ADAPTIVE P2P PLATFORM FOR DATA SHARING

By
Ng Wee Siong

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
NATIONAL UNIVERSITY OF SINGAPORE
REPUBLIC OF SINGAPORE
MARCH 2004

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF
COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Adaptive P2P Platform for Data Sharing**" by **Ng Wee Siong** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Dated: March 2004

External Examiner: _____
Karl Aberer, Alon Halevy

Research Supervisor: _____
Ooi Beng Chin

Examing Committee: _____
Ang Chuan Heng

_____
Teo Yong-Meng

_____
Anthony K. H. Tung

Table of Contents

List of Tables

List of Figures

Summary

Peer-to-peer (P2P) systems are becoming increasingly popular as they enable users to exchange digital information by participating in complex networks. In a distributed P2P system, nodes of equivalent capabilities and responsibilities pool their resources together in order to share information and services. Such systems are inexpensive, easy to use, highly scalable and do not require central administration. However, many of the existing P2P systems are limited in several ways. First, they provide only file-level sharing (coarse granularity) and lack object/data management capabilities and support for content-based search. Second, there is no predetermined global schema shared among nodes. As a result, the query is largely based on keywords. Third, they are limited in extensibility and flexibility. Finally, a node's peers are typically statically defined.

In order to deal with the scale and dynamism that characterize P2P systems, a paradigm shift is required; that includes self-organization, adaptation and fine granularity query support as intrinsic properties. In particular, we focus on the effectiveness of a P2P sharing systems with respect to the concept of data management. First, we present a conceptual framework that facilitates finer granularity data access and sharing. Second, we investigate the impact of decision making without relying on global knowledge. Third, we study the effectiveness of various data placement policies on a network with dynamic participants. Finally, we attempt to provide a methodology for data acquisition on heterogeneous data sources environments. In this thesis, we have implemented and experimented with a variety of P2P strategies with the objective of solving the aforementioned tasks.

BestPeer is a generic P2P platform which facilitates fast and easy P2P application development. It supports finer granularity of data sharing where partial content of a file may be shared, and it also shares computational power. Moreover, BestPeer integrates two powerful technologies: mobile agents and P2P technologies. While P2P technology provides resource-sharing capabilities amongst nodes, mobile agents technology further extends the functionalities. Our solution incorporates a self-configurable approach, by which a node in the BestPeer network can dynamically reconfigure itself by keeping peers that benefit it most. We evaluated BestPeer on a cluster of 32 Pentium II PCs, each running a Java-based storage manager. Our experimental results show that BestPeer provides excellent performance compared to traditional non-configurable models. Further experimental study reveals its superiority over Gnutella's protocol.

For decision making without relying on global knowledge, we have proposed PeerDB, which is a full-fledged data management system that supports fine-grain content-based search. Our solution incorporates Information Retrieval (IR) techniques which enable peers to share data without a shared schema. PeerDB employs a name-based matching technique that matches schema elements by relying on the user to supply additional information (meta-data) in order to reduce mismatch. PeerDB primarily concerns itself with online information exploration. Online information exploration contrasts with traditional data translation and schema integration strategies in the way that the results of the former are transient and users are more tolerant to mismatched candidates. Schema integration, on the other hand, needs to be ensured of a certain degree of consistency and accuracy, which in turn, requires more complicated approaches.

PeerOLAP has been proposed as a new data placement strategy for P2P systems, in particular, for data warehousing applications. PeerOLAP acts as a large distributed cache for OLAP results by exploiting under-utilized peers. We have proposed and evaluated three cache control policies (Isolated, Hit Aware and Voluntary) that impose different levels of cooperation among the peers. Notably, our approach

facilitates fast and efficient query performance since data can be placed in strategic locations that are based on different cache control policies. PeerOLAP achieves significant performance gains with respect to traditional client-side cache systems. This is accomplished by (i) query optimization techniques that determine which chunks should be requested from the warehouse, and which should be retrieved from the peers; (ii) caching policies that enable cooperation among caches and eliminate unnecessary replication of objects; and (iii) re-configuration mechanisms that create virtual neighbors of peers with similar access patterns.

Content-based similarity queries have received considerable attention in the P2P community. In this work, we focus specially on similarity search in a broadcast-based P2P system since such queries are considerably fuzzy. We propose FuzzyPeer, which deals with the problem of data acquisition on heterogeneous data sources environments. In our system, the participation of peers is ad hoc and dynamic, their functionalities are symmetrical, and there is no centralized index. To avoid flooding the network with messages, we develop a technique that takes advantage of the fuzzy nature of the queries. Specifically, some queries are "frozen" inside the network, and are satisfied by the streaming results of similar queries that are already running. We describe several optimization techniques for single and multiple-attribute queries, and study their trade-offs. Our results suggest that by reusing the existing streams, the scalability of the system improves both in terms of the number of users and throughput.

In this research, we present some preliminary fundamental results, and describe our initial work in the construction of an adaptive P2P data sharing and management system. Our results indicate that with proper and innovative strategies, it is possible to achieve significant performance gains over traditional systems despite the dynamism of participants and heterogeneity of data sources. To this end, we believe that our contributions have successfully addressed some of the issues concerning the performance, flexibility and scalability improvement of P2P-like distributed data sharing systems that support dynamic data and dynamic workloads.

Acknowledgements

I would like to thank Professor Ooi Beng Chin, my supervisor, for his many suggestions and constant support during this research. His constant motivation, exemplary assiduousness and deep insight have enabled me to develop as a researcher. I would like to take this opportunity to thank Associate Professor Tan Kian Lee, whose detailed comments and suggestions concerning my work have not only contributed significantly to the enrichment of this thesis, but also shaped my research capabilities to a considerable extent. I am also thankful to Dr. Stephane Bressan for his guidance through the early years of chaos and confusion.

I sincerely wish to thank Associate Professor Dimitris Papadias for giving me the wonderful opportunity to work with him during my one-month research attachment at the Hong Kong University of Science and Technology. I also wish to express my appreciation to Dr. Panagiotis Kalnis for the useful discussion that I had with him and also for making my time in HKUST meaningful.

I have had the pleasure of meeting Professor Zhou Aoying and many students who are working in the database research lab at Fudan University, China. They are wonderful people, and their support makes research like this possible.

I would like to thank copy-editor Alexia Leong for editing the thesis. Of course, I am grateful to my parents for their patience and *love*. Without them, this work would never have come into existence. I wish to especially thank my wife Liau Yen Peng for encouraging me to do something I had only talked about for years, and for helping me with this opportunity to pursue it to completion.

Finally, I wish to thank the following: Mr Cui Bin, Mr Rajiv Panicker, Mr Liau Chu Yee and all members of the Database and Electronic Laboratories for their friendship and willingness to help me in various way.

I sincerely thank the National University of Singapore for providing me with a scholarship to support the early years of my doctoral studies, and for awarding me

the Graduate Dean's Award. Last, but not the least, I have been supported financially by the NSTB/MOE research grant RP960668. For this assistance, I am very grateful.

Chapter 1

Introduction

Peer-to-peer (P2P) technology, also called peer computing, is an emerging paradigm that is now viewed as a potential technology that could re-architect distributed architectures (e.g., the Internet). In a P2P distributed system, a large number of nodes (e.g., personal computers connected to the Internet) can potentially be pooled together to share their resources, information and services. These nodes, which can both consume as well as provide data and/or services, may join and leave the P2P network at any time, resulting in a truly dynamic and ad hoc environment. The distributed nature of such a design provides exciting opportunities for new killer applications to be developed.

The P2P model can be best deciphered in terms of the client-server computing model (Figure 1.1). The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. In the client-server model, there is a centralized server that is dedicated to managing data storage, sharable printers, applications software, databases and different varieties of computing resources; the client is defined as a requester of services from the server and is normally a less powerful personal computer. The core concept behind P2P computing is that each edge system can function

both as a client and a server. This suggests that the role and relationship of these edge systems can be best described in terms of "peer-to-peer".

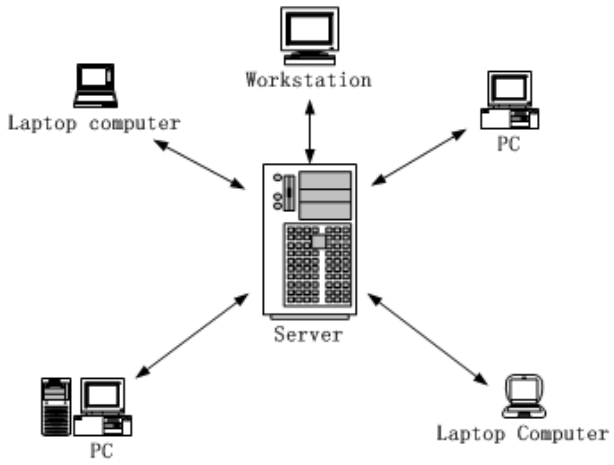


Figure 1.1: Client-Server Computing Model

Although the concept of P2P is not new, the pervasiveness of the Internet and the publicity gained as a result of music-sharing have caused researchers and application developers to realize the untapped resources, both in terms of computer technology and information. Edge devices such as personal computers are connected to each other directly, forming special interest groups and collaborating to become a large search engine of the information maintained locally, and in virtual clusters and file systems. Indeed, over the last few years, we have seen many systems being developed and deployed; e.g., Freenet [39], Gnutella [42], Napster [75], ICQ [52], SETI@home [95] and LOCKSS [67].

The initial thrusts of the use of P2P platform were mainly social. Applications such as ICQ [52] and Napster [75] enable their users to create online communities that are self-organizing, dynamic and yet collaborative. The empowerment of users, freedom of choice and ease of migration, form the main driving force for the initial

wide acceptance of P2P computing [83]. When deployed in a business organization, the accesses and dynamism of P2P can be constrained as data and resource sharing may be compartmentalized and restricted according to the roles that users play.

Consequently, various forms of P2P architectures have emerged and will evolve and mutate over time to find a natural fit for different application domains. One such success story is the deployment of the paradigm of edge-services in content search, where it has been exploited in pushing data closer to users for faster delivery and solving network and server bottleneck problems.

In summary, the P2P architecture is more cost-effective, compared to the traditional centralized client/server architecture. In the traditional centralized client/server architecture, servers typically bear the predominant cost of the system, e.g., maintenance and administration overheads. The cost increases gradually, in a manner proportional to the number of clients it serves. More resources such as processing power and disk space are needed to handle increasing workloads. When the main cost becomes too large, a P2P architecture can help spread the cost over all the peers. Each node in the P2P system brings with it certain resources such as computing power or storage space. Applications that benefit from huge amounts of these resources, such as computation-intensive simulations or distributed file systems, naturally lean towards a P2P structure to aggregate these resources to solve the larger problem. In addition to cost-effectiveness, P2P systems can scale to a large extent by adding more peers into the community. The scalability provided by P2P architectures is important because it implies that the system can be built gradually depending on the workload and with minimum administration cost. Furthermore, autonomy is an essential hallmark of P2P systems which allow users to store their own data locally

instead of relying on dedicated centralized servers.

1.1 P2P Applications

Broadly, P2P applications can be classified into two categories: resource sharing and data sharing. In resource sharing, applications allow enterprises or individuals to leverage on available (idle or otherwise) CPU cycles, disk storage and bandwidth capacity within a network. P2P computing enables the harnessing of underused resources to perform tasks that would otherwise require a much more expensive machine such as a super computer. Similarly, data storage devices could be exploited to create a wide area storage network, and to push the data closer to the users. SETI@Home[95] which is computation and storage intensive is one of the most well known examples.

In data sharing, applications allow users to access, modify and exchange information in a flexible manner. Notable application domains are instant messaging, groupware and file sharing. Instant messaging applications provide services such as test messaging, email, voice-over-IP and mobile phone short messaging services. Such facilities provide the convenience of the immediacy of phone calls, while providing opportunities for new and sophisticated applications that require real-time streaming and response. Groupware are applications that enable inter-organization communication and collaborations, providing functionalities such as information sharing, scheduling, calendaring and workflow. File sharing has so far attracted the most attention, and has resulted in many systems that allow the copying of files and search of the contents of files.

Efficient and effective resource location mechanisms are necessary to facilitate speedy search in a vast volume of data sources. It is a major concern in the design

of P2P data sharing systems, such as P2P file sharing systems, which share different varieties of data e.g., text documents, executable files, audio, image and video. There are many mechanisms for locating resources in P2P systems. A naive approach is to index these objects according to their file name and store the information in a specialized index node [75]. Alternatively, resource locating can be based on the propagation of messages from peer to peer until a match is found [42, 39]. More recently, concepts from the "*small-world*" [60] phenomenon are employed to facilitate finding information with a distributed index in P2P systems. A useful approach based on the distributed hashing table (DHT) has become increasingly common. Each object consists of a hashed identifier, which corresponds to a set of coordinates in a structured hashed space [92, 31, 100]. Another representation of the distributed index is the routing indexes [25], in which case, retrieval is achieved by means of forwarding queries to neighboring peers that are more likely to have the answer. The clear difference between routing indexes and DHT-based systems is that the former does not require a specific structured network. Unfortunately, it has been shown recently that existing resource location mechanisms do not support complex queries and provide only coarse granularity of sharing [50].

Complex queries facilities are essentially vital components of many data management applications such as bioinformatics applications. In bioinformatics applications, the ability to retrieve similar sequence patterns would be useful to researchers in sequence analysis, structural prediction and reasoning in genomic data. As an example, for a nucleotide sequence ACCTGATT, one can build an index over n-grams for the various values of n (e.g., AC, CT, GA, TT) so as to provide for the retrieval of similar patterns.

From the above discussion, it is clear that P2P data sharing systems must have the following intrinsic properties: the ability to support fine-granularity queries, extensibility and flexibility to support complex queries, and no need for any specific network structure.

1.2 Motivation

Various types of resource management schemes have been designed with the objective of resolving the problem of data sharing in P2P environments. In P2P environments, mostly the schema is not given in advance or it might be implicit in the data. Consequently, it is especially challenging to impose an efficient query processing technique across heterogeneous data sources as that usually triggers off data integration problems. One approach is to enforce uniform global semantics among peers as in Napster-like systems. It has been observed that such a scheme allows for easier implementation and management of resources. However, such a scheme is conceivably inflexible for most applications, owing to the autonomous nature of each peer. Further, a scheme updates operation, e.g., adding a new data type, which might have a global effect that causes a reorganization of existing data objects. Instead of creating a global scheme to represent the heterogeneity of data sources, one may define limited global semantic schemas to be enforced on all participants. As a result, the fruitful of traditional data integration approaches can potentially be reused [89, 45, 22, 103]. This approach has shown its usefulness in systems such as in [44, 48, 90, 84]. For example, the PIAZZA system [44, 48, 47, 46] creates a schema mapping mechanism to capture the structural and terminologies between a given source schema and a new target schema. Consider that given a new target schema, a GAV (*global-as-view*)

definition that relates to the source schema is used to identify matching parts of the source and target schemas. In contrast to the GAV formalism, PIAZZA allows users to specify the mapping of data sources to the missing attributes in the target schema, which is essentially a property of the LAV (*local-as-view*) formalism.

In contrast to conventional distributed data management systems, the schema in P2P systems is relatively large and updates frequently. This poses a basic challenge for a query optimizer in distributed computing, in that there is a need to provide a minimum cost query plan based on limited knowledge of its environment. In addition, other criteria such as the current workload status of peers, network bandwidth, data objects shared by peers and location may not be constant from time to time. Therefore, much literature has sought to derive a good decision with the constraint of a small scope of global knowledge, since gathering complete knowledge of all available resources of the environment requires a significant amount of collaboration among peers and is not a practical viable option. The decision making for query processing may be made in one of two ways: (1) By building a centralized catalogue of the global knowledge collection of all available information. The decision here is made in the centralized peer or among a few peers [111, 75, 74]. Incidentally, this approach reduces the intensity of the collaboration among peers. However, this model introduces a single point of failure and a potential bottleneck from the standpoint of scalability. (2) By having every peer making autonomous decisions with limited knowledge of each other – which is a better solution in terms of scalability and feasibility for P2P environments [59, 48, 78, 10]. Autonomous query decision making with limited global knowledge is however understandably challenging. Take for example a

broadcast-based system (e.g., Gnutella [42]), which uses message flooding to propagate queries. A peer knows only its neighbors as part of its global knowledge. Every neighbor peer is contacted and forwards the message to its own neighbors until the message lifetime expires. Even though this is an extreme simple case of autonomous query processing, there remains the issue of determining an optimal message lifetime for applications. The decision on message lifetime is very important since it significantly affects performance; a long message lifetime may be counter-intuitive in some environments (to minimize network traffic), while in others, they can be a prerequisite (to explore more results).

Like semi-structured data sources, the data shared in P2P environments is not strongly typed. It may be possible that different objects with the same attribute may be of different types or vice versa. Notwithstanding this, there are varieties of objects stored in a computer and each may require different access granularities. Some objects only provide atomic granularity level access in which they are indivisible, e.g., an executable file. Others, such as text files and database objects, can be accessed at different granularity levels, e.g., a relation entity in a relational database that can be accessed in terms of rows, columns or tuples depending on the query requirements. Clearly, implementing a P2P system that is able to support all kinds of granularity level access without enforcing strongly typed relationships among objects is truly a challenging task.

The network formed with the P2P architecture is dynamic as participant nodes are allowed to join and leave the system at will. This characteristic is particularly unique to P2P environments as compared to the traditional distributed computing systems which treat an inaccessible node as an exception. Hence, the primary task of

data placing in P2P systems is to impose a mechanism to guarantee reliable behavior in a dynamic and ad hoc environment. However, satisfying both these constraints (i.e., reliability and dynamism) simultaneously may not always be possible in the case of P2P systems, and hence a trade-off is usually called for. There are several intuitive solutions. All the data can be placed only on reliable peers, which can greatly increase the reliability of the system (e.g., superpeer architecture [111]). Yet this approach will reduce flexibility and create bottlenecks that impede system performance. Alternatively, based on the selectivity approach, one can try to categorize peers into reliable and dynamic peers. All original content can then be stored in the reliable peers and replicated at the dynamic peers. Unfortunately, this complicates the peer selection problem (i.e., selection of reliable and dynamic peers). Meanwhile, maintaining consistency over replicated objects becomes a necessity in such cases.

In summary, many P2P data sharing systems have been proposed and deployed [39, 42, 75, 52, 95, 67, 7], but most have their own inherent limitations. First, they provide only file-level sharing (i.e., sharing the entire file) and therefore lack object and data management capabilities and support for content-based search. Departing from the existing work on distributed data management, we propose the sharing of data without any predefined schema. Second, many existing P2P data sharing systems are limited as far as extensiblity and flexibility are concerned. As such, there are no easy and rapid ways to extend their applications quickly to fulfill new user needs. Moreover, a node's peers are typically statically defined. Based on the above observations, there is a great need for research on data sharing and query processing in the presence of dynamic peers and heterogeneous data sources.

1.3 Thesis Goal and Contributions

The main goal of this thesis is to consider, outline and figure out a paradigm that includes self-organization, adaptation and fine granularity query support as its intrinsic properties in order to deal with the scale and dynamism that characterize P2P data sharing systems. Therefore, according to the goals to be stratified, this thesis focuses on the following research lines:

1. **P2P Platform** - a platform that facilitates finer granularity data access and sharing.

2. **Query Processing** - the impact of decision making without relying on global knowledge.

3. **Data Placement** - effectiveness of various data placement policies in a network with dynamic participants.

4. **Data Acquisition** - retrieving information from heterogeneous data sources environments.

For this thesis, we have implemented and experimented with a variety of P2P strategies, with the objective of solving the aforementioned tasks. In summary, we have made the following contributions:

1. We have proposed a generic P2P platform, BestPeer, that facilitates fast and easy P2P applications development. BestPeer not only facilitates finer granularity of data sharing where partial content of a file may be shared, but also shares computational power. Our solution incorporates a self-configurable approach, where a node in the BestPeer network can dynamically reconfigure itself

by keeping peers that are most beneficial to it.

2. We have extended the BestPeer architecture to support data management in P2P environments. We have proposed PeerDB, which is a full-fledged data management system that supports fine-grain content-based searching. PeerDB incorporates the use of Information Retrieval (IR) techniques that enables peers to share data without relying on a global shared schema.

3. We have presented new data placement strategies for P2P systems, particularly, for data warehousing applications. PeerOLAP acts as a large distributed cache for OLAP results by exploiting under-utilized peers. When a query is issued, the initiating peer decomposes it into chunks, and broadcasts the request for the chunks in a fashion similar to Gnutella. However, unlike Gnutella, PeerOLAP employs a set of heuristics in order to limit the number of peers that are accessed. Missing chunks can be requested from the data warehouse. PeerOLAP also supports the adaptive reconfiguration of the network structure, which results in reduced query costs. The system maintains statistics for the most frequently accessed peers. Each peer, at regular intervals, reconsiders its set of neighbors and stays connected to the most beneficial ones.

