia-Molina. Improving search in Peer-to-Peer networks. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 2002. 20, 36, 51

[147] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 49–60, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 119

[148] K.-H. Yang and J.-M. Ho. Proof: A DHT-based Peer-to-Peer search engine. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006)*, pages 702–708, December 2006. 32

[149] W. Yang and N. Abu-Ghazaleh. GPS: a general Peer-to-Peer simulator and its use for modeling BitTorrent. In *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 425–432, September 2005. 139

[150] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC)*, 22(1):41–53, January 2004. 6, 21, 30, 31, 139

[151] M. Zimand. A list-decodable code with local encoding and decoding. In *Proceedings of Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/SAWN)*, pages 232–237, Washington, DC, USA, 2005. IEEE Computer Society. 111