4. We have proposed a heuristics-based method to support content-based similarity queries on ad hoc P2P networks. FuzzyPeer deals with the problem of retrieving information from P2P networks without limiting itself to only exact key matching queries. Due to the absence of centralized indexing in FuzzyPeer, it is difficult to predefine a unified terminating criterion that is optimized for all queries. We have addressed this issue by introducing the freezing technique:

some queries are paused and attached to answer streams from similar concurrently running queries, since the answers to both queries are expected to overlap. We have proposed a simple yet efficient distributed optimization algorithm, which improves the scalability and the throughput of the system. Numerous applications, including full-text search in large archives or fuzzy queries in distributed multimedia repositories, can benefit from our techniques. We have demonstrated this with a case study of an image retrieval application.

1.4 Organization of the Thesis

The thesis is organized as follow:

- Chapter 2 gives a general introduction and discusses related work in the field.
- Chapter 3 describes the basics of the BestPeer platform, its architecture, and its features that ease P2P application developments and overcome the limitations of existing P2P systems. The chapter also presents an overview of the BestPeer network, the relationship of each peer, and the message routine protocol of the BestPeer platform. The performance study of the BestPeer architecture is also presented.
- Chapter 4 provides a description of our proposed P2P-based data sharing and management system (PeerDB). In the chapter, we cover the mechanism of finding data without any predefined global schemas using an IR-like technique. It also introduces the two steps of agent-assisted query processing. The performance study on the effectiveness of the proposed method is also presented.

- Chapter 5 discusses our proposed technique for supporting OLAP applications with the advantages of P2P technology. The chapter introduces the architecture of PeerOLAP and discusses several heuristics of query processing methodologies and data replacement policies. Extensive experiments that have been conducted are presented in the chapter.

- Chapter 6 provides a description of our proposed FuzzyPeer. It presents the architecture and concept of "frozen queries". In the chapter, we discuss the two different query processing techniques, Adaptive Query Freezing and Similarity Query Freezing. In a case study, we also investigate the support for multiple-feature queries, which is particularly useful for multimedia applications. The performance study pertaining to the proposed schemes is presented.

- We conclude in Chapter 7 with a summary of our contributions. We also indicate directions for future work.

Chapter 2

Related Work

2.1 Introduction

Peer-to-peer (P2P) computing is not a totally new concept. It has existed since the beginning of distributed computing. With the advent of powerful computing resources, a new breed of P2P technology has emerged. P2P has been studied extensively in recent years partly due to the popularity of the Napster system that has caught the attention of millions of Internet users. The incredible popularity of the system has drawn many researchers to further study the various issues of P2P systems. In this chapter, we review several topics related to our work. In order to gain a better understanding of the P2P system, we shall start with the taxonomy of computing systems and look especially at P2P in the hierarchy. Next, we will briefly introduce some prior works in P2P from the perspective of their architectures and resources allocation. The fruitful of the facilities provided by the P2P community can potentially be reused by other disciplines, for instance in agent development. Agent computing provides developers with a way to define problem-solving computation at an abstract level, whereas, the key strength of current P2P development centers on

resources gathering and defining efficient resource locating strategies. The integration of the two paradigms is required for the development of self-evolving, open and scalable systems. Thus, we will discuss broadly the different ways of integrating the two paradigms. Finally, we will review P2P from the point of view of database research, specifically describing its complexity and some current solutions.

2.2 P2P Taxonomies

There are many ways to classify computing systems. In this section, we are particularly interested in classifying them according to their role and organization. In general, computing systems can be classified into two main categories, namely centralized and distributed. Milojicic et. al. [72] present a taxonomy of computer systems from the P2P perspective as in Figure 2.1.

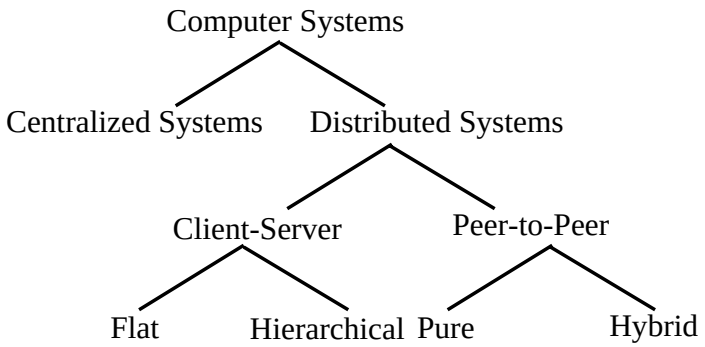
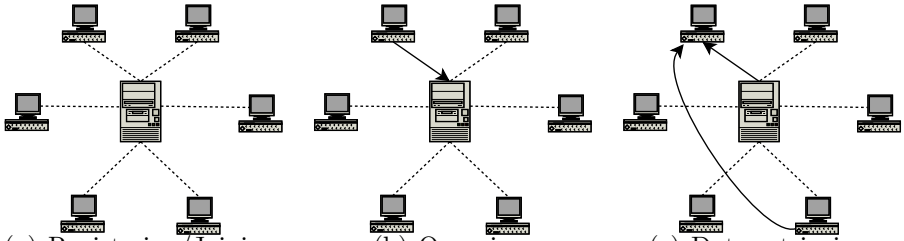


Figure 2.1: A Taxonomy of Computer Systems

Distributed computing can be divided into two models: client-server and P2P. The client-server model can be further classified into the flat and hierarchical models. In the flat model, all clients are equal and they only communicate with a single

server. Examples of a flat model include traditional middleware solutions, such as the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) standard [81], where there are object-request brokers and distributed objects. Many CORBA implementations have been developed and are commercially available, for example Visibroker [4] which has developed by Borland, Voyager [41] by ObjectSpace and WebSphere [5] by IBM. In contrast with the flat model, the servers of one level in the hierarchical model are clients of higher-level servers. Examples of a hierarchical model include the DNS server and mounted file systems [76]. More recently, the concept of the hierarchical model is employed in web proxy caches such as Squid [99].

The P2P model can either be a pure model or a hybrid. Napster [75] is one of the famous P2P systems that utilize the hybrid model (some literature may refer to it as the centralized server model). In this architecture, there exists a central server, which is responsible for maintaining indexes on the meta-data of all peers in the network. Figure 2.2 depicts the architecture of this category. The central server



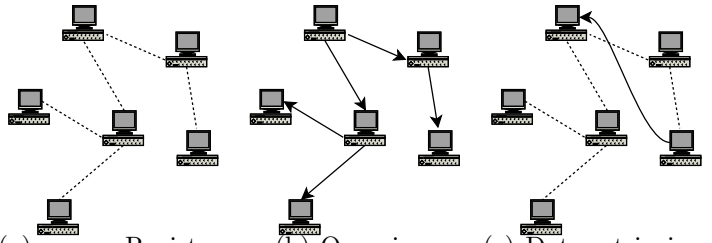
(a) Registering/Joining. (b) Querying. (c) Data retrieving.

Figure 2.2: Centralized P2P Architecture

maintains a master list of all the meta-data of peers in the network. This meta-data is used for describing the data housed in the peers and it may include file names, IP

addresses, line speed, etc. However, the data is located in the peers. Peers upload only the meta-data of its local data to the server on startup, but not the data (see Figure 2.2(a)). In order to locate resources, queries are sent to the central server and the server performs database lookup for each query (see Figure 2.2(b)). The query results, including the locations of files and ping numbers, user names, file sizes, bit rates and other relevant information, are sent back to the peer which initiated the query.

In this case, the servers are simply playing the role of answering queries and indexing the meta-information submitted by connecting peers. However, this model differs from the traditional client-server model. In this model, there exists interaction among the peers to get a job done. While the hybrid model uses a centralized server to perform part of its job, there is no centralized server in a pure P2P model. They are completely decentralized in organization, with each peer playing an equal role. Examples of a pure P2P model include Gnutella [42] and Freenet [39]. Figure 2.3(a) illustrates the architecture. A node joins the network by "connecting" to any of the nodes in the network. Most of the existing pure P2P systems, e.g., Gnutella, employ the message propagation approach as their routing strategy, while others such as Freenet, employ distributed catalogues to avoid flooding the network and to reduce traffic. Figure 2.3(b) illustrates the search strategy adopted in Gnutella. A query node submits its search query to neighboring nodes, which in turn forward the query to their neighbors. This process continues until all the peers receive the query (assuming Time to Live (TTL) has not expired, TTL decreases with every hop it passes through, and expires when it equals zero). If a peer has a match for the query, it will transmit the meta-data (e.g., file name, location, file size, etc.) along

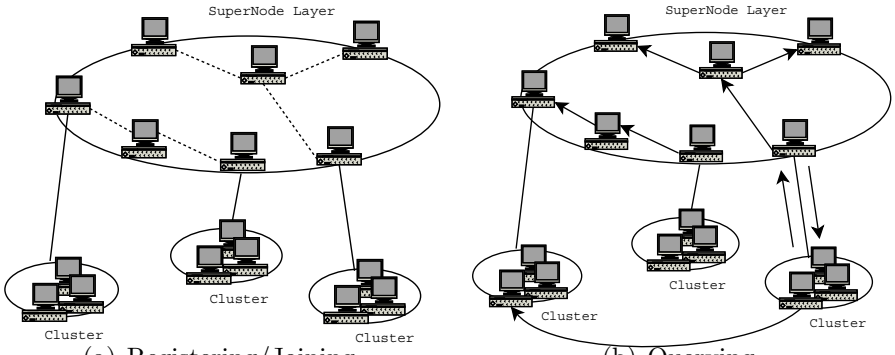


(a) Registering/Joining. (b) Querying. (c) Data retrieving.

Figure 2.3: Fully Autonomous P2P Architecture

the original path to Peer A. However, the actual data downloading is done out of the network (Figure 2.3(c)).

In addition, there are intermediate solutions for the pure P2P model where the SuperNode architecture is employed. The P2P architecture with supernodes [111] is structured hierarchically, and it consists of a supernode layer and a "normal" peer layer (Figiure 2.12(a). Peers in the supernode layer are assumed to be more stable and have more processing capabilities. An example of such an architecture is Morpheus [74], where peers are automatically elected to become supernodes if they have sufficient bandwidth and processing power. Normal peers upload their shared file meta-data to the selected supernode on joining the network. Each supernode maintains indexes for several normal peers, and together, they form a local cluster. A search query will first be sent to the supernode that the peer is connected to (as in the centralized model). The supernode then searches its own database, check whether it can be answered within its own cluster, and at the same time, propagates the query message through the supernode layer with the intention of finding more results. Queries are generally routed and propagated only within one supernode layer. Figure 2.12 illustrates the search process. In Table 2.1, we show a comparison of these



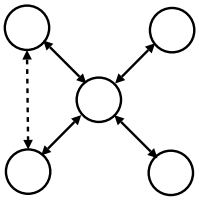
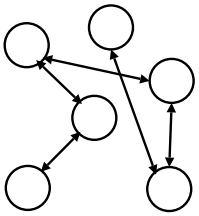
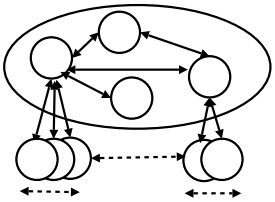
(a) Registering/Joining. (b) Querying.

Figure 2.4: P2P with Supernodes

three different P2P architectures: centralized servers model, fully autonomous model and supernode model.

2.2.1 Comparison of Architectures

Table 2.1: Three Different Architectures of P2P

	Centralized servers	Fully autonomous	Supernode
Definition	Indexing is centralized, but data is distributed.	Indexing and data are distributed.	Hybrid of the previous two.
Graphical view (*solid line and dashed line respectively denote direct and ad hoc connection*)			

	Centralized servers	Fully autonomous	Supernode
Representative system	Napster	Gnutella	Morpheus
Network topology	Flat and frequently changing topology, caused by frequent logon and logoff of the peers. Uses centralized, proprietary servers.	Flat and frequently changing topology, caused by frequent logon and logoff. No centralized, proprietary servers; totally decentralized.	Hierarchical and frequently changing topology. Supernodes tend to have higher capacities. Each supernode maintains several peers (supernode cluster).
Routing	Central database which holds indexes. Clients(connect to this server, search the index and learn from which clients they can retrieve files.	Query message propagated through the network with TTL as life time control. Message is forwarded from a peer to its neighbors if its time has not lapsed. Each node that has requested objects passes back its result set.	A peer sends a request to its assigned supernode. Supernode first searches its own database while probing other supernodes.
Advantages	Centralized control; easy to implement and optimize.	No single point of failure; more robust and comprehensive.	More responsive than the fully autonomous P2P architecture; better load balancing and less single point of failure than P2P with centralized servers.
Disadvantages	Single point of failure; Vulnerable censorship.	Expensive search cost; more traffic on the network.	Single point of failure, though not too severe.

2.3 Search Mechanism and Algorithms

In general, the search mechanism in P2P systems can be categorized into two main components: resource locating and query routing. Together, these two components pose fundamental problems in resource sharing. The design of the search mechanism in a P2P system will affect the performance of the overall system. In resource locating, given a resource *id*, the challenge is to locate the resource in minimal time to yield better performance and response time. In contrast, query routing focuses on optimizing the cost of the query being routed to the next peer in order to achieve minimal time or bandwidth. The first step toward solving this problem is to have a centralized model of resources sharing [75]. However, there are problems with using a centralized server including having a single point of failure. In addition, maintaining a unified view is computationally expensive and scaling up can be a serious problem.

In the following survey, we focus on routing and search strategies in a decentralized environment. As presented in [9], the routing and search problem in P2P computing is defined as follows: Given a set of peers, $P = \{p_1, ..., p_n\}$. Each peer p_i has an address p_i^r storing resource object r that can be identified by a key k. In order to locate a peer that has resource r, we have to search for key k in the lookup table consisting of tuples of form (k, p^r). The information (k, p^r) is distributed over the peers and each peer stores some of this information locally. Let $p \rightarrow locate(k)$ denote the search request for k that can be addressed to every peer with the address p. If a peer gets a request for information that is not locally available, it routes the request to another peer $p' \rightarrow locate(k)$. Clearly, selecting p' becomes an important issue then; the selection process is called a routing strategy. Many routing strategies have been proposed in the literature. In the following section, we first classify them into different

categories and then describe in detail the representative system for each category.

Breadth-first – Gnutella [42] is a pure P2P system and performs search by breadth-first traversal (BFT) of the nodes around the initiator peer. Each peer that receives a query propagates it to all of its neighbors up to the maximum number of hops (Figure 2.5). Each peer that has matching terms passes back its results set. To save on bandwidth, a peer does not have to respond to a query if it has no matching items.

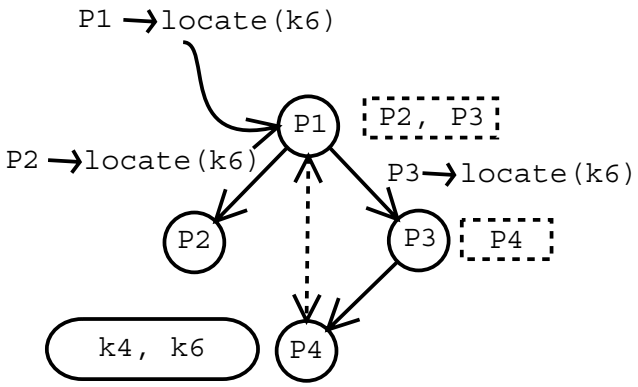


Figure 2.5: Breadth-first Routing and Locating; Dash-box Denotes Routing Table, Oval-box Denotes Local Shared Objects, Dash-arrow Denotes Download

Gnutella is completely decentralized. Its cost of information routing is low and it is very robust. Peers are organized loosely and no global knowledge is required. The advantage of BFT is that by exploring a significant part of the network, it increases the probability of satisfying the query. The disadvantage is the overloading of the network with unnecessary messages. Moreover, the search cost of this routing technique is *O(N)*, and therefore it is affected by the size of the network. Yang and Garcia-Molina[110] observed that the Gnutella protocol could be modified in order to reduce the number of nodes that receive a query, without compromising the

quality of the results. They proposed three techniques: (i) *Iterative Deeping*, where multiple BFTs are initiated with successively larger depths, until either the query is satisfied or the maximum depth d is reached. (ii) *Directed BFT*, where queries are propagated only to a beneficial subset of the neighbors of each node. Several heuristics for selecting these neighbors are described. This method is extended in [25] with the maintenance of summarized information on the neighbors' contents.

Depth-first – Freenet[39] uses depth-first traversal (DFT) up to depth d. Each node forwards the query to a single neighbor and waits for a response before contacting the next one. One of the main characteristics of Freenet is the preservation of anonymity among peers. It uses the 160-bit SHA-1 [SHA-1] as its hash function to generate the key for each file that stores information in the system. Freenet provides varieties of mechanisms to generate the desired hashes, but the simplest is derived from a short descriptive text string chosen by the user, which is referred to as a keyword-signed key (KSK). The descriptive text string is then used as input to generate a key pair: public key and private key. The public key becomes the file identifier and the private key is used to sign the file to provide some form of file integrity check. However, KSK is unable to prevent two users from independently choosing the same descriptive string for different files. This problem is addressed by introducing the signed subspace key (SSK) scheme, which allows a user to create a personal namespace. The namespace is then used as input to generate a key pair as before. The public namespace key and the descriptive string are hashed independently, XOR'ed together, and then hashed again to yield the file key. The descriptive string, together with the subspace's public key, is then made available to the outside world for retrieving the file. The third type of key is the content-hash key (CHK), which is simply

derived by directly hashing the contents of the corresponding file.

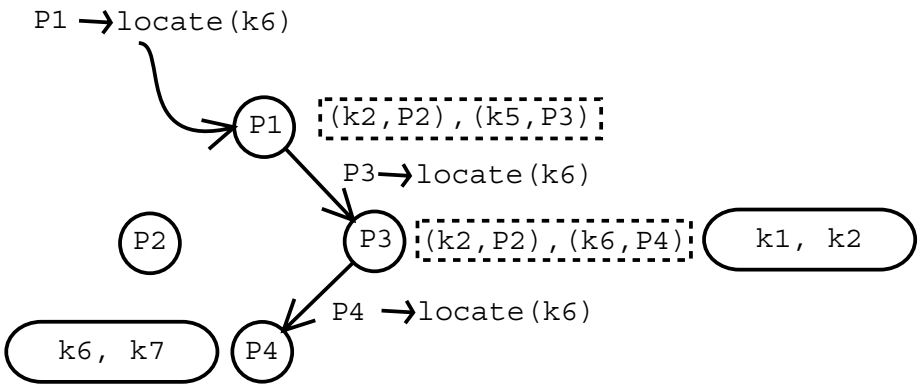


Figure 2.6: Depth-first Routing and Locating; Dash-box Denotes Routing Table, Oval-box Denotes Local Shared Objects

Each peer knows a fixed number of other peers and the keys that they store. The keys are used to assist in the routing of query messages (Figure 2.6). For query optimization, Freenet attempts to cluster files with similar keys into a single node. Hence, search requests are routed to the peer with the most similar key. The next similar key is used if the process does not yield any successful search result. When a file is successfully located, it is passed back and replicated. The file's key is inserted into a local routing table as a successful result. Based on this mechanism, popular files become highly replicated for more accessibility.

Like Gnutella, Freenet is fully decentralized and supports only equality search where the exact keys need to be known, e.g., published in a common access directory. However, in contrast to Gnutella's BFT approach, a query that is submitted by an initiator peer in the Freenet network will be propagated to one of its peers, where there will be a wait for a reply before the query can be forwarded to another peer. If

there is no reply, the initiator peer selects a new peer to process the query. Depth-first traversal has the advantage of minimizing the number of messages used in object locating, but it increases the response time as messages are not able to propagate in the network concurrently – unlike in BFT.

Implicit Binary Tree – Chord[100] is a distributed lookup protocol that supports fast data locating and allows node joining and leaving as a natural process. Each peer is assigned a binary key of length m as its *nodeID* p, usually obtained by hashing its IP address, p=SHA-1(IP). All the *nodeID*s are mapped onto a virtual one-dimensional circle of $N = 2^m$ possible entries according to their *nodeID*s. For each *nodeID*, the first physical peer next to it in a clockwise direction is called its successor node, denoted by $successor(p)$. Likewise, the predecessor node is the first physical peer next to it in the anti-clockwise direction on the identifier circle, and is denoted by $predecessor(p)$ (see Figure 2.7).

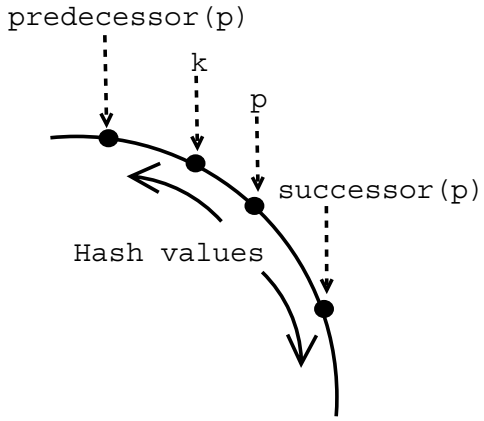


Figure 2.7: Relationship of $predecessor(p)$, $successor(p)$, k and p

On the other hand, each data item key is also assigned an m-bit ID, k, by hashing the key where k=SHA-1(key). Both *nodeID*s and *keyID*s are uniformly distributed

and exist in the same ID space. Each peer with hashed identifier p is responsible for all hashed keys k such that $k \in]predecessor(p), p]$.

In order to support efficient routing, each peer p stores a "finger table" which consists of the first peer with hashed identifier p_i such that $p_i = succ(p + 2^{i-1})$ where $1 \leq i \leq m$. Two important properties can be derived from this scheme. First, each node only stores information about a small number of other nodes. Figure 2.8 depicts the property with m=4 and each label in the circle indicating an entry for finger table in p. The furthest $succ(p)$ is $succ(p+8)$ which is the identifier that is located directly opposite of *p+1* in the identifier circle. Note that a peer knows more about nodes following closely on the identifier circle than nodes farther away. Also, a node's finger table generally does not contain enough information to determine the successor of an arbitrary key k. The routing algorithm needs *O(logN)* hop in order to find the target destination.

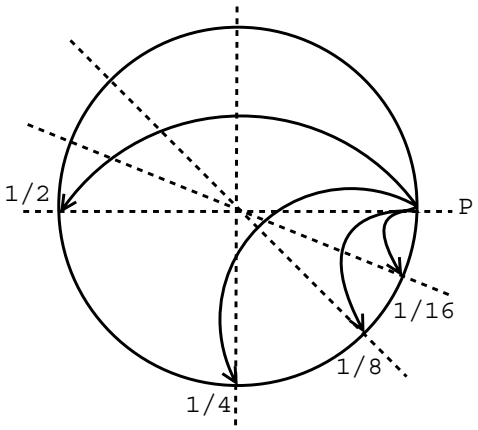


Figure 2.8: Key Assignment in Finger Table

When querying for a record with key k, the virtual position in the identifier circle is first calculated by hashing the key k. The query can start from any physical machine,

node n. Node n searches its finger table for node j, which has an ID most immediately preceding k. The query will be routed to the node j to identify the next node having an ID that is closest to k. The process is repeated until key k is located. For each hop, the distance between the target and the current nodes in the Chord system will decrease by half. Thus the routing time of Chord is *O(logN)* hop, where N is the number of nodes in the network.

Considering the example in Figure 2.9, suppose node *P3* wants to locate *k1*. Since *k1* belongs to the circular interval [P7,P3), node *P3* therefore checks the successor of entry [P7,P3) in its finger table, which is *P0* in this case. Because *P0* precedes *k1*, node *P3* will ask node *P0* to find the successor of *k1*. In turn, node *P0* will infer from its finger table that *k1*'s successor is the node *P1*, and return node *P1* to node *P3*.

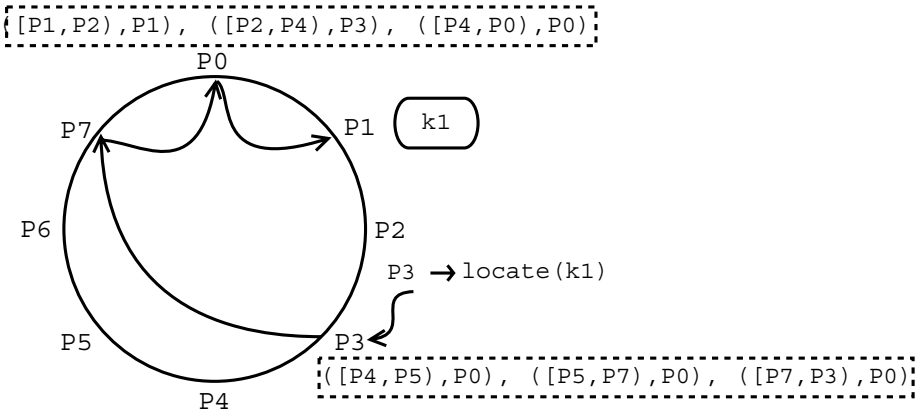


Figure 2.9: Chord Routing Strategy

D-Dimensional Space – CAN Content-Addressable Network (CAN) [92] is a distributed hash-based infrastructure that hashes keys into points in a d-dimensional virtual space. The point indicates the virtual position for the data. The virtual space

is partitioned into many small d-dimensional "zones", with a peer serving as owner of the zone. An object O is mapped to a key $k(O)$ in the space by a hash function. A peer P responsible for object O is the one which has key $k(O)$ in its zone. In the d-dimensional space, two nodes are considered neighbors if their coordinate subspaces adjoin each other. The d-dimensional space of the CAN architecture is illustrated in Figure 2.10. In this figure, a two-dimensional coordinate virtual space is partitioned into seven district zones and owned by seven different peers; A, B, C, D, E, F and G. For example, peer D owns an X-Y coordinate zone of X [0-0.5] and Y [0.51-1]. In addition to the self-zone information, neighboring information such as the coordinates of the neighbor set is stored in each peer to facilitate routing between arbitrary points in the space. For example, peer B and peer D are identified as neighbors of peer A since their coordinate subspaces adjoin each other.

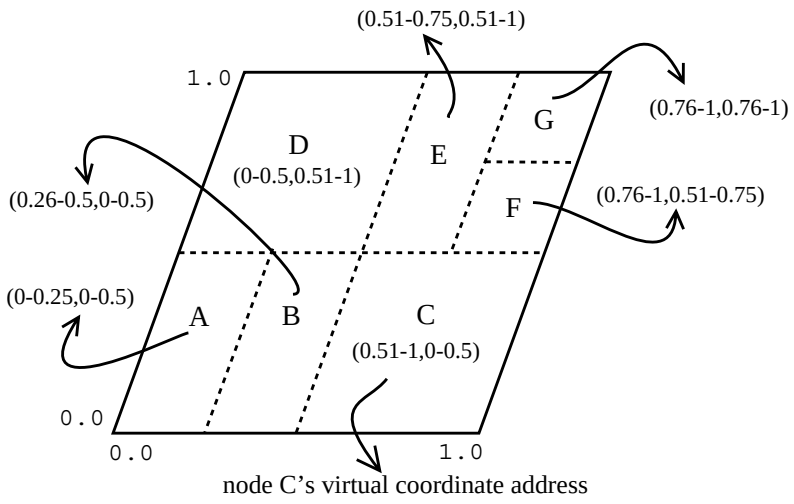


Figure 2.10: 2-D Coordinate Overlay with Five Nodes

Using its coordinate neighbor set, CAN applies a greedy forwarding methodology to the peers in the closest zones as its routing strategy (Figure 2.11). In a d-dimensional space, each node maintains 2d neighbors and the average routing path length is $(d/4)(n^{1/d})$ hops.

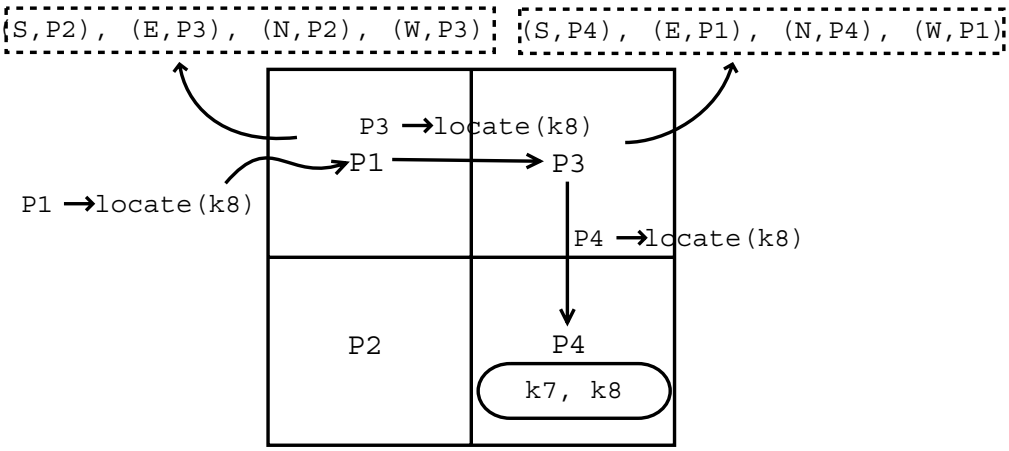


Figure 2.11: CAN Routing Strategy

Some refinements have been proposed to increase the robustness of the system. The entire d-dimension space can be replicated to create two or more "realities". In each reality, the same set of information is stored and maintained by different peers. In other words, the redundancy of a pointer to a piece of information increases the robustness of the system. To improve the fault tolerance of the system, CAN proposes the overloading zone approach in which different peers are responsible for the same zone. Splits are only performed if a maximum occupancy (e.g., four peers) is reached.

2.3.1 DHT-based Schemes: The Limitations

CAN and Chord are distributed indexing schemes using hashing mechanism (DHT) to locate content. Other frameworks such as Tapestry [114] and Pastry[31] are built on DHT mechanism too, but they use different techniques to spread (key, value) pairs across the community and deploy different query routing strategies. These works demonstrate certain important points concerning data placement and site selection. They all have $logN$-like performance in the lookup operation.

However, we observe that these systems have several limitations. In general, all of the DHT-based systems using a uniform hash function perform object location selection and retrieval. Through the careful arrangement of the index structure, lookup queries for an object k will be routed incrementally from a node to another that has an ID that is closest to k. Although the DHT-based approach does provide guarantee of performance and can help locate content deterministically, it also has potential drawbacks. First, it has poor usability due to a lack of semantic flexibility. Similarly, the measurement between two objects is determined by a predefined distance metric (e.g., obj_1 equals obj_2 if and only if both have the same keys). This can be easily visualized with the following example based on the Tapestry [114] routing strategy. Tapestry is based on a longest suffix protocol that selects the next hop to be the peer that has a suffix that matches the desired location in the greatest number of positions. In a formal definition, a suffix routing from A to B at h^{th} hop, arrives at the nearest node hop(h) such that hop(h) shares a suffix with B of length h digits. For instance, a query from nodeID 5324 will be routed to nodeID 0629 with traverses over the following routing path 5324 $\rightarrow$ 234$\underline{9}$ $\rightarrow$ 1$\underline{429}$ $\rightarrow$ 7$\underline{629}$ $\rightarrow$ $\underline{0629}$. Since each of the IDs is generated via a uniform hash function, there are no semantical meanings

or relations defined in between any hash values. Second, how objects are chosen is predetermined due to the structured arrangement of DHT-like schemes. Importantly, applications are not allowed to choose an operator to define how objects are being selected. As a result, there is no easy way of supporting complex queries in DHT-like schemes. Third, it is conventionally assumed that random keys are mapped to a single peer (i.e., the peer is then responsible for storing all the contents of the files that are associated with the keys). A random choice of keys results in an *O(logN)* imbalance[91, 19] factor in the number of items stored at a peer. Finally, users may lose control of the objects they offer to share. Since all objects have to be placed in predetermined hosts (which are usually remote hosts), it is almost impossible for the user to keep his/her sharable objects locally, unless the object identifier is mapped to the local host identifier. This is a violation of the P2P philosophy of peer autonomy.

2.4 Agents and P2P Computing: A Promising Combination of Paradigms

As mentioned earlier, the key strength of the current P2P development is that the community provides varieties of routine strategies for efficient resource locating. The earlier work follows centralized models of resources sharing, such as Napster [75]. Perhaps this centralized architecture is most similar to the existing development of multi-agent systems [56, 55, 106, 80]. For example, the Concordia platform [73, 23] developed by Mitsubishi Electric provides support for Java-based mobile agents. Agent mobility is achieved via Java's serialization and class loading mechanisms. Each agent object is associated with a separate Itinerary object, which specifies the agent's

migration path (using DNS hostnames) and the methods to be executed at each host. In [55], the Aglets environment allows the creation of a group of agents that could work cooperatively to solve a complex task. In [56], Ajanta, a Java-based system that supports agent mobility, makes use of Java's serialization for state capture. The agent code is loaded on demand, from an agent-specified server. In all these systems, the agents are required to contact a centralized resource manager to locate services.

2.4.1 Merging of Infrastructures: P2P and Agent

Agent and Peer-to-Peer (P2P) are two paradigms that realize the real power of computing through autonomous, distributed and dynamic systems. These systems are becoming increasingly popular as they enable users to exchange digital information and share in problem-solving by participating in complex networks. In particular, many researchers consider the agent system as an autonomous problem-solving entity while P2P provides support for resources pooling. Merging these two disciplines by adopting the best of each approach could potentially provide an ultimate solution that is inexpensive, easy to use, self-learning, self-modifying, highly scalable and needing no central administration.

To deal with the autonomy, scale and dynamism that characterize P2P and agent systems, a merged paradigm is required and it should embody the following intrinsic properties: self-organization, self-adaptation, automated information matching, and support for discovery.

Given the respective infrastructures of P2P and agent technologies, from the design point of view, the key to facilitating the success of future developments of agent and P2P lies in a neat integration of both technologies. On the one hand, the main

focus of agent technology is relevant to an abstract level of interaction, negotiation, content analyzing and domain-specific protocol handling. On the other, P2P is particularly focused on meeting the challenges of scalability, robustness and effectiveness of message routing at the lower level. The core mission of the infrastructure merger is to ensure that the merged infrastructure is inter-operable between P2P and agent technologies.

There are three broad approaches to merging the two technologies. One is based on integrating P2P technology to underlie agent systems (the left image of Figure 2.12). For instance, a DHT-based [92, 100, 31] routing strategy could be integrated into an agent system for efficient agent routing. This approach is more agent-oriented since it defines P2P as a subset of tools to facilitate efficient routing by agents. The second approach is a P2P-oriented merging strategy, where the main idea is to build a proprietary software agent on top of an existing P2P system (the right image of Figure 2.12). The third approach operates on three tiers, with a middleware in between the agent and P2P layers (the centre image of Figure 2.12).

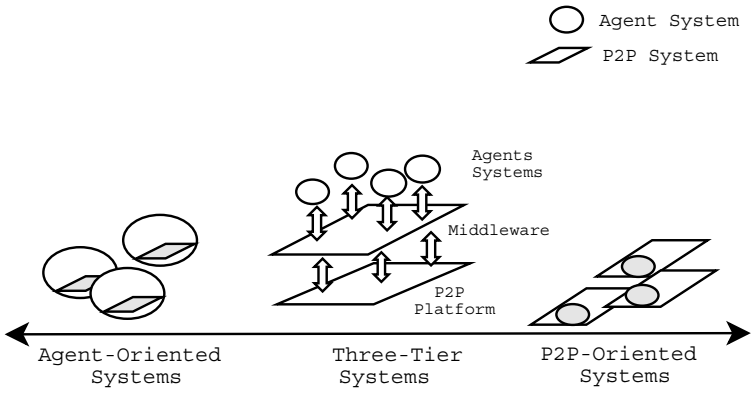


Figure 2.12: Infrastructure of P2P and Agents

Most of the existing agent systems provide support for agent collaboration and communication but are not native to P2P technology. The development of P2P applications based on these platforms would require a longer and more costly effort. There are several reasons that suggest the limitation of applying a traditional agent system in a P2P model. First, traditionally, mobile search agents perform search operations by moving themselves to the site containing the target information and executing a given task. The agent's path is either predefined or the agent has knowledge of where to find the services. For example, in order to find the cheapest airfare, a travel agent is given a set of sites that provide airfare query services. The agent's programmers have to know where the agent needs to go and where the next destination is after the task at a site is completed. However, this may require a predefined knowledge of the environment – which is not always be feasible, e.g., there may not exist any predefined knowledge of who is offering a particular service and where. The problem may be solved by integrating P2P query routing strategies into agent systems to form agent-oriented systems. Obviously, the main drawback concerns the extensibility of the system, for each upgrade of the services, e.g., incorporating new routing strategies or new P2P services into the system, will cause a major disruption of the system. Moreover, the whole architecture may possibly become fatter, which may in return result in unpredictable behavior. Also, there may exist several agent systems with P2P support but which are unable to communicate with each other. This may be due to the fact that they employ either different agent communication languages or different P2P protocols. In apparent recognition of this problem, the agent community has started to standardize agent communication languages such as in KQML [37] and FIPA ACL [98]; meanwhile, P2P is still evolving.

P2P-oriented system mergers have inherited issues that are similar to those faced by the agent-oriented approach. Rather than incorporating existing agent systems to facilitate extensibility in the functionalities of P2P systems, specially designed agents may provide assistance. This paradigm may be useful in a specific corporate environment where the predefined protocol and languages have been set up as in the agent-oriented approach. The two approaches that have just been discussed tend to be closed systems rather than sustainable ones that could adopt any future publicly-advertised standards.

The alternative solution – which is the third approach to the merger of agent and P2P technologies – operates at the following three tiers: 1) an agent system running on the peer to provide application-related services, 2) a P2P platform to handle communication and the necessary message routing strategy, and 3) a middle tier that handles the communication between the agent and P2P layers. Each tier focuses exclusively on its assigned tasks. For example, when a new P2P routing strategy is invented, only the P2P layer needs to be updated. Similarly, to accommodate large numbers of participants, only the middle tier needs to be scaled by employing industry agreed protocols and languages. Such an approach would help to develop a fully open and truly scalable distributed data sharing system that supports dynamic networking and heterogeneity in the data environment.

In Chapter 3, we shall discuss BestPeer, the working prototype of an integrated agent-P2P system that is being developed to serve as a platform on which P2P applications can be developed easily and efficiently using agent technologies.

2.5 P2P: From the Data Management Perspective

In this section, we shall review P2P from the perspective of database management, describing in particular, its complexity and some current solutions.

Database management systems (DBMSs) have dominated the marketplace for years. Data is stored and modified, and information is extracted from a central server. This provides an easy-access and controlled environment for the data. Nevertheless, things have since changed dramatically. Most organizations and research community have moved toward distributed DBMS (DDMS). One of the major motivations behind the use of DDMS is the desire to provide an economical method of harnessing more computing power by employing multiple processing elements. Significant achievements have taken place in the development and deployment of DDMS. These include mechanisms to provide transparency in accessing data from multiple servers [35, 34, 103, 22], and the support of distributed transactions to facilitate transparency [89] and execute queries over fragmented and heterogeneous data sources [45, 89].

With many of the challenges in designing DDMS systems seeming to fall under the banner of the P2P paradigm, the paradigm raises many new data management issues and challenges on closer evaluation. Traditional DDMS is designed to run in a stable and manageable environment, which is commonly described as the Distributed Computing Environment (DCE). In network computing, DCE is an industry-standard software technology for setting up and managing computing and data exchange in a system of distributed computers. DCE is typically used in a larger network of computing systems that include different size servers scattered geographically. DCE uses the client/server model. Using DCE, users can access applications and data at remote servers. Application programmers need not be aware of where their programs

will run or where the data will be located. Data integration and exchange between heterogeneous data source are provided mainly through the use of views that map and restructure data between heterogeneous schemas [24, 66]. These programs require the preparation of unifying the logical structures of the underlying data sources so that DCE applications and related data can be located when they are needed for use. In addition, DCE is assumed to be a stable environment of the client/server model, i.e., where the server is accessible 7x24. In DDMS, the case where the server leaves the network and causes the data to be inaccessible is considered exceptional.

In contrast, the P2P environment is dynamic and sometimes ad hoc. Peers are allowed to join the network at any point of time and may leave at will. This results in an evolving architecture where each peer is fully autonomous. With such a dynamic environment, the need of maintaining inter-operability among peers is a great challenge. In addition, finding ways to cope with DBs that are incomplete, overlapping and mutually inconsistent is perhaps the most exciting challenge, and which forces us to significantly extend the previous techniques addressed by the database community.

2.5.1 Complexity of Data Management in P2P

Building systems to solve any of the aforementioned tasks requires that we choose a method for modeling the underlying domain. In particular, in this work, we need to model the P2P system itself. We define the complexity of data management in the P2P system as follows. Assume we are given a set of N peer nodes connected by a network that has limited bandwidth and a variety of data transfer speed. Each node p_i is heterogeneous in terms of storage, processing power, workload and schemas. A peer, p_i, may have data to share with other peers and the database is a relational

database. Therefore, the primary objective is to model, control, store and retrieve data in this complex environment. We focus on three classes of tasks related to the complexity of data management in P2P.

- *Data Modeling and query capabilities*:

  Suppose we view the P2P environment as a directed graph and nodes are peers in the P2P network and every pair of peers are connected by an edge. Assume objects shared by each peer are atomic elements with object identifiers. Objects can exist in several forms in the P2P environment, depending on the specific system, e.g., file elements [42, 39, 75], unit of storage [63], computational power [33, 88, 105], etc. The first task that we consider is formulating queries for retrieving resources in the peer environment. The simplest instance of a query, which is provided by a Napter-like search engine, is to locate objects, for example, MP3 files, based on the filename. In general, the engine supports only one form of object queries: given an object identifier, *oid*, return object *o*. Clearly, this simple model has many limitations for applications which require more complex predicates on the contents of an object, e.g., "*find an image which has filename like "sunshine" and contains "car" shape in the image*". In addition, consider an example of a query asking for the top k similar images between the search space of n hops. The complexity of content search notwithstanding, the effectiveness of queries such as the aforementioned example is highly dependent on the data placement that supports it.

- *Data Caching and Placement*:

  Data placement is the assignment of a set of objects to be stored at each peer in the network. The objective is to minimize the overall execution time of

a program graph in which the peers represent parallel operations and data is communicated along the edges. The objective can either be achieved through minimizing the number of routing hops [31, 92, 93, 100] or maximizing the replication of objects [63, 39]. A data placement may be described extensionally with a global set of *oids* at each peer [75] or by a set of local views for each peer which describes the objects stored at the peer [42, 44, 78, 53]. A hybrid data placement policy such as [74, 111] uses a set of selected peer which act as centralized resources. This set of peers maintain a set of *oids* for a small number of peers, and they are connected to each other to form a pure P2P data distribution network. The cost of data placement is context-specific. Most of the current P2P systems measure cost as the number of application-level network hops. For example, the data read cost in Pastry is *O(logN)* and CAN is proved to be *O(N1/d)* (refers Section 2.3).

- *Schema Mediation and Data Integration*:

  Varieties of data may exist in each peer's data repository, e.g., images library, music files or document collections. Since these data are related in some way i.e., semantically, it is possible to integrate the diverse data stores under one uniform and homogeneous view. Conventional schema mediation such as the GARLIC [22] and DISCO [103] systems require wrapper programs and these systems presume all participant hosts are willing to share their schemas (if they exist). Moreover, tight cooperation is required for programs and queries translation. In essence, we find that this assumption is not desirable in the sense that it requires close cooperation among peers (while some peers may refuse to disclose their schemas for privacy reasons). Furthermore, the assumption may

not be feasible due to the limitation of resources and dynamism of peers in the network [90, 44, 78]. Even worse, non-database systems handle data in an application-specific format, causing the problem of integrating broad non-standard data into a database environment.

In the following section, we survey various approaches and classify them by their dominant way of solving the aforementioned tasks.

2.5.2 Data Modeling and Query Capabilities

In general, there exist two classes of data models used in P2P applications: *Graph Data Models* and *Semi-structured Data Models*.

Graph Data Models

It is natural to represent the data in a P2P network with a labeled graph. Consider a set of N peers $P = \{p_1, .., p_N\}$ with data object $d \in D$. In the graph data model, nodes represent peers ($p \in P$) or the objects shared by the peer ($d \in D$), and arcs represent the relationships among them. Along with the graph model, several paradigms have been proposed in order to support queries over graphs such as *breadth-first* e.g., Gnutella, *depth-first* e.g., Freenet, and *implicit binary tree* e.g., Chord. The details of each implementation can be found in Section 2.3.

Semi-structured Data Model

P2P is an environment that does not have a unified fixed schema that can be applied to all peers. The representation of attributes might differ from peer to peer. Significant

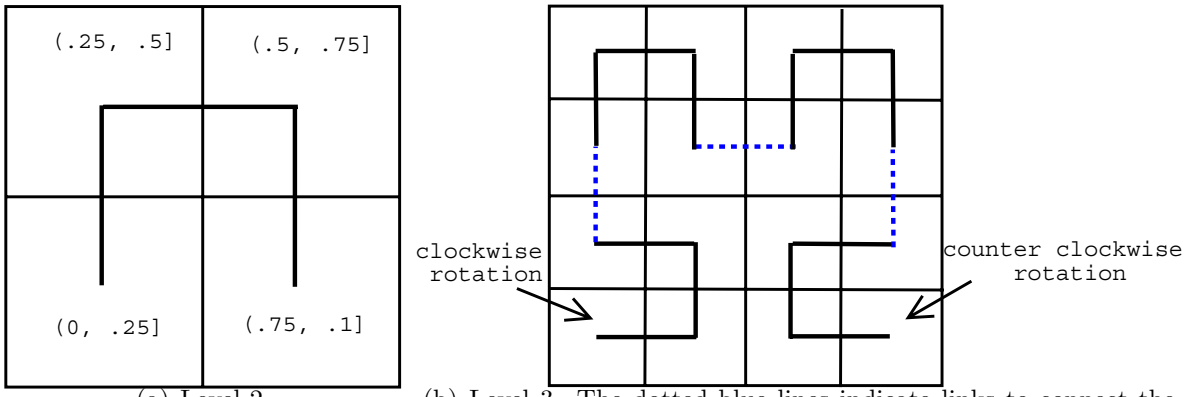
amount of research has been conducted into data transformation or schema integration in static networks, such as the integration of web data sources, data warehouse loading and XML message mapping. To address the automation of schema matching, various techniques aiming at different types of schema information have been devised. In [17, 69, 71, 64, 32], element names, data types and structural properties are exploited. In [29, 30, 70, 32], the characteristics of data instances are examined to facilitate finding semantic correspondences between elements of two schemas, while the proposal in [17, 32] focuses on finding a solution to the problem based on utilizing auxiliary sources, such as taxonomies, dictionaries and thesauri. These models were not developed specifically for the P2P environment. However, these models inherited some of the characteristics which can be taken into consideration when designing P2P data models. Broadly speaking, semi-structured data refers to data with some of the following characteristics [38]:

- the schema is not given in advance and may be implicit in the data
- the schema is relatively large and updates frequently
- the schema is descriptive rather than prescriptive, i.e., it describes the current state of the data, but violations of the schema are still tolerated
- the data is not strongly typed, i.e., different objects with the same attribute may be of different types.

Currently, several works on issues concerning the management of semi-structured data in the P2P environment have been proposed [78, 90, 48]. For example, K. Aberer et al. [8] focus on semantic interoperability in a P2P network with a gossiping technique.

Complex Query Capabilities

Andrzejak and Xu [14] proposed a range queries support for the CAN architecture. CAN is a data structure that is designed for the distributed storing of pairs (key, data) to allow fast locating of data when a key is given in the P2P network (details can be found in Section 2.3). However, CAN does not support queries of ranges. Each discrete value in a range must be queried individually, which is infeasible in most of the applications. In the work, the authors proposed an extension of CAN to support range queries by using the two-dimensional Hilbert curve [15], with R^2 as a hash function. The Hilbert curve is a recursive function that maps the unit interval [0.0, 1.0] to the unit square in the plane.



(a) Level 2.

(b) Level 3. The dotted blue lines indicate links to connect the elements of the Hilbert Space filling curves.

Figure 2.13: Hilbert Curve for Approximation Level 2 and Level 3

Figure 2.13 is a two-dimensional Hilbert curve that passes through every point of the unit square $[0, 1]^2$. Figure 2.13(a) shows the continuous curve with approximation level 2 that has four equally sized intervals. Assume that the attribute values are in the range of [0.0, 1.0]. Each zone corresponds to a certain subinterval of [0.0, 1.0]; (0,

0.25], (0.25, 0.5], (0.5, 0.75] and (0.75, 1.0]. The Hilbert curve with approximation level l+1 can be generated by copying the sub-zones of level l to it shrunk, then possibly rotating the curve for level 1 by 90 degrees, either clockwise or counter-clockwise, and finally by linking together the elements of the Hilbert Space filling curves. Figure 2.13(b) shows the example of approximation level 3.

In order to support range queries in the CAN structure, the authors introduced *interval keeper* (IK) servers. IKs are the subset of peers in the CAN network that respond to a certain sub-interval of [0.0, 1.0] of the attribute values. Each IK owns a zone in the logical d-dimensional space. Two properties have been used to design the mapping between the intervals and the zones. First, if two IKs have close-by intervals, then their zones should also be close by. Second, if an interval I is split into interval I_1 and I_2, then the zones of I_1 and I_2 must partition the zone of I. These properties match well in the Hilbert curve for supporting efficient range queries in CAN.

The query processing is defined as follows: Given a query range attribute value, the mechanism first computes the hypercube determined by the Hilbert function which encompasses all zones of the IKs intersecting the query range. For a range query whose lower and upper bounds are l and u, it first routes to the IK that owns the middle point $(l + u)/2$, and then recursively propagates the request to its neighbors until all the IKs which intersect the query are visited.

2.5.3 Data Caching and Placement

Piazza [44] is the first system to deal with database management issues in P2P systems. In Piazza, each peer can have any of the following four roles: data origin which provides the original content, storage provider which stores materialized views, query

evaluator which uses its CPU resources to evaluate a query, and query initiator which poses new queries to the system. Piazza deals primarily with the data placement problem, i.e., the selection of strategic places to store data in order to improve query performance. Although this is also an issue in distributed databases, there are fundamental differences since P2P systems do not have a centralized schema. In addition, the membership of a peer in the system is ad hoc and dynamic, therefore it is very difficult to predict or reason out the location and quality of the system's resources. In Piazza, the data placement problem is solved by logically dividing the system into smaller spheres of cooperation and advertising the set of materialized views to all the nodes of a sphere.

2.5.4 Schema Mediation and Data Integration

Peer-Programming Language (*ppl*) [44] is a formalism for mediating between peer schemas and it has been used in the Piazza system as described before. It provides decentralized schema mediation especially in defining the mapping expression syntax between schemas and answering queries over multiple schemas. Research on data integration and schema mediation has been extensively studied in the past decade. Generally, two commonly used formalisms are the global-as-view (GAV) and local-as-view (LAV) [11, 20, 104] approaches. GAV is an approach in which the mediated schema is defined as a set of views over the data sources. In contrast, LAV uses the approach that the source relation is defined as the view over mediator. The comparison between GAV and LAV is presented in [66]. The *ppl* combines both LAV- and GAV-style reformulation in a uniform way, and it is able to chain through multiple peer descriptions to reformulate a query.

Bernstein et al. [90] introduce the Local Relational Model (LRM) as a data model specifically designed for P2P applications. LRM assumes a set of peers in which each of the peer is a node with a relational database. It exchanges data and services with *acquaintances*, i.e., other peers. The set of *acquaintances* changes often due to site availability and changing usage patterns. Peers are fully autonomous and there is no global control or uniform view. A peer is related to another by a logical *acquaintance* link. For each *acquaintance* link, *domain relations* define translation rules between data items, and *coordination formulas* define semantic dependencies between the two databases. In [59], mapping tables are proposed for data mapping in the P2P environment. We observe that the notion of mapping table is similar to the notion of domain relations proposed in [90]. They extend [90] by providing domain relation management through capabilities of inferring new mapping tables and determining consistency of mapping constraints. Lenzenrini [66] describes a general framework for modeling data integration applications that can be used to represent P2P applications. There is a sharp contrast between the work in [66] and [59]. The former focuses on expressing constraints on the information contained in a peer whereas the later imposes constraints on the information exchanged between peers.

2.6 Summary

In this chapter, we have provided a brief description of commonly used architectures, routing strategies, data modeling and placement, as well as schemas integration for P2P systems. Specifically, we have noted that the problem of data and resources management in P2P becomes significantly challenging owing to the dynamistic and

heterogeneous nature of the data and resources. A survey of some existing P2P systems has also been presented to demonstrate the importance of such systems in today's technological world.

Chapter 3

The Architecture of BestPeer: A Self-Configurable P2P System

Peer-to-Peer (P2P) has opened up a new area of research in networking and distributed computing. It has been studied extensively in recent years, partly due to the popularity of Napster [75] which has caught the attention of millions of Internet users. Such systems are inexpensive, easy to use, highly scalable and do not require central administration. Despite the advantages offered by P2P technologies, they pose many novel challenges for the research community.

A P2P system is a program that integrates different data sources from multiple remote nodes and forms a virtual resources-rich community. Many systems have been proposed recently [108, 2, 42, 75, 74]. However, most of the existing P2P systems are limited in several ways. First, they provide only file-level sharing (i.e., sharing of the entirety of a file) and lack support for content-based search. Second, they lack extensibility and flexibility. As such, there are no easy and rapid ways to expand their applications quickly to fulfil new user needs. Third, a node's peers are typically statically defined. Fourth, current P2P systems either rely on a DNS server to resolve domain names or deploy a centralized server to maintain globally unique names

[52]. For the former, since a domain name server's entries usually refer to permanent IP addresses, the arrangement reduces the participation of nodes with variable connectivity and temporary network addresses in the activities of peers. For the latter, the server may become a bottleneck. Moreover, like all centralized approaches, such systems are not scalable.

In this chapter, we present our solutions to the above problems. First, we integrate mobile agent and P2P technologies. Since agents can perform operations at the peers' sites, the network bandwidth is better utilized. More importantly, agents can be coded to perform a wide variety of tasks, making it easy to extend the capabilities of a P2P system. For example, while an agent may search for files based on file names, another may perform a content-based search on the file. Second, we incorporate a mechanism to dynamically keep promising (or best) peers in close proximity based on some criteria. For example, peers that are most frequently accessed are directly communicable while nodes that are less frequently accessed can be reached through peers. This significantly reduces the response time to queries. Third, we introduce a location independent global names lookup server (LIGLO) to uniquely recognize nodes whose IP addresses may change frequently. Thus, a node's peer whose IP address may be different at different time remains uniquely recognizable. To avoid the server being a bottleneck, we adopt a distributed approach where several LIGLO exists in the BestPeer network.

We implemented BestPeer, a prototype of the integrated agent-P2P system that incorporates all the above features. It is a three-tier architecture with an agent layer at the top of the hierarchy, a middleware layer in the middle, and a P2P layer at the bottom. The P2P layer is the lowest layer of the hierarchy for supporting

low-level communication and resource-sharing capabilities amongst nodes, and it is self-network reconfigurable. To evaluate BestPeer, we propose a systematic methodology for evaluating P2P systems. Our methodology considers both efficiency and effectiveness (quality of answers) of P2P systems. We conducted our experiments on a cluster of 32 Pentium II PCs each running a Java-based storage manager [43]. Our results show that BestPeer provides excellent performance compared to traditional non-configurable models. We also evaluated BestPeer against the protocol of Gnutella. Our study shows that BestPeer is superior to Gnutella.

3.1 The BestPeer Network

BestPeer is a generic P2P system designed to serve as a platform on which P2P applications can be developed easily and efficiently. Figure 3.1 illustrates a BestPeer network. The network consists of two types of entities: a large number of computers (nodes), and a relatively fewer number of *location independent global names lookup* (LIGLO) servers. Each participating node runs the BestPeer (Java-based) software and is able to communicate or share resources with any other nodes (i.e., peers) in the BestPeer network. Each node comprises two types of data: private data and sharable data. Nodes can only access peers' data that are sharable. Using Figure 3.1 as an example, Peer A can *directly connect*[1] to Peer B to obtain its sharable data, while it can only reach Peer C via Peer B. However, in BestPeer, data are downloaded out-of-network, i.e., a direct connection between Peer A and Peer C is established in order to perform the data transfer (without having to go through Peer B). In addition, messages that are transmitted from the peer to the query initiator need not follow

[1]Note that this is only a logical 'connection'.

the query path.

We shall defer the discussion on the LIGLO servers to a later section. It suffices to say here that they are used to uniquely identify nodes whose IP addresses may change as a result of frequent connection to and disconnection from the BestPeer network. Through the LIGLO servers, a node knows exactly who its peer is; otherwise, the same peer with a different IP address each time it joins the network may be considered a 'new' participant. Strictly speaking, if a node does not care about the identity of its peers, then, it need not use the service of LIGLO servers.

The BestPeer software essentially provides each node with an environment in which (mobile) agents can reside and perform their tasks. This makes the system highly extensible and powerful.

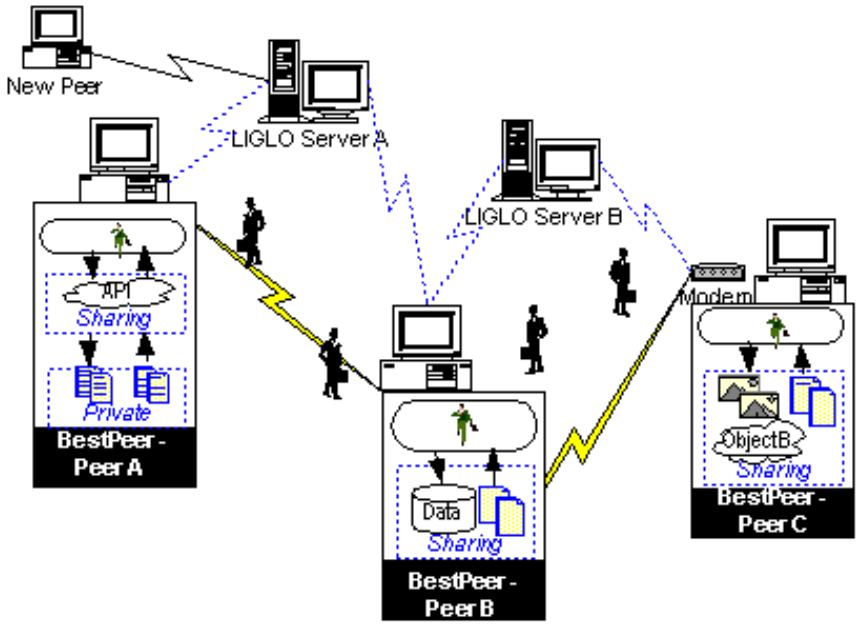


Figure 3.1: BestPeer Network

Now, consider a node (not a registered member of BestPeer) that would like to

become a participant of BestPeer. The process is as follows:

- The node registers with a LIGLO server. This is similar to a user registering with a mail server in the Internet environment.
- The LIGLO server will issue the node with a global and unique identifier, which we shall refer to as BPID (BestPeer ID). This BPID serves to uniquely recognize the node regardless of its current IP address. BPID is essentially a (LIGLOID, NodeID) pair where LIGLOID is the IP address of the LIGLO server and NodeID is a unique ID for the node assigned by the LIGLO server.
- At the same time, the LIGLO server will also send a list of (BPID, IP) pairs that the node can communicate directly with, i.e., the direct peers of the node. Here, the ith BPID value is the identifier of the ith peer, and the corresponding IP value is the current IP address of this peer. We note that since the peer is not obliged to inform LIGLO of its disconnection, the IP address may not be a valid one. In BestPeer, LIGLO will periodically check the validity of its registered participants' IP addresses.
- The node is now a participant of BestPeer and is ready to communicate with any peers (without going through LIGLO anymore).

For a participating node that wants to rejoin the BestPeer network after disconnecting, the process is as follows:

- The node will send its IP address to its LIGLO. This allows its LIGLO to update the IP address if it has changed.

- For each peer of the node, say p, it will send p's BPID to its (i.e., p's) registered LIGLO server. Recall that p's registered LIGLO can be obtained from p's BPID.

- p's registered LIGLO server will reply with the IP address of p if it is currently connected to the network; otherwise, it will indicate that p is now offline. This is necessary for the node to know its peers' new IP addresses if they have been changed. We note that p is not obliged to inform its LIGLO server that it will be (or is) disconnected. As such, the information may not be accurate anyway.

- The node has rejoined the BestPeer network, and is ready to communicate with its peers.

We note that this process is not necessary for a participating node that rejoins the BestPeer network (except to inform the LIGLO server of its new IP address). It can simply communicate with its existing peers. Should the IP addresses of some peers be invalid (i.e., they may have changed their IP addresses or are disconnected), then it can simply replace those peers with new peers that it encounters (based on certain criteria).

Once a node is connected to the BestPeer network, it is ready to share its resources, and has access to other nodes' sharable resources. A node essentially broadcasts its query to its directly connected peers, and its peers then broadcast the message to their peers, and so on. Any nodes with matching results will respond to the initiating node directly.

In BestPeer, there are two modes in which a node can have access to data from other nodes:

1. In the first mode, nodes with matching answers will return the answers directly.

This method can provide fast answers but may result in overloading and poor bandwidth utilization, especially if a significant amount of data is not desirable (e.g., too much overlap, files too large, etc.).

2. In the second mode, nodes with matching answers will only indicate that they have the information, e.g, by returning the file name, etc. The initiating node will then send a further message to some, if not all, of these nodes to obtain the desired information. This mode provides better resource utilization at the expense of a delayed request. Since there is a delay, and the request is initiated by the source of the query, it is possible that the target node may have removed the desired content or updated it during the period of delay.

```
1.  On UserQuery(q)
2.      IF Local Request THEN
3.          broadcast the request with propagation terminating condition
4.          IF q can be satisfied locally THEN
5.              Obtain results and update statistics
6.      IF Remote Request Arrival THEN
7.          IF propagation terminating condition is not met THEN
8.              broadcast q with propagation terminating condition -1
9.          IF q can be satisfied locally THEN
10.             Send Reply //two modes; either results or meta-data/
```

Figure 3.2: Search Algorithm

A search algorithm is shown in Figure 3.2. The algorithm distinguishes between the case when i) a request arrives from the local user, and ii) it arrives from another peer. Time to Live (TTL) is used as the propagation terminating criterion. TTL defines the maximum number of hops that a request may perform.

3.2 Features of BestPeer

In designing BestPeer, we sought to overcome the limitations of existing P2P systems. As such, BestPeer was designed with several distinguishing features:

1. BestPeer combines the power of agent technology and P2P technology in a single system.

2. BestPeer not only facilitates a finer granularity of data sharing where partial content of a file may be shared, it also shares computational power.

3. BestPeer facilitates the dynamic reconfiguration of a BestPeer network so that a node is always directly connected to peers that provide the best service (based on certain optimization criteria such as providing the most number of answers or providing answers most of the time).

4. BestPeer adopts a distributed approach to minimize bottlenecks at servers acting as LIGLO.

In this section, we shall discuss these features in greater detail.

3.2.1 Integration of Mobile Agents and P2P Technologies

BestPeer, to our knowledge, is the first system to integrate two powerful technologies: mobile agent and P2P. While P2P technology provides resource sharing capabilities amongst nodes, mobile agents technology further extends the functionalities. In particular, since agents can carry both code and data, they can effectively perform any kind of functions. With mobile agents, BestPeer provides not only files and raw data, but also processed and meaningful information. For example, in BestPeer, an agent

can be sent to a peer which has the data file being sought after, to "digest" its content and generate reports for the requester.

In BestPeer, we have implemented our own Java-based agent system instead of using existing systems (e.g., [65]). Like the existing systems, both the agent and its class have to be present for the agent to resume execution at the destination engine. Thus, if the class is not already at the destination node, the class has to be transmitted also. For the moment, we have adopted a purely "code-shipping" strategy where a node will always perform its operation at the destination node (where the data resides). This is a reasonable approach as it exploits parallelism by enabling all peers to operate on their data simultaneously; otherwise, the node will become a bottleneck.

More importantly, the use of agents allows BestPeer nodes to collect information (e.g., what files/content are sharable, statistics, etc.) on the entire BestPeer network, and this can be done offline. This allows a node to be better equipped to determine who should be its directly connected peers or who can provide it with better service.

Traditionally, mobile search agents perform search operations by moving themselves to the site containing the target information and executing a given task. The agent's path is predefined. The agent's programmers have to know where the agent need to go and where the next destination is after the task at a site is completed. Another problem with the traditional agent approach is that when an agent has more than one directly connected host, the agent's developers have to decide which path to follow and then keep track of it. When the network grows more complicated, searching through the network becomes a nightmare.

BestPeer adopts a different strategy. It solves these problems by providing a

simple interface to search all directly and indirectly connected hosts. An agent's path is transparent to the agent's developer. An agent is sent to all connected hosts automatically without a statically defined path. The agent is cloned and sent to all the connected hosts in parallel. The process of cloning and forwarding will keep on going until the agent's lifetime expires. The lifetime of an agent is determined by Time-to-live (TTL) and Hops variables. It is similar to other packet approaches used in the networking environment. Once an incoming agent is received, and if the agent has not expired (if TTL $>$ 0), the remote host will decrease the TTL values of an agent before sending it to other hosts that it is directly connected to. Hops variable will be increased at the same time too. The seemingly superfluous use of TTL and Hops together is to enable hosts to drop any incoming agent that already has a copy on site.

3.2.2 Resource Sharing

The notion of sharing is one of the main factors that fueled the growth of the Internet. Most P2P applications permit the sharing of static files such as MP3 audio files, text files and image files. BestPeer supports the sharing of static digital files, *active objects* that facilitate finer granularity of data sharing (and hence access control), as well as computational power. For uniformity, all requests for these resources are performed with agents.

Static Files

In BestPeer, any kind of files in digital format can be traded in its entirety including text files, word documents, images, music files, movie files, executable code (programs,

software), and so on.

Active Objects

However, in many applications, different users may have different access rights to the content of a file. While one may be allowed to see the entire content of the file, another may be denied access to some sensitive information. To support finer granularity of data sharing, BestPeer employs the concept of an *active object*. In active objects, two types of elements are defined: data elements and active elements. A data element describes the content of an object; an active element, on the other hand, contains the name of an *active node* that operates on the object to generate the content. Essentially, an active node is a 'black box' (i.e., an executable code) that receives and sends messages and interacts with the outside through an interface. Depending on the access right of the requester, the active node returns the appropriate content. Using the same illustration, for a person who should be denied sensitive information, the active node will scan the input file, filter away the sensitive information and return the non-sensitive portion to the requester. It is the responsibility of the owner to ensure the correctness of the active object (i.e., that sensitive information should only be accessed by those with the proper access rights).

Computational Power

BestPeer also facilitates the sharing of computational power for requests to local files as follows. The requester sends his/her request for a file together with an algorithm (executable code) that operates on the file. In other words, the requester performs the filtering task at the provider's end. This feature has several advantages. First,

it allows filtering to be performed where the provider's end does not provide the capability (e.g., the owner does not provide an active object). Second, it allows individual requesters to filter the content according to what they desire (e.g., different requesters may be interested in analyzing stock data differently). This is in contrast with the use of active objects where the owner defines what to filter. Third, it facilitates extensibility – new algorithms or programs can be used without affecting other parts of the system. Fourth, existing non-distributed objects can be easily extended for use by a P2P application by leveraging on the support provided by BestPeer. Finally, it optimizes network bandwidth utilization as only the necessary data is transmitted to the requester.

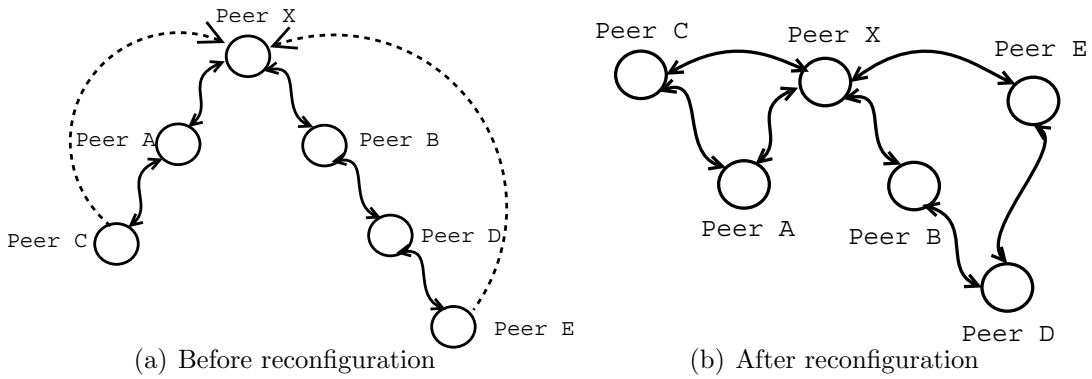
This feature is easily realized by the integration of mobile agents into the P2P framework. Agents that carry code can be dispatched to the data provider.

3.2.3 Reconfigurable BestPeer Network

Existing P2P systems either adopt a static peer network where a node always has the same set of peers or allows users to manually determine the peers of a node (that does not change automatically during runtime).

BestPeer takes a different approach – a node in the BestPeer network can dynamically reconfigure itself by keeping peers that benefit it most (subject to the individual node's definition of 'most benefit'). The rationale is based on a simple assumption: peers that benefit a node most for a query are also likely to provide the greatest gain for subsequent queries. Thus, BestPeer will always try to make a direct connection to the nodes that have the highest priority. In this way, promising peers are traversed before the less promising ones. Every BestPeer node has its own control over the

maximum number of direct peers it can have. Figure 3.3 illustrates an example of BestPeer's reconfigurable feature. In Figure 3.3(a), Peer X is the base node that initiates a request. Here, Peer X initially has two directly connected peers – Peers A and B. However, only Peer C and Peer E contain objects that match Peer X's current query. Peer X can then obtain the results from Peer E and Peer C directly. At the same time, Peer X determines that Peer C and Peer E are not its direct peers and they benefit it most. As such, Peer X will keep these two peers as its directly connected peers (assuming Peer X can keep at least four directly connected peers), resulting in the new network layout shown in Figure 3.3(b).



(a) Before reconfiguration (b) After reconfiguration

Figure 3.3: Example of BestPeer's Reconfigurable Feature

Our approach is to keep promising peers as close as possible with no (or little) information exchange between peers. This is to keep the nodes as autonomous as possible. Moreover, since nodes can redefine the number of direct peers it would like to have and implement their own reconfiguration strategies, any tight form of "collaboration" will be complicated to realize and maintain. In BestPeer, three default reconfiguration strategies have been designed and deployed.

The first strategy, MaxCount, maximizes the number of objects a node can obtain

from its directly connected peers. It works as follows:

- The node sorts the peers based on the number of answers (or bytes) they return.[2] Those that return more answers are ranked higher, and ties are arbitrarily broken. The assumption here is that a peer that returns more answers can potentially satisfy future queries.

- The k peers with the highest values are retained as the k directly connected peers, where k is a system parameter that can be set by a participating node.

We note that this strategy only keeps track of the k directly connected peers, without any knowledge about these peers' direct peers.

The second strategy, MinHops, implicitly exploits collaboration with peers by minimizing the number of hops. It requires that peers piggyback with their answers the value of Hops. This will indicate how far the peers are from the initiator of the request. More importantly, this information provides an indication on what one can access from one's indirect peers. The rationale is as follows: If one can get the answers through one's not-too-distant peers (with small Hops value), then it may not be necessary to keep those nodes (that provide the answer) as one's immediate peers; it is better to keep nodes that are further away so that all answers can be obtained with the minimal number of hops. Thus, this strategy simply orders peers based on the number of hops, and pick those with the larger hops values as the immediate peers. In the event of ties, the one with the larger number of answers is preferred.

The third strategy, TempLoc, is a temporal locality based strategy that favors nodes that have most recently provided answers. It uses the notion of stack distance

[2]We note that many different criteria can be defined. However, their usefulness is domain dependent. We believe a simple strategy like MaxCount should suffice to cover a wide range of applications.

to measure temporal locality. The idea works as follow. Consider a stack that stores all the peers that return results. For each peer that returns answers, move the peer to the top of the stack, and push the existing peers down. The temporal locality of a peer is thus determined by its depth in the stack. The top k peers in the stack are retained as the k directly connected peers, where k is a system parameter that can be set by the node.

```
1.    On KeepBestPeers
2.        K[] = Sort peers according to MaxCount, MinHops or TempLoc policy
3.        // select top-k most beneficial peers based on the policies
4.        P[] = Select all the direct neighbors
5.        // currently connected best peers
6.        FOR EACH P_i in P[]
7.            Evict P_i if it is Not in K[]
8.        LOOP
9.        FOR EACH K_i in K[]
10.           Connect K_i if it is not a direct neighbor
11.       LOOP
```

Figure 3.4: Algorithm KeepBestPeers.

The algorithm of keeping the best peers is shown in Figure 3.4. Selecting appropriate timing to trigger the network configuration is crucial. Obviously, too frequent updates may be prohibited in some situations, while in others they can be a prerequisite. There are two potential solutions. First, periodical update based on a time parameter. For example, reconsidering the candidatures of *best peers* every 10 minutes. Alternatively, the system can employ an event-based mechanism: update *best peers* whenever the statistics indicate that a non-*best peer* is more beneficial than at least one of the current *best peers*.

3.2.4 Location-Independent Global Names Lookup Server

In P2P systems, since nodes can join and leave the network at any time, their IP addresses may be different each time. As such, under DNS, a participating node is effectively treated as a different peer whenever its IP address is different. However, for some applications, recognizing a node (even if the IP address may change each time it is connected to the network) is important. For example, a set of nodes may agree to be peers to collaborate in performing some tasks. As another example, a node may particularly be interested in monitoring the updates of a set of peers. These cannot be realized with DNS alone. To facilitate the identification of a single node that may have different IP addresses on different occasions, each participating node can be assigned a unique BPID, and a centralized server keeps track of the (BPID, IPaddress) pair whenever a node is connected. In this way, one can always be certain of its peers and their "new" IP addresses.

BestPeer adopts such an approach – it introduces a Location-Independent Global Names Lookup Server (LIGLO). LIGLO is a node that has a fixed IP and is running the Location-Independent Global Names Lookup Server software. It provides two main functions: generates a BestPeer Global Identity (BPID) for a peer and maintains the peer's current status, such as the current IP address and whether the peer is currently online or offline (if this information is available). As mentioned, BPID is a unique identifier for a peer. Unlike the centralized approach that is used in systems like ICQ [52], where only one server has control to maintaining the consistency of defined names, there is no limit on the number of LIGLO servers that can run in a BestPeer network. Each LIGLO needs only to maintain its members' name uniquely.

Most of the centralized name servers have to be powerful machines because they

have to handle huge numbers of requests. In contrast, a LIGLO server can limit the number of members that it will handle based on its capability. When the limit is reached, a LIGLO server can reject any new request to assign BPID in order to preserve the efficiency for the existing members. The node has to seek another LIGLO for registration. Once a node is registered with the BPID, it has to inform the LIGLO each time it is connected to the BestPeer network by submitting its current IP to the LIGLO. This information can be used by other nodes that may want to uniquely track a node whose IP address may have changed.

The use of distributed LIGLO services has the following advantages:

1. No single point failure – LIGLO is a distributed name server system; therefore it does not have any single point failure problem. For example, if a peer registered with LIGLO A finds that LIGLO A is down, it can still communicate with other peers that it has. In addition, other peers that are registered with other LIGLO servers will not be affected at all. This is in contrast with the centralized name server approach (such as the ICQ server), where a failure at the centralized server means that all peers will lose their connection.

2. Unlimited name resources – One of the problems with the centralized name server is that all the names must be uniquely defined. For example, if somebody has registered the domain name of "www.mydomainname.com", then that is the one and only one. In LIGLO, on the other hand, a name is unique only with respect to its own server. Two nodes can register to two different servers and be assigned the same name as long as this name has never been previously registered.

3. Scalability – A LIGLO server can be added easily into the network without affecting any of the existing network environments.

4. Support temporary network addresses as the norm – LIGLO defines its own protocol-specific addresses, BPID, to replace the dynamic IPs. This BPID is fixed and can be used in place of dynamic or fixed IPs. Therefore, there is no difference between nodes that have DNS entries and those that do not have. All of them now have the same ability to hosting data and net-facing applications locally.

3.3 A Performance Study

We implemented the BestPeer software with the features discussed in the previous sections. Any node that installs the BestPeer software and registers with a predetermined set of LIGLO servers can participate in the BestPeer network. In this section, we report an extensive performance study conducted to evaluate BestPeer. We compare BestPeer against the Single-Thread Client/Server (CS) Architecture and Multi-Thread CS Architecture in different network layout topologies. The basic difference between CS and P2P is that in a P2P model, the interacting processors can be a client, server or both, while in a CS model, one processsor assumes the role of a service provider while the other assumes the role of a service consumer. Our CS model has some flavor of P2P in that a node can be both a client and a server. However, like the CS model, the server must return its result to the client – as such, the results must be returned along the query path. We also compare BestPeer's protocol against Gnutella's. We study two versions of BestPeer – a static BestPeer where the

reconfiguration feature is turned off, and a a dynamic BestPeer with the reconfiguration feature turned on. This allows us to see the benefits of the reconfiguration scheme. We shall denote these two schemes as BPS and BPR respectively. Before we look at the experiments and findings, we shall propose a evaluation methodology for P2P systems.

3.3.1 Experimental Setup

The experimental environment consisted of 32 PCs, each with an Intel Pentium 200MHz processor and 64M of RAM. Nine of the PC ran on WinNT4.0 operating system, 22 on Window98, and one on Window Millennium. The physical network layout is shown in Figure 3.5. The experiments were conducted when the machines and the network were fully dedicated and the results presented correspond to the average of at least three different executions. The variance across different executions was not significant.

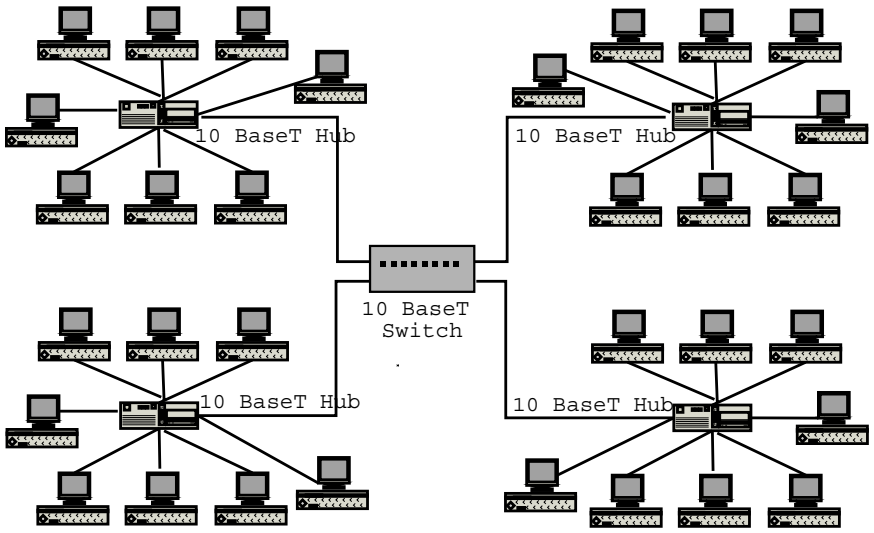


Figure 3.5: Experimental Environment

In the experiment, there was a set of nodes in the network and each of these nodes had a local copy of StorM object [18]. StorM is a 100% Java persistent storage manager. Data to be shared were stored in StorM. For our study, each node stored 1,000 objects in StorM to be shared, and these objects were accessible via StorM's API. For simplicity, we set all objects to be of the same size: 1K bytes. Moreover, there was no replication, i.e., there was only one copy of an object in the BestPeer network used in our study.

We implemented a StorM agent that took as input a query from the user (in the form of a keyword), and then searched through the entire BestPeer network. The goal was to find the occurrences of objects in StorM of each node that matched the query. The whole search process of StorM agent operated as follows:

1. Send a StorM agent. The base node sends an agent to its directly connnected peers.

2. Execute the StorM agent. All the peers that receive the incoming agent prepare a new thread of execution for the agent.

3. Interact with StorM object. The agent makes a comparison for each object stored in the Shared-StorM database with its query. All the matched results are stored in a temporal array.

4. Send the result back. The result is sent back to the base node.

We also incorporated the GZIP data-compression algorithm in the current implementation of BestPeer. All the agents and messages used for communication between every node or peer were in compressed data representation. Compression and

un-compression were performed automatically by the BestPeer platform and were transparent to the software developers.

3.3.2 On Different Network Topology

We first began by evaluating BestPeer on different logical network topology – namely the Star, Line and Tree structures as shown in Figure 3.6. In the Star structure (see Figure 3.6(a)), the central node was the base node that initiated the search query, and all other nodes were directly connected to the base node. In the Tree structure (see Figure 3.6(b)), the root node was the base node that initiated the search request. Each node in the Tree structure, except for the leaf nodes, had k directly connected peers. In the Line structure (see Figure 3.6(c)), all nodes had two peers, except for the end nodes which had only one peer. Here, we used the left most node as the base node that initiated the search query.

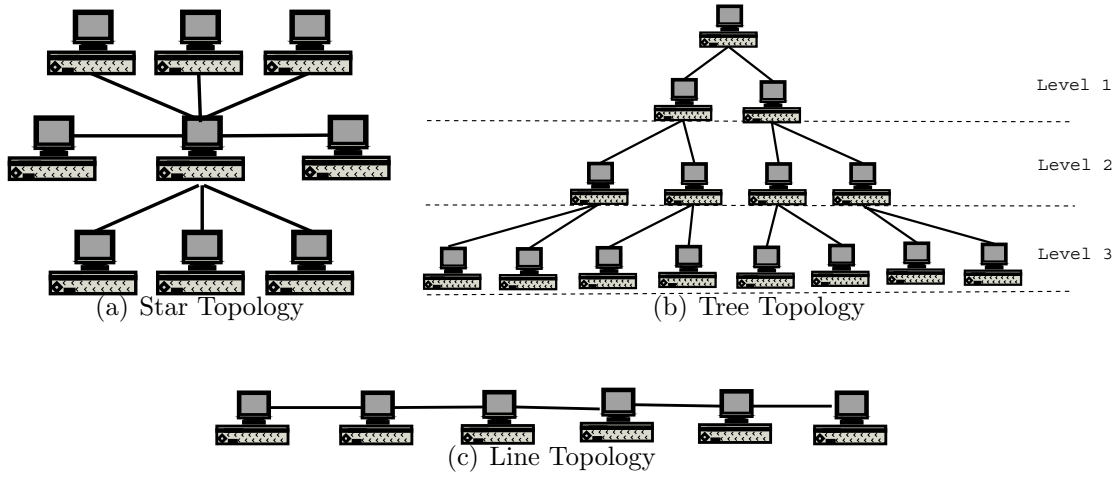


Figure 3.6: Different Network Topologies Used in the Experiment

In this experiment, we varied the number of nodes from one to 32. We ran each scheme several times and used the completion time as the performance metrics. The

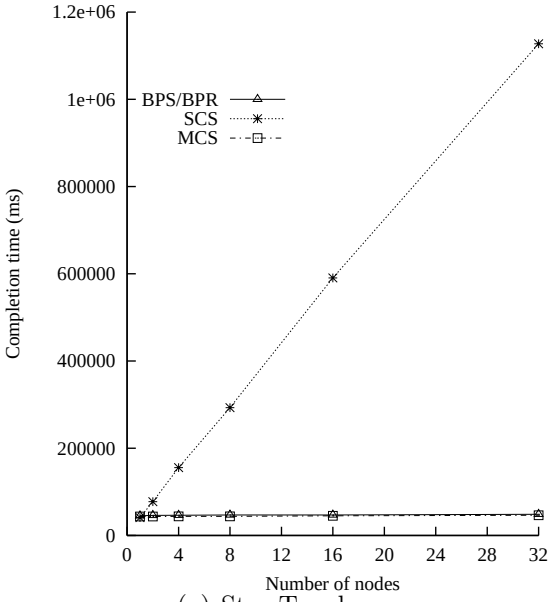
completion time was taken to be the time when all answers from all nodes had been received. Figure 3.7 shows the results.

On Star Topology

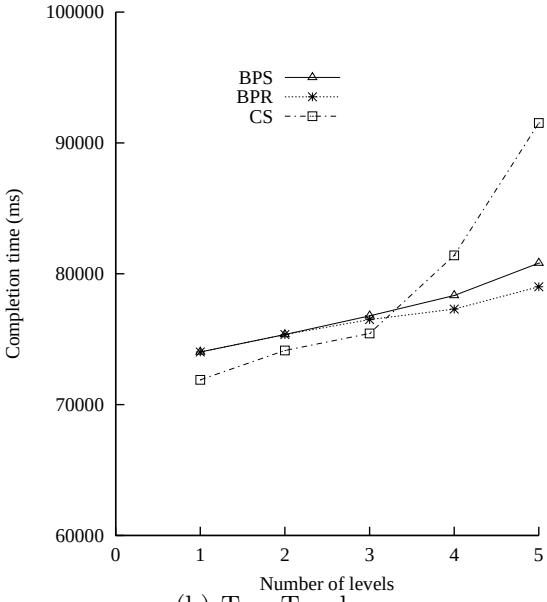
Figure 3.7(a) shows the results for Star topology. First, we note that Static BestPeer (BPS) and Reconfigurable BestPeer (BPR) show similar performance. This is because under the Star topology, there is no difference between the two schemes. As shown in the results, when we increase the size of the network, the Single-Thread CS (denoted SCS) performs worse than the other models. This is because SCS can only handle one connection at any moment – it has to complete the first operation before switching to the second node for another operation. We also note that both the MCS and BP-based schemes outperform SCS significantly. This is so because these schemes exploit parallelism by simultaneously handling multiple connections and transmitting multiple queries to all peers. We also observe that MCS is slightly better than BPS/BPR, but the gain is not significant enough to be visible. We shall explain this further when we look at the results for the Tree and Line topologies. Since SCS performs poorly, we shall not discuss it further. For all subsequent experiments, we shall use MCS only, and for simplicity, we shall denote it as CS.

On Tree Topology

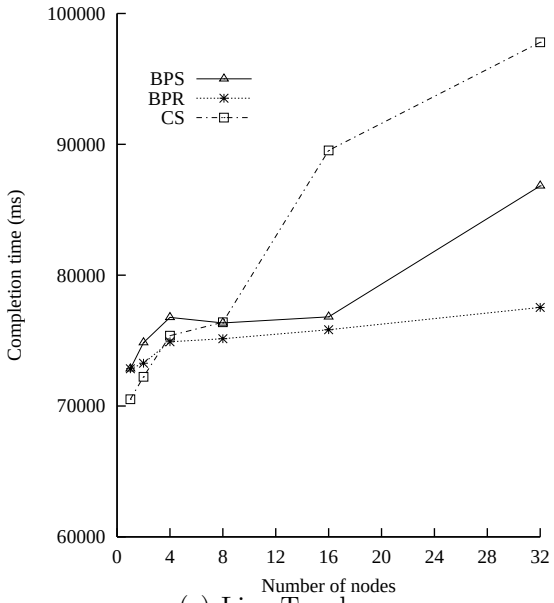
Figure 3.7(b) shows the result on the Tree topology. We note that in this experiment, we used only 48 nodes instead of 63 for level 5. There are several interesting observations. First, we note that CS can outperform BPS and BPR (as noted in the earlier experiment). This is expected as BPS and BPR are essentially code-shipping



(a) Star Topology



(b) Tree Topology



(c) Line Topology

Figure 3.7: On Network Topologies

strategies – not only do they need to transmit the code/agent to the peers, they must also incur the overhead of reconstructing the agent at the peer site.[3] On the other hand, under CS, it simply transmits a query, and the algorithm at the server performs the task there. As a result, when the number of levels is one (which means all peers are directly connected as in the Star network), CS is superior. However, as the number of levels increases, CS begins to degenerate. This is because CS requires the data to be returned along the path by which the request is sent. For BPS and BPR, the answers are returned directly to the query node.

Comparing BPR and BPS, it is clear that BPR outperforms BPS by virtue of the fact that BPR is able to reconfigure itself, resulting in a more optimal network structure. BPS, on the other hand, must always pass through the same set of nodes regardless of their service quality.

On Line Topology

The results on Line topology (see Figure 3.7(c)) show a behavior similar to that of the Star structure. Essentially, the various schemes have the same relative performance for the same results as with the Tree topology, i.e., BPR is the best and BPR outperforms CS for most cases (except when the number of nodes is very small).

3.3.3 Comparison of BestPeer and Gnutella

FURI [2] is a Gnutella protocol compatible Java program that can participate in a Gnutella network. It is a full version program with a GUI interface that can perform

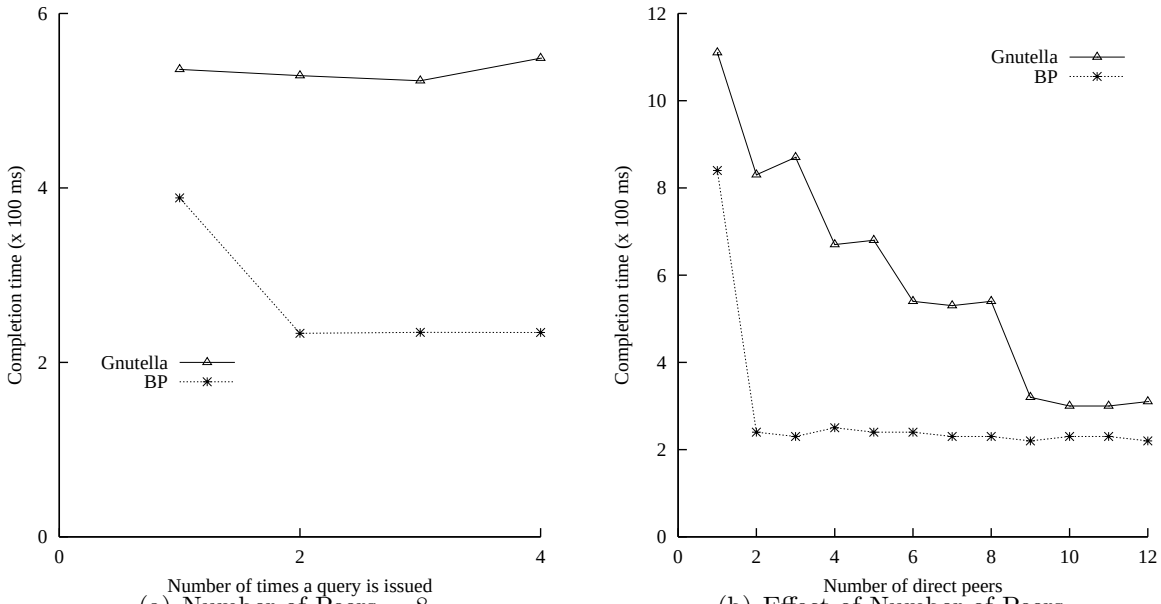
[3]There are two possible implementations for CS. In the first implementation, a server that acts as a client consolidates all answers from its servers before returning the answers to its clients. In the second implementation, a server acting as a client returns any answers that its servers may return through it immediately. We adopt the second implementation in this work.

most of the tasks of a Gnutella servant. In this experiment, we shall compare Gnutella with BPR (denoted BP here). We note that Gnutella essentially adopts an approach similar to BPS, i.e., a node has a fixed set of peers and there is no dynamic adjustment of the set of peers one is directly connected to. In this experiment, each node had 1,000 sharable text files (since the source we obtained from [2] can only evaluate keyword search on text files). We also restricted the answers to those coming from only a few nodes. The completion time was thus determined by the time when all the answers arrived. We repeated a single search query four times during an experiment, and several experiments were conducted to obtain an average result. Figure 3.8 shows the results of the experiments.

Figure 3.8(a), where each node has up to eight directly connected peers, shows the results for each run of a query. We observe that Gnutella is essentially not affected by the number of times the query is run since it employs the same search path each time. On the other hand, we find that for BP, the completion time for the first search is much higher than that for the other searches for the same query. This is because for the first search, BP also needs to route through the entire intermediate peers before reaching nodes with the answers. For subsequent searches, BP's reconfiguration feature ensures that it can directly connect to these nodes with answers. As such, for subsequent searches, the response time is significantly reduced. We also observe that BP outperforms Gnutella in all runs. This is because, in this experiment, we do not return the data files as output as Gnutella will not return results directly; it simply sends the list of files that matches the query. Therefore, while BP and Gnutella return results out-of-network, this feature is not used in the experiment. In addition, Gnutella requires messages to be sent to the peers along the path of

the query traversal, i.e., the list of files have to be transmitted through the query traversal path. On the other hand, under BP, nodes with matching files will send the information directly back to the initiating node.

From Figure 3.8(b), we see the effect of the number of peers over four queries each time. As the number of directly connected peers increases, BP remains superior. While Gnutella's performance also improves with more peers, traversing the same path each time and returning answers along the query path lead to its poorer performance.



(a) Number of Peers = 8 (b) Effect of Number of Peers

Figure 3.8: BestPeer vs Gnutella

3.4 Summary

In this chapter, we have presented a P2P system called BestPeer that can be used to support a wide range of applications. BestPeer has several nice features. First,

because it integrates agent and P2P technologies, it provides easy extensibility to existing systems. Second, it provides a mechanism to reconfigure a node's peers based on some optimization criteria. Third, it supports distributed LIGLO servers to maintain crucial information of BestPeer participants. Our extensive experimental studies show that BestPeer is a promising system for distributed processing.

Chapter 4

PeerDB: A P2P-based System for Distributed Data Sharing

In this chapter, we will extend the BestPeer framework that we have developed in Chapter 3 for application to database functionalities.

Given the explosive growth of data available to us, the abilities to manage this vast amount of data and provide fast and relevant answers to the questions have assumed paramount importance. Managing and utilizing huge collections of data requires a DBMS. However, the existing systems proposed in the literature are not designed to support complex data processing. For example, many domain-specific P2P systems that have already been deployed [85], such as Freenet [39], Gnutella [42] and Napster [75], rely on simple keyword-based methods to extract data. Hence, these systems support only coarse granularity sharing (sharing of the entirety of a file) and lack object/data management capabilities and support for content-based search.

Based on this observation, we present PeerDB, a P2P-based system for distributed data sharing. PeerDB has several distinguishing features. First, each participating node is a full-fledged object management system that supports content-based search. Second, in PeerDB, users can share data without a shared global schema. Third,

PeerDB adopts mobile agents to assist in query processing. Since agents can perform operations at the peers' sites, the network bandwidth is better utilized. More importantly, agents can be coded to perform a wide variety of tasks, making it easy to extend the capabilities of a PeerDB node. For example, an agent may further manipulate the data retrieved from a node, filtering away uninterested objects or sending back only summarized data. Finally, PeerDB supports mechanisms to dynamically keep promising (or best) peers in close proximity based on some criteria. For example, peers that are most frequently accessed are directly communicable while nodes that are less frequently accessed can be reached through peers. This significantly reduces the response time to queries.

We implemented PeerDB, a prototype P2P distributed object management system that incorporates all the above features. To evaluate PeerDB, we propose a systematic methodology for evaluating P2P systems. Our methodology considers both efficiency and effectiveness (quality of answers) of P2P systems. We conducted our experiments on a cluster of 32 Pentium II PCs. Our experimental results show the effectiveness of PeerDB for distributed data sharing.

4.1 P2P Distributed Data Management: What Is It?

As noted in the introduction, practically all existing P2P systems are designed to support data sharing at a coarse granularity (e.g., files, documents). In this section, we first distinguish between P2P systems and distributed database systems. We then "define" P2P distributed data management by looking at three examples (due to

space constraints) of how P2P technology can be employed for distributed database applications. These examples will also show the need for database technology in P2P systems.

4.1.1 P2P vs Distributed Database Systems

There are several notable features that distinguish P2P systems from distributed database systems (DDBS)[78]:

1. In P2P systems, nodes can join and leave the network anytime. In DDBS, nodes are added to and removed from the network in a controlled manner, i.e., when there is a need for growth or retirement.

2. In P2P systems, there are usually no predetermined (global) schemas among nodes. Queries are largely based on keywords. There are several reasons for this. First, most of the current applications do not require a fixed schema. (Napster is one exception where data is shared with a fixed schema – the one that describes music files.) Second, as nodes can join and leave the network at anytime, a fixed schema does not reflect the actual information that may be available at a single time. In DDBS, nodes are typically stable and have some knowledge of a shared schema.

3. In P2P systems, nodes may not contain the complete data. Further, nodes may not be connected. Thus, answers to queries are typically incomplete. By "completeness", we mean all answers that satisfy a query. In DDBS, one expects and can actually retrieve complete sets of answers.

4. In P2P systems, content location is typically by "word-of-mouth", i.e., a node routes its query to its neighboring nodes, and so on. In DDBS, the exact location to direct the query is typically known.

Based on the above points, we do not consider data integration systems to be P2P distributed data management systems (even if each node has the capabilities to act as middleware and server). Instead of formalizing the concept of P2P distributed data management systems, we show with sample applications what such systems may be like.

4.1.2 Health Care

In a hospital, each specialist has a group of patients who are solely under his care. While some patient data is stored in a centralized server of the hospital (e.g., name, address, etc), other data (e.g., X-rays, prescription, allergy to drugs, history, reaction to drugs, etc) is typically managed by the specialist on his personal PC. In most cases, the specialist is willing to share the patient data that he maintains on his PC, but there are always some cases where he is unwilling to share the data for different reasons (e.g., part of his research program on a new drug, etc). Meanwhile, by making the sharable patient data available to others, the specialist can compare notes with his colleagues on different patients suffering from similar symptoms, hence helping all specialists in the hospital to make better decisions on their treatment (e.g., drugs to prescribe, reactions to look out for, etc).

Here, we can deploy a P2P distributed management system: (1) any specialist can join/leave the network; (2) the answers need not be complete (i.e., missing data from some specialists is not critical), (3) nodes have to search for content as in P2P

systems, (4) the schema defined by each specialist may be different from those defined by the others, (5) there is a need for data management, and (6) each specialist has something to share and is also interested in the data of the others.

4.1.3 Genomic Data

P2P can also be applied in bioinformatics applications such as sequence analysis, gene finding, structural prediction, molecular mining or biological reasoning in genomic data. The discovery of new proteins necessitates complex analysis in order to determine their functions and classifications. The main technique that scientists use in determining this information has two phases. The first phase involves searching known protein databases for proteins that match the unknown protein. The second phase involves analyzing the functions and classifications of similar proteins in an attempt to infer commonalities with the new protein. While there are several known servers on genomic data (e.g., GenBank, SWISS-PROT and EMBL), there is much more data that is produced each day in many laboratories all over the world. These scientists create their own local databases of their newly discovered proteins and results, and are willing to share their findings with the world. Clearly, this is an application for P2P distributed data management systems for the same reasons as heath care applications.

4.1.4 Data Caching

In the above two examples of P2P distributed data management systems, each participant is actively involved in the process of consuming and supplying data. P2P distributed data management can also be deployed in passive nodes: nodes that are

used to allocate resources (storage or computational power) to data that they may or may not be interested in. Caching results from earlier queries is one such example – a node may have issued a query to some server (e.g., a data warehouse), the query's results can be cached on the node (or some other neighboring nodes), another node that requests for data that overlaps with the existing query results can potentially obtain partial answers quickly from this node, and the remainder from the original server. This also lightens the load on the original server, and moves the data to or closer to edge devices.

4.2 Peering Up for Distributed Data Sharing

In this section, we will present PeerDB, a prototype P2P-based system for distributed data sharing. PeerDB's P2P-enabling technologies are provided by BestPeer [1, 77]. However, it extends BestPeer in the following ways. First, data in each node is managed by a database system. In other words, PeerDB is a network of database-enabled nodes. Second, data can be shared without a global schema. Third, query processing is assisted by mobile agents. Fourth, each node can reconfigure itself based on some optimization criteria from the answers returned. We shall discuss these features here.

4.2.1 Architecture of a PeerDB Node

Figure 4.1 illustrates the internal structure of a PeerDB node. There are essentially four components that are loosely integrated. The first component is a data management system that facilitates the storage, manipulation and retrieval of the data at

the node. We have used MySQL [3], which is a popular Open Source Database, as our storage server. Thus, the system can be used on its own as a standalone DBMS outside of PeerDB. We note that the interface of the data management system is essentially an SQL query facility. For each relation that is created, the associated meta-data (schema, keywords, etc.) is stored in a **Local Dictionary**. There is also an **Export Dictionary** that reflects the meta-data of objects that are sharable with other nodes. Thus, only objects that are marked for export can be accessed by other nodes in the network. We note that the meta-data associated with the Export Dictionary is a subset of those found in the Local Dictionary, and the distinction here is a logical one (as the actual implementation minimizes redundancy). We shall defer the discussion on how the Export Dictionary will be used, and the details on the meta-data to when we address the query processing strategy.

The second component is a database agent system called DBAgent. DBAgent provides the environment for mobile agents to operate in. Each PeerDB node has a *master agent* that manages the query of the user. In particular, it will clone and dispatch *worker agents* to neighboring nodes, receive answers and present them to the user. It also monitors the statistics and manages the network reconfiguration policies.

The third component is a cache manager. We shall defer the discussion of the cache manager to a later subsection. Here, it suffices for us to know that we are dealing with caching remote data in secondary storage, and the cache manager determines the caching/replacement policy.

The last component is the user interface. This provides a user-friendly environment for users to submit their queries, maintain their sharable objects, and insert/delete objects. In particular, users search for data using SQL-like queries.

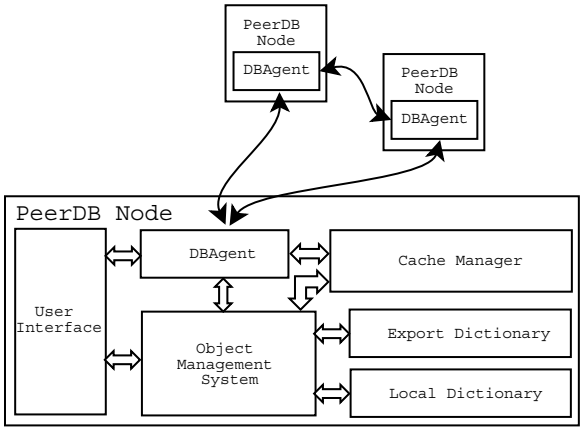


Figure 4.1: PeerDB Node Architecture

4.2.2 Sharing Data without Shared Schema

One of the main objectives of PeerDB is to allow users to manage their (private and sharable) data using a database management system (DBMS). However, as noted, there are no predetermined and uniform schemas that nodes share. Unless users interact with one another somehow, we can expect data to be defined differently by different users, even if they may have interests in data from a common domain. For example, in naming a relation, a genome scientist may label his set of protein database by protein name (e.g., kinases, annexin) while another may name it by species (e.g., mouse, human, zebrafish). Similarly, at the attribute level, one scientist may call the length of sequences `length` while another may use the term `len`. To complicate matters further, a scientist may create a single "universal" schema while another may "normalize" his database to multiple tables. Thus, it is difficult to locate data if the traditional method of exact matching of relation names/attributes is used.

To address this issue, we adopt an approach based on Information Retrieval (IR) [16]. For each relation that is created by the user, meta-data is maintained for each

relation name and attribute. These are essentially keywords provided by the users on the creation of the table, and serve as a kind of synonymous names. (One can think of this as a miniature thesaurus.) Continuing with our example, for a table of Kinases proteins, while the relation name may be `Kinases`, the keyword `protein` will be useful during search. Similarly, two users defining the length of a sequence variously as `len` and `length` are likely to have the common keyword `length`. In this way, potentially relevant data can be determined using the following *relation-matching* strategy:

- Consider a query (R, A, C) where R is the set of relations, A is the set of target attributes, and C is the set of conditions. (This corresponds to a simple SPJ query in SQL.) Let V denote all attributes that appear in A and C.

- R is searched against keywords for relation names, and V is searched against keywords for attribute names. Note that this search involves looking for matching keywords of R against keywords for other relations; the same holds for attributes. The result of this search process will be a list of relations whose relation name keywords match R (or their keywords) and/or attribute keywords match attribute names in V (or their keywords).

- Given a query Q of the form (R, A, C), and a relation D with attributes T, the degree in which D matches Q can be computed as follows:

$$Match(Q, D) = \frac{(wt_r * r) + \left(wt_a * \underset{match}{N}(A \cup C, T) \right)}{wt_r + (wt_a * N(A \cup C))}$$

  where wt_r and wt_a are weights assigned to reflect the importance of matching relation and attribute names respectively. r takes the value of 1 or 0; r is 1 if and only if D and R share some common matching keywords. Otherwise, r is

0. $\underset{match}{N}(A \cup C, T)$ refers to the total number of matching keywords between the attributes involved in Q and those of D. $N(A \cup C)$ indicates the total number of distinct keywords for the attributes in Q. The set of relations that potentially contain the answers to Q are those that have scores above a certain threshold value.

With the above strategy to locate matching relations, we note that we can share data without the explicit sharing of schema. This flexibility is also an important distinction between PeerDB and existing distributed DBMS. Note that the relations and meta-data will be returned to the user first, who will then decide on the data that is of interest (see the next section on query processing strategies).

We illustrate the strategy with an example. Suppose we have four peers that share genomic data. Peer P1 defines a relation Kinases(SeqID, length, proteinSeq). Peer P2 defines a relation Protein(SeqNo, len, sequence). Peer P3 defines two relations ProteinKLen(ID, seqLength) and ProteinKSeq(ID, sequence). Peer P4 defines a relation Protein(name, char). Figure 4.2 shows the keywords defined for these relations by the various peers. Suppose the user at peer P1 (who knows his own schema but not the schemas of other peers) issues the following query to look for kinases sequences that are longer than 30 base pairs:

```
SELECT SeqId, proteinSeq
FROM Kinases
WHERE length > 30;
```

Now, since one of the keyword for Kinases (relation name) is protein, and protein is also a keyword for P2's relation `Protein` and P3's relations `ProteinKLen` and

`ProteinKSeq`, these relations match the query relation. Similarly, we find that the attributes `SeqID`, `proteinSeq` and `length` all have matching keywords in P2 and P3. For P3, we note that the query may have to be turned into a join query when it is evaluated there. For P4, we only have a match in relation name but not in the attributes. Thus, P4 will be ranked lower than P2 and P3. Semantically, we note that P2's data is not actually those that P1 is interested in (since it is not Kinases data). As such, it is important to have the meta-data and additional information returned to the users before fetching the data.

Peer	Names	Keywords
P1	Kinases	protein, human
	SeqID	key, identifier, ID
	length	length
	proteinSeq	sequence, protein sequence
P2	Protein	protein, annexin, zebrafish
	SeqNo	number, identifier
	len	length
	sequence	sequence
P3	ProteinKLen	protein, kinases, length
	ID	number, identifier
	seqLength	length
	ProteinKSeq	protein, sequence
	ID	number, identifier
	sequence	sequence
P4	Protein	protein, kinases, annexin, ...
	name	name
	char	characteristics, features, functions

Figure 4.2: Keywords for Relation/Attribute Names

4.2.3 Agent Assisted Query Processing

In PeerDB, we adopt a two-phase query processing strategy. In the first phase, the relation matching strategy is applied to locate potential relations. These relations are then returned to the query node for two purposes. First, it allows the user to select the more relevant relations. This is to minimize information overload when the data may be syntactically the same (having the same keywords) but semantically different. Moreover, this can minimize transmitting data that is not useful to the user, and hence better utilize the network bandwidth. Second, it allows the node to update its statistics to facilitate future search processes. The second phase begins after the user has selected the desired relations. In this phase, the queries will be directed to the nodes containing the selected relations, and the answers are returned.

PeerDB's query processing is completely assisted by agents. In fact, it is the agents that are sent out to the peers, and it is the agents that interact with the DBMS. Moreover, a query may be rewritten into another form by the DBAgent (e.g., a query on a single relation may be rewritten into a join query involving multiple relations). To elaborate on the query processing strategy, we shall distinguish two types of queries: the *local* query and the *remote* query. A query is local to a node if it is initiated there, and remote otherwise.

Processing Local Queries

When a user issues a query (SQL-like selection query), a master agent will be created to oversee the evaluation of the query. The following operations are performed by the agent:

Phase I

- The agent "parses" the query to extract the list of relation and attribute names.
- The relation matching strategy is applied on the local dictionary. Promising relations can then be returned to the user immediately.
- At the same time, the master agent will clone *relation matching agents* and dispatch them to all neighbors of the node. Besides the query, the agent also carries with it two other information: (a) IP address of the node that initiates the query; (b) TTL (Time to live). The former is needed to allow remote nodes to return answers directly to the query node. The latter indicates the lifetime of an agent. This allows the process of cloning and forwarding to keep on going until the agent lifetime expires.
- The master agent will wait for the answers (relations schema) from remote nodes. On receiving any answers, they will be returned to the user for selection.
- For peers that return multiple relations, the master agent returns the individual relations (if their scores on the number of matching keywords exceed the threshold) as well as the combinations of relations that are related (e.g., have a key-foreign key relationship). Referring to our earlier example, P3 will produce three answers: `proteinKLen`, `proteinKSeq`, and `proteinKLen` $\bowtie$ `proteinKSeq`.

Phase II

- For each relation selected by the user, the master agent will clone a *data retrieval agent* for that relation. One of the first tasks of the agent is to reformulate the query so that it matches the relation name and attributes at the target node.

Clearly, it is possible that some attributes may be dropped because the target relation has no such matching attributes. For a combination of relations, the data retrieval agent will also rewrite the orginal query into a join query involving the combination of relations.

- If the target relations are found locally, the worker agent will submit a reformulated SQL query to the DBMS to retrieve the data. The data is then returned to the agent, formulated for output and returned to the user.

- If the target relations are on a remote node, then the worker agent will be dispatched with the query node's IP address. Answers will be returned directly from the remote host to the master agent who will then formulate and return the answers to the user.

We note that the two phases are only logical. In fact, as soon as relations are returned, they are shown to the user, and the user can start selecting relations; and as soon as a relation (or combination of relations) is selected, the agent is sent out to retrieve the data. In this way, answers are returned progressively (without a long waiting time). Moreover, users could be viewing answers (data) while other agents are still searching the PeerDB network for candidate relations.

Processing Remote Queries

As mentioned, for a remote query, it is essentially an agent that arrives at the node.

Phase I: Relation Matching Agent

- If the agent has not visited the node previously, the TTL value is reduced by

one.

- The agent will search the **export dictionary**. Promising relations are then returned to the query node at the IP address provided by the agent.
- If TTL > 0, the agent will clone more relation matching agents and dispatch them to the neighbors of the current node; otherwise, the agent will be dropped.

Phase II: Data Retrieval Agent

- The agent will formulate an SQL query and submit it to the DBMS.
- Once the answers are retrieved, they are returned to the query node directly. If the retrieved data needs to be further processed before being returned, then the agent will perform the task (with the code that it carries along) and return the summarized data.
- The agent may then be dropped.

4.2.4 Monitoring Statistics

One of the tasks of the master agent is to perform the reconfiguration of the network based on a reconfiguration policy selected by the user. The master agent monitors two types of statistics. The first is the relation information obtained from the first phase of the query processing strategy. In particular, the keywords of selected relations may be "exchanged" to update the meta-data. The second is the number of answer objects obtained from the selected relations. This can be used to determine the nodes to be connected directly.

PeerDB also extends BestPeer with a temporal locality based reconfiguration policy that favors nodes that have most recently provided answers. It uses the notion of stack distance to measure the temporal locality. The idea works as follows. Consider a stack that stores all the peers that return results. For each peer that returns answers, move the peer to the top of the stack, and push the existing peers down. The temporal locality of a peer is thus determined by its depth in the stack. The top k peers in the stack are retained as the k directly connected peers, where k is a system parameter that can be set by the node.

4.2.5 Cache Management

PeerDB supports the caching of answers returned from remote nodes in order to reduce the response time for subsequent answers. For every relation that the user retrieves (in Phase II of the query processing strategy), the answers are cached. Caching raises many complicated issues. We look at three of them here. First, the cached copy may be outdated. To handle this, PeerDB only keeps the answers for a fixed period of time, after which the cache is invalidated. Second, since storage space is limited, we adopt the least recently used (LRU) replacement policy: whenever we run out of disk space, we replace the cache that is least recently used. Finally, in a P2P environment, many PeerDB nodes may be caching the same data. As such, a search may give rise to multiple "copies" of the same data. While this is a semantic issue that is to be left to the user, we attempt to minimize the effort as follows. For each cached relation, we also maintain the information on the BPID of the source node (recall that each node has a unique identifier BPID provided by the BestPeer technology). When a node is not the source of a relation, its response to a search will

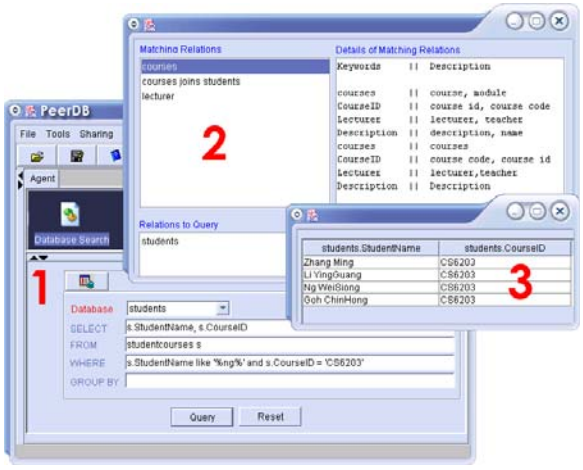


Figure 4.3: PeerDB Interface.

also include the BPID of the source node. All relations, except one, with the same keywords from the same source node will be pruned away during Phase I of query processing.

4.3 A Performance Study

We implemented the PeerDB software with the features discussed in the previous sections. Any node that installs the PeerDB software and registers with a predetermined set of LIGLO servers can participate in the PeerDB network. Figure 4.3 shows the PeerDB interface – Window 1 shows the query interface, Window 2 displays the results of matching schemas, and Window 3 displays the answer tuples from a selected relation.

In this section, we report an extensive performance study conducted to evaluate PeerDB in two aspects: its relation matching strategy and its performance.

4.3.1 On Relation Matching Strategy

In this section, we present the experimental results of PeerDB in the search for matching relations in the P2P environment without a global schema.

We generated a large number of relations as follows. First, we created a set of semantically equivalent categories, C. In each category, we had c keywords which were assumed to represent the same semantic meaning, i.e., any two keywords referred to the same meaning. Next, we created a set of relations, and each relation was assigned two to five keywords (since users are not expected to enter too many keywords) selected randomly from an arbitrary category picked from C. Each relation had a number of attributes, and each attribute was also assigned two to five keywords picked randomly from an assigned category from C.

The following query form was used in our experiment:

```
SELECT attribute_X
FROM relation_i
WHERE attribute_Y = value_1
        and attribute_Z > value_2;
```

We used standard precision and recall measures as the performance metrics. Precision measures the purity of search results, or how well a search avoids returning results that are not relevant; recall refers to the completeness of the retrieval of relevant items. We consider a relation to be relevant to the query if more than k keywords from the relation names match. In our study, we set k to 2. This set formed the basis for the computation of precision and recall.

For each relation that was examined, we computed its matching score. We varied the threshold value from 0.1 to 0.9. For each result returned, we computed its precision and recall. The results are shown in Table 4.1.

Threshold	Precision	Recall
0.1	0.33	0.85
0.3	0.36	0.78
0.5	0.50	0.57
0.7	1.00	0.28
0.9	1.00	0.21

Table 4.1: Precision and Recall for Varying Threshold Values (Synthetic Data)

In order to further verify the findings from the previous experiments, we used the real data Gene/Protein dataset extracted from KDD CUP 2001 [57] for this experiment. First, we partitioned the data into various relations, some of which had the same semantic meaning, and some did not. Then we distributed the relations to the peers, so that the relations with the same semantic meaning would go to various peers. Lastly, we populated the tables, and added meta-data to them.

We issued the query from one peer, and collected retrieved relations from other peers. Since we could tell whether the retrieved relations were relevant, we could compute the precision and recall to measure the performance. We varied the threshold value from 0.1 to 0.9. The results are shown in Table 4.2

As shown in Table 4.1 and Table 4.2, when the threshold value was large, the precision was high as most of the relevant relations could be identified. In fact, in this experiment, we had 100% precision when the threshold was 0.7 and above. However, the recall was low because of the large number of irrelevant relations that shared some common keywords. These results are consistent with typical IR search

Threshold	Precision	Recall
0.1	0.42	1.00
0.3	0.50	1.00
0.5	0.60	1.00
0.7	1.00	0.66
0.9	1.00	0.30

Table 4.2: Precision and Recall for Varying Threshold Values (Real Data)

results, showing that the proposed strategy is effective.

4.3.2 On PeerDB Performance

To evaluate PeerDB's performance, we conducted different sets of experiments. We first compared PeerDB against the Client/Server (CS) Architecture. The basic difference between the two models is that in a P2P model, the interacting processors can be a client, server or both, while in a CS model, one processor assumes the role of a service provider while the other assumes the role of a service consumer. Our CS model had some flavor of P2P in that a node could be both a client and a server. However, like the standard CS model, the server must return its result to the client – as such the results must be returned along the query path. We studied two versions of PeerDB – a static PeerDB where the reconfigurable feature was turned off, and a dynamic PeerDB with the reconfiguration feature turned on. We compared both schemes with the CS architecture. This allowed us to see the benefits of the reconfiguration scheme. We shall denote these two schemes as PDMS and PDMR respectively. We also compared PeerDB with the pure message-passing based protocol and the agent-based protocol. The objective in this experiment was to show the cost and effect of using agents in PeerDB. Before we look at the experiments and findings,

we shall propose an evaluation methodology for P2P-based systems.

Evaluation Methodology

Any system has to be evaluated for its efficiency and effectiveness. The former deals with the performance issue, while the latter deals with the quality of the answers. Based on these two issues, the criteria for the evaluation of a system may be drawn up. Unlike existing distributed systems, there are no clear criteria on how P2P systems should be evaluated. Like an Internet search engine, the answers to queries depend on the peers that are searched, which may not include every peer in the P2P network. In addition, every query may involve different peers (since peers change over time).

For the purpose of evaluation, a controlled environment is necessary. We propose that the following three scenarios be evaluated. First, different schemes should be evaluated based on a fixed set of nodes. This can be useful for a set of nodes that exploit P2P technology to facilitate collaboration, i.e., it is essentially a traditional distributed environment where all nodes participate in answering a query. Here, we can study how different P2P protocols or reconfiguration strategies perform.

Second, in a P2P network, the rate at which answers are returned is important. This is because users have no idea as to which peers will be providing the answers to their queries, and how many peers will be searched. A long initial waiting time is not likely to be acceptable to users.

Third, the quality and quantity of the answers returned are important measures too. A node may return answers quickly, but it may return only very few answers or answers that are not really relevant to the query. While quality is based on the semantics of the query, the quantity of answers is easy to obtain and use as a performance

metric.

Experimental Setup

The experimental environment consisted of 32 PCs, each with an Intel Pentium 200MHz processor and 64M of RAM, all running on the WinNT4.0 operating system. The physical network layout is shown in Figure 3.5.

There were a total of 10,000 objects, each of which was 10 KB. The data was randomly assigned to nodes, such that each node held 1,000 objects.

In practice, we expect users to be interested only in part of the entire dataset. There will always be some data that is of no interest to them, and will never be accessed by them. For example, in the case of Napster, while there is always classical music being shared, a user who prefers contemporary music may just dislike totally the classical music that is being shared. In our experiments, we tried to model this by dividing the queries of each user as follows: (a) x% of queries were directed at 'hot' data in the entire dataset. This hot data was also frequently accessed by other users. (b) y% of queries were directed at z% of the cold data. This modeled the case that individual users may have their own taste on cold data. (c) The remaining queries were directed at the remaining cold data. As default, we set x as 80%, y as 15%, and z as 20%. Moreover, 20% of the data was hot, and 80% of it was cold.

The experiments were conducted when the machines and the network were fully dedicated. Moreover, each node was "warmed up" to fill out its local storage before we started to collect results on the experiments. The results presented correspond to the average of at least three different executions. The variance across different executions was not significant.

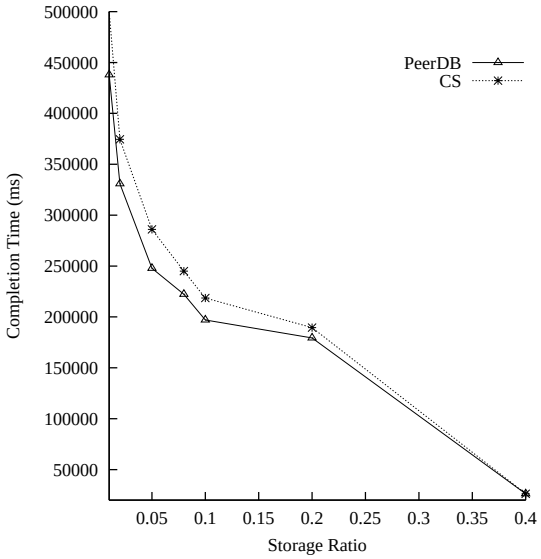


Figure 4.4: Effect of Storage Capacity

Effect of Storage Capacity on Caching

In the first set of experiments, we compared PeerDB with the CS architecture by varying the storage capacity of each peer. We defined the storage ratio as the size of the storage size of a node to the size of the objects stored at the central server. Figure 4.4 shows the effect of storage ratio on response time. First, we observe that as the storage ratio increases, the response time of both methods decreases. This is expected as more objects could be found in the local and neighboring peers. This also clearly illustrates the benefits of sharing storage resources. Second, we note that PeerDB outperforms the CS model. This is expected as the CS model requires the answers to be returned via the search path.

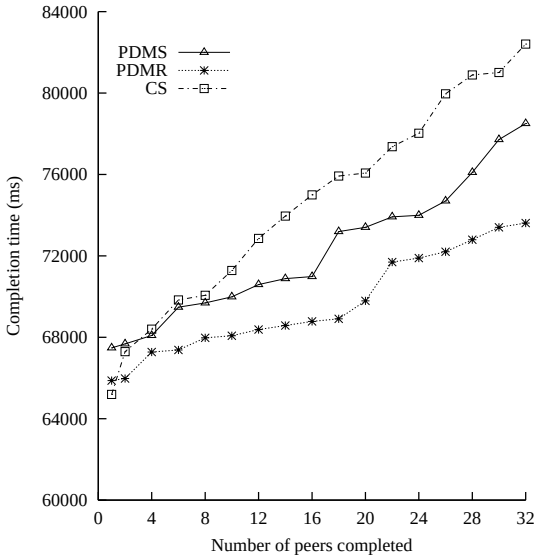
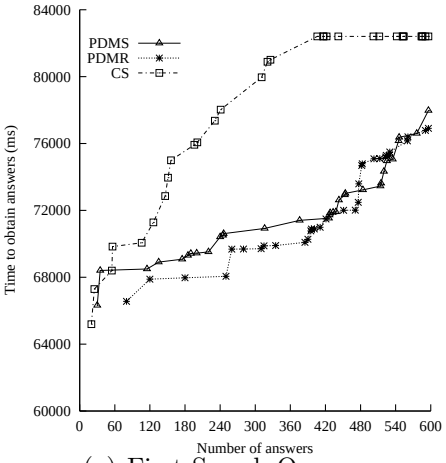


Figure 4.5: Rate of Returning Answers

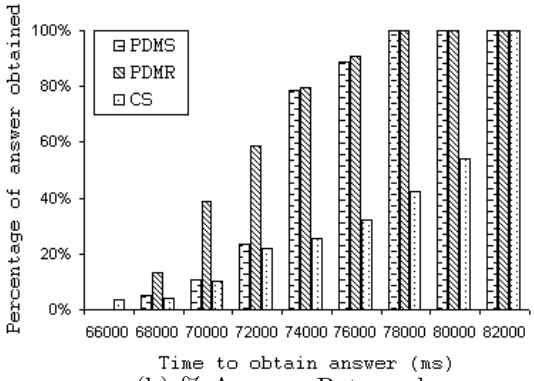
PeerDB vs CS

In this experiment, we first evaluated the performance of PDMS, PDMR and CS on the rate at which answers were returned. The number of nodes was fixed at 32. A search query was issued four times, and the average time at which nodes responded was noted. Figure 4.5 shows the results of the experiment. In the figure, the point (K, T) indicates that K nodes have responded after T time units. We note that it is possible that under different schemes, different nodes respond at different time and with different answers. We shall defer this discussion to the next experiment.

As shown in the figure, PDMR is still the best scheme, outperforming PDMS by virtue of its ability to reconfigure the network. It is able to reach out to more promising nodes directly – after each query, PDMR reconfigures itself so that the next query can be directed to the more promising nodes first. We note that, except for the first few nodes, CS returns answers much slower than PDMR/PDMS – as it



(a) First Search Query



(b) % Answers Returned

Figure 4.6: Number of Answers Returned

can only return answers along the path that the query has been transmitted.

Having a fast initial response time is not good enough. It is possible that nodes that return answers first provide very few answers. For the earlier experiments that studied the initial response time, we also kept track of the number of answers that were provided by each node. Figure 4.6(a) shows a plot of the results. As shown, it is clear that CS returns the first few answers much faster than PDMS and PDMR. This is expected since the first few directly connected nodes that receives the query

can return their answers immediately. For PDMS/PDMR, the overhead of the code-shipping strategy results in a longer initial response time performance. However, as more answers are returned, PDMS/PDMR proves to be superior to CS, demonstrating the superiority of P2P technologies over traditional CS models. We also note that PDMR performs generally better than PDMS. From Figure 4.6(b), we can further confirm the effectiveness of PDMS/PDMR over CS. By the time PDMS and PDMR have received all the answers (100%), CS has only returned about 40% of the answers. As observed earlier, PDMR can generate more answers quickly by virtue of its ability to keep more relevant peers "closer".

Benefits of Agent-based Querying

In this experiment, we studied how much could be gained by using an agent-assisted query processing strategy. Here, we assumed that the query required some functions that were not supported by the DBMS. As such, the operation could not be pushed down to the DBMS. Instead, the data had to be first retrieved, and the operation performed on the data, before the answers to the query could be obtained.

In this experiment, we showed the cost and effect of using the pure message passing protocol and the agent-based protocol in the P2P environment. Here, we assumed the query required only one remote access. The whole process was divided into three phases: sending message (message-passing protocol) or agent (agent-based protocol) to remote host, remote host processed the request, and remote host returned the result to the originator. The answer size was set as 0.1% of the whole dataset. The difference between the message-based and the agent-based protocol is that the message-based protocol is a data-shipping strategy, i.e., remote data is transferred to

the query node to be processed there. On the other hand, an agent-based protocol is a code-shipping strategy that carries the processing code to the remote site and performs remote execution. Only answers produced by the agent will be returned. The total response time includes the cost of data transfer, i.e., message, code and data, and processing time. In Figure 4.7, we observe that the completion time of the message-based protocol increases exponentially when the data size increases. The overhead of the data-shipping results in a longer response time performance. As a result, when the number of data to be transferred across the network increases, the mobile agent-based protocol is superior.

Most network applications (client/server-based or P2P-based) require more than one communication with another node to complete each transaction. Therefore, in this experiment, we looked into the messages overhead in the pure message-based protocol vs. the agent-based protocol. In this experiment, multiple remote executions were required to answer a query. Essentially, the query requested for multiple objects from a remote site; however, the query only knew the object to retrieve after the earlier requests were completed. Thus, we had a chain of queries and computations. In our test, each of the object requested would cause 5MB of data transfer across the network.

The results on multiple-communication transaction is shown in Figure 4.8. Clearly, the agent-based approach is superior. Under the message-passing protocol, the query is transmitted, and the data is returned to the node to be processed on the node. This has to be done before subsequent operations can be issued. On the other hand, in the mobile agent approach, all operations can be performed at the remote node once the code is transmitted. Once the agent is constructed at the remote site, it can

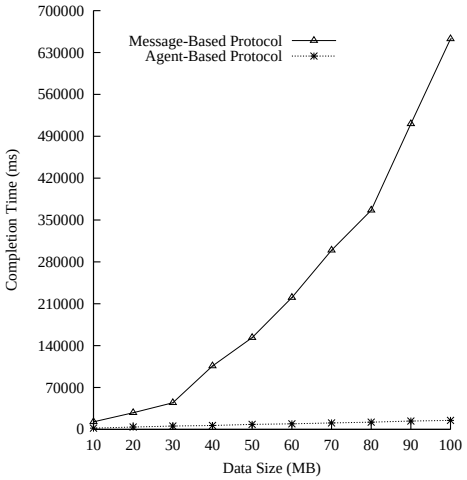


Figure 4.7: Completion Time vs. Data Size

interact with the remote node directly until the final result is obtained. Therefore, it optimizes network resources and bandwidth.

4.4 Summary

In this chapter, we have presented a P2P-based distributed data sharing system called PeerDB. PeerDB has several nice features. First, it employs a data management system. Moreover, it facilitates data sharing without any predetermined shared schemas. Second, because its query processing capabilities are assisted by mobile agents, it provides easy extensibility to existing systems. Third, it provides a mechanism to reconfigure a node's peers based on certain optimization criteria. Our extensive experimental studies show that PeerDB is a promising system for distributed processing.

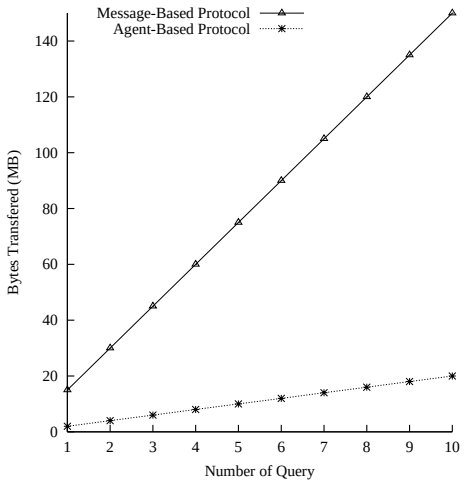


Figure 4.8: Communication Overhead

Chapter 5

PeerOLAP: An Adaptive P2P Network for Distributed Caching of OLAP Results [1]

5.1 Introduction

In this chapter, we will study various data placement strategies for P2P systems, in particular, how a query optimizer select a minimum cost query plan based on limited knowledge of its environment. We shall focus on data warehouse applications by extending BestPeer to support On-Line Analytical Processing (OLAP) queries. OLAP queries typically involve large amounts of data and their processing should be efficient enough to allow interactive usage of the system.

Distributed database technology is extensively used in data warehouses to access the operational databases, and to extract, clean and integrate the data. [40] discusses the particular issues of the data warehouse environment for distributed and parallel computation. The warehouse itself can also be implemented as a distributed database.

[1] This chapter is based on a collaboration [53], which has been partially used in a thesis [87]. The work reported here is a substantially extended version of [53].

If a central warehouse exists, standard replication methods can be used to transfer data to departmental data marts [86]. For decentralized implementations, where each department builds an independent data mart, [51] employs a global schema in a middleware to allow transparent access to all data. [13] has also proposed an architecture for executing OLAP queries over a decentralized schema. Its middleware component follows an economical approach, similar to the *Mariposa* system [101].

These systems assume that the users belong to the organization that owns the data warehouse and have access to the proprietary infrastructure. The query requirements are well defined and the problems are related to data placement, materialized view selection and query optimization, given a static network of servers.

Here, we investigate a different problem: a large number of ad hoc and geographically spanned users, sporadically accessing a number of separate warehouses and possibly correlating information from all of them. Imagine, for instance, many individual investors from all around the world trading stocks at the New York Stock Exchange (NYSE). Unlike professional stockbrokers, these users are unlikely to have any proprietary tool to access the stock market's warehouse; most of them probably connect with a simple applet through their web browser. The primary problem in this case is not the processing of the queries in the warehouses, but rather the efficient usage of the available bandwidth, since the size of results from OLAP queries may greatly vary from a few tuples to many megabytes of data. This can be especially true if the user is not satisfied with highly aggregated information, but needs access to detailed data in order, for example, to correlate New York prices with the ones from the major European markets.

Intuitively, the problem is similar to accessing web pages from remote web servers.

Caching in web proxy servers has been used extensively in practice to deal with the latency caused by slow network connections and accelerate the retrieval of the same URL from users in the same geographical area. [68] employs active caching techniques [21] to cache OLAP data together with web pages in web proxy servers using a three-tier architecture. [54] employs similar ideas to implement a multi-tier caching system for OLAP queries on a dedicated infrastructure.

In this work, we take a different approach by focusing on the client. Continuing with the previous example, assume that users from Singapore pose queries to the NYSE warehouse and some results are cached at their local computers [27, 28, 62, 94] which subsequent queries may reuse. However, the size of each client's cache is relatively small compared to the size of the warehouse, while the network cost of transferring large amounts of data from overseas is high. On the other hand, it is possible that some other user in Singapore, who has fetched part of the required data recently, can be accessed through a much faster network connection. By sharing their cache contents, all clients can benefit because the collective available space for caching is larger and the amortized network cost is lower.

In the following sections, we will describe PeerOLAP which is a distributed caching system for OLAP queries based on a Peer-to-Peer[2] (P2P) network. The contributions of this work include: (i) the proposal of the PeerOLAP architecture, (ii) the employment of three cache control policies that impose different levels of cooperation among the peers, and (iii) the development of adaptive techniques that dynamically reconfigure network structure in order to minimize query cost.

[2]The term 'P2P' has been used in database literature to identify systems where each node may act both as a server and a client, assuming static configuration [61]. Such systems are generalizations of the traditional client-server model and standard distributed techniques can be applied. Here, 'P2P' refers to dynamic systems with ad hoc participation, such as Napster and Gnutella.

PeerOLAP is complementary to distributed data warehouses, which deal with the efficient execution of OLAP queries, since the focus is on the effective utilization of client resources. The same relationship exists with middleware approaches like [68] and [54]. Also, the traditional client-side caching in client-server systems is a special case of our system, where client caches do not cooperate.

We focus on OLAP for several reasons: (i) OLAP data has a regular structure which allows the easy decomposition and reuse of previous results; (ii) the size of the results is typically large and justifies the overhead of searching neighbor peers; (iii) updates in data warehouses are infrequent compared to transactional databases, therefore the cached data is valid for a long time; and (iv) queries exhibit temporal and geographical locality; for instance many Singapore users are likely to request from NYSE similar data (e.g., data related to Singapore's ST index).

5.2 Background

Conceptually, data warehouses deal with multidimensional views of data. Under this model, there is a set of measures that are the objects of analysis, such as *sales*. The measures depend on a set of dimensions (i.e., business perspectives). As an example, consider *product*, *customer* and *supplier*. Thus, a measure is a value in the multidimensional space which is defined by the dimensions. Each dimension is described by a domain of attributes (e.g., product IDs). The set of attributes may be related via a hierarchy of relationships, a common example of which is the temporal hierarchy (day, month, year).

There are $O(2^d)$ possible group-by queries for a data warehouse with d dimensional attributes, which compose the data cube. A detailed group-by query can be used

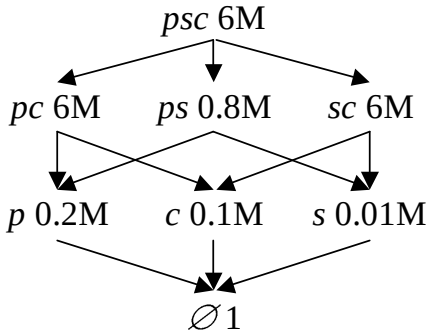


Figure 5.1: A Data Cube Lattice. The dimensions are *Product*, *Supplier* and *Customer*.

to answer more abstract aggregations. [49] introduces the search lattice L, which is a directed graph whose nodes represent group-by queries and edges express the interdependencies among group-bys. There is a path from node u_i to node u_j if u_i can be used to answer u_j (Figure 5.1).

A common technique to accelerate OLAP is to pre-calculate some aggregations and store them as materialized views, provided that some statistical properties of the expected workload are known in advance. [49, 97, 96] describe greedy algorithms for the view selection problem. These methods follow a static approach, where the views are selected once when the warehouse is set up. A dynamic approach is inspired by semantic data caching [26, 58]: instead of caching a list of physical pages or tuple identifiers, the results of previous queries together with their semantic descriptions are stored.

5.3 The PeerOLAP Network

The PeerOLAP network is a set of peers that access data warehouses and pose OLAP queries. Each peer P_i has a local cache and implements a mechanism for publishing its cache contents and its computational capabilities. Other peers can connect to P_i and request a result. P_i may either answer the query (or part of it) locally if it has the required data, or propagate the query to its neighbors. In either case, all results return directly to the peer that initiated the query. The goal of PeerOLAP is to act as a combined virtual cache, where all the components offer resources to lower query cost.

Figure 5.3 depicts a typical PeerOLAP network consisting of seven peers and two data warehouses. There is an arbitrary set of connections among peers denoted by solid lines, and each peer also connects directly to one or multiple warehouses simultaneously. Assume that P_2 issues a query q referring to chunks c_1, c_2 and c_3. If c_1 is already at the local cache, P_2 will send a request for c_2 and c_3 to its neighbors P_1 and P_3. P_1 contains c_2, therefore it computes an estimation for the cost of retrieving and transferring this result back to P_2, and at the same time it forwards the request to P_6. Note that both c_2 and c_3 are requested[3] since P_6 may be able to provide c_2 at lower cost compared to P_1.

In order to avoid flooding the network with messages, a maximum number of hops is assigned to each message. Assuming that this number is two, the query will not be propagated to the neighbors of P_6. On the other hand, P_3 will not forward the message although there is still one hop allowed, since a peer can direct to the warehouse only its local queries in order to avoid overloading the server with the

[3] A variation, where only the not-yet-found chunks are requested, is discussed in the next section.

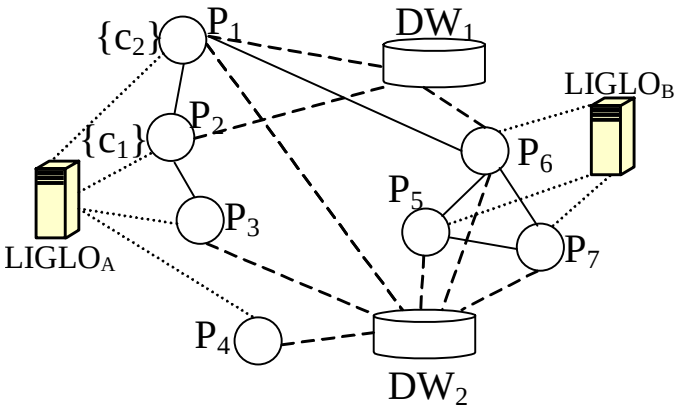


Figure 5.2: A Typical PeerOLAP Network

same query. There is also a mechanism for breaking message loops: each peer keeps a queue of the recent messages and rejects the ones that have been processed.

P_2 does not have any knowledge about the number of peers that will respond. Therefore, it waits until all the requested chunks are found or a timer expires. Missing chunks are requested from the data warehouse. Note that although the warehouse can provide any chunk, it is the last option due to the high network cost.

After search has terminated, P_2 decides which chunks to keep in the local cache. Each chunk is assigned a benefit value. If an incoming chunk c has a higher benefit than some cached results, these results are evicted and c is stored, else c is rejected. For chunks that are sent directly from the warehouse (meaning that they are not found in the neighborhood of P_2), we explore the option of caching them in some neighbor of P_2 (i.e. P_1 or P_3).

Since peers can enter and leave the network dynamically, a mechanism is necessary to provide the newcomer with an initial set of neighbors. Here, we employ LIGLO servers to maintain a list of online peers, together with details about the warehouses

that they access, their physical location, speed of network connection, etc. A newcomer peer P contacts a LIGLO server and gets a set of potential neighbors. Then P decides independently the set of peers that it will try to connect to. Except for LIGLO servers, the PeerOLAP network is fully distributed without any centralized administration point. Furthermore, LIGLO servers are not involved in query processing and can be completely eliminated if the set of initial neighbors can be otherwise determined; for instance, peers on the same segment of a LAN may connect to each other.

The set of initial neighbors is by no means optimal since their cached results may be irrelevant to P. Furthermore, connections may be dropped as some peers leave the network. Therefore, each peer implements a mechanism that constantly evaluates the current neighbors and drops or adds peers to the neighbor list in order to lower query cost. Intuitively, peers with similar query patterns should be neighbors. In such cases, if a result is not found in the initial peer, there is a high probability that one of the direct neighbors will contain it. Due to the limited availability of resources, each peer cannot have more than k neighbors, where k is a parameter of the system. Even if there are unlimited resources on a peer, it is not appropriate for the peer to have too many neighbors since the network will be overloaded with messages, most of them being negative responses.

Here, we make an effort to optimize the set of neighbors of each peer by formulating the problem as a second level of caching. The size of the second level "cache" is the number of available network resources while the "cached" objects are the connections to neighbors. Each neighbor is assigned a benefit and may be dropped if a more beneficial neighbor is found. Continuing our previous example, assume that during

the last 10 queries from P_2, five chunks were found in P_1, eight in P_6 and none in P_3. It is obviously beneficial for P_2 to have P_6 as a direct neighbor in order to avoid the overhead of reaching it through P_1. Therefore, the connection to P_3 is dropped and is substituted with a (virtual) connection to P_6.

In the next section, we describe in detail the components of our architecture, and present the query processing and caching algorithms. We also discuss the algorithm for network reconfiguration.

5.4 Peer Architecture

The PeerOLAP network consists of numerous low-end workstations which connect to data warehouses, pose OLAP queries and process results. Every peer maintains a local cache and implements a P2P protocol for connecting with other caches. The application layer is separated from the cache control unit; therefore the cache is not aware of the semantics of the data. Both the creation of the execution plan and the caching policy are fully decentralized.

Figure 5.4 depicts the architecture of an autonomous peer. There are two basic layers: the *application* and the *cache layer*. The application layer implements the user interface, the query optimizer and the query execution engine. It has knowledge about the schema of the warehouse and the semantics of the data. In our implementation, the application layer is built as a Java agent. When the user connects to a data warehouse (e.g., by accessing its web site), the warehouse server sends to the peer a mobile agent which implements all the logic of the application layer. The agent then connects to the cache layer, which is already running on the peer, and all the data requests are directed through the cache layer. More than one agent are allowed

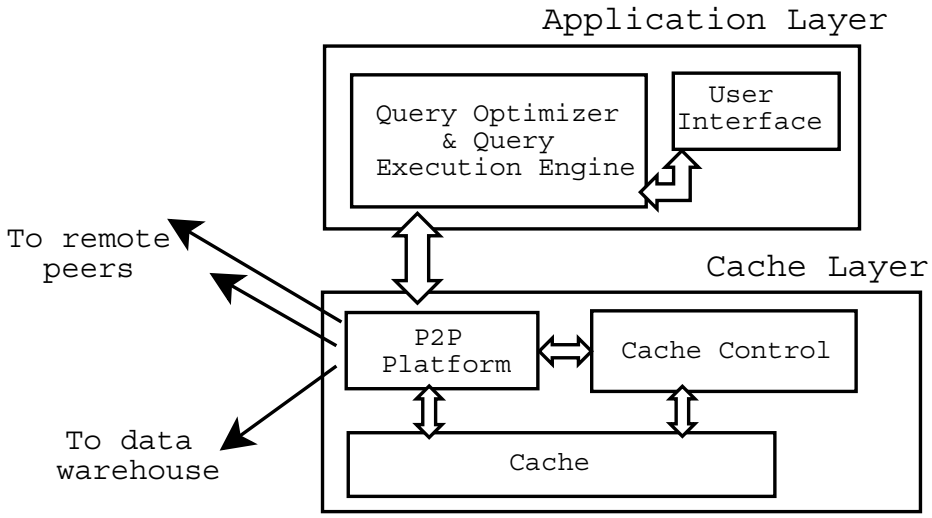


Figure 5.3: Architecture of a Peer

to run simultaneously at the same peer if the user wants to connect to multiple warehouses. In our implementation, the logic of all agents is the same although every warehouse supplies its own schema. PeerOLAP provides an environment where different mobile agents can reside and perform their tasks. The versatility to adapt to different requirements for query optimization and execution renders the system highly extensible and powerful. Note that the application layer does not have to be implemented as an agent. Assuming that the user routinely connects to some data warehouses, the client software can be permanently installed on the local peer.

The cache layer consists of three modules:

1. The local cache, which is organized as a chunk file [28].

2. The cache control module, which implements the admission and replacement policy of the cache.

3. The P2P platform, which implements low-level communication (among the

peers, and between the peer and the warehouse), data transfer and remote agent support. Also, in collaboration with the cache control module, it is responsible for network reconfiguration.

Apart from simplifying the development process, there is another advantage of dividing the peer into two layers: by distinguishing the cache from the semantics of the data, the cache can store, simultaneously, data from multiple warehouses. From the cache's point of view, each piece of data is a chunk, which is identified by a unique ID. It is the responsibility of the application layer to ask for the correct set of chunks and advise the cache about the benefit of storing a specific chunk. Therefore, each peer can support simultaneous access to multiple warehouses by allowing many agents to run at the same time. Also, a peer can store chunks that do not belong to its local warehouses, but are beneficial to some neighbors. In an extreme case, a peer may have only its cache layer running without executing any local application.

5.4.1 Cost Model

Let c be a chunk and $size(c)$ its size in tuples. $S(c, P)$ denotes the cost of computing c in node P. If P is a peer of the cache network and we do not allow any aggregation on cached results, then $S(c, P) = a \cdot size(c)$, where a is constant. On the other hand, if cached results can be further aggregated, $S(c, P)$ is the total cost of reading the required set of more detailed chunks and performing the necessary computations. The network cost N for transferring c from node Q to node P is:

$$N(c, Q \rightarrow P) = \frac{Cn(P \rightarrow Q)}{k} + \frac{size(c)}{Tr(Q \rightarrow P)} \tag{5.4.1}$$

where $Cn(P \rightarrow Q)$ is the cost of establishing a connection between the two nodes, k is the number of chunks that will be transferred together in a batch operation, and

$Tr(Q \rightarrow P)$ is the transfer rate between Q and P. If there is already an established connection between the two nodes, $Cn()$ is zero.

When c is asked for at peer P, the peer decides the location Q from where it will request the data. Therefore, the total cost T of answering c at P by using data from Q is:

$$T(c, Q \rightarrow P) = S(c, Q) + N(c, Q \rightarrow P) \tag{5.4.2}$$

Obviously, if the chunk exists locally (i.e. $P \equiv Q$), $N()$ is zero.

5.4.2 Query Processing

A query q has the form:

```
SELECT <grouping predicates> AGR(measure)
FROM data
WHERE <selection predicates>
GROUP BY <grouping predicates>
```

Let σ and γ be the set of selection and grouping predicates, respectively. View v is the representative view of q, if the set of dimensions of v is $\sigma \cup \gamma$. For example, a query that asks for the sum of sales of a set of products for each customer, corresponds to the *pc* view of Figure 5.1. A node in the PeerOLAP network can compute the results of such queries by first accessing the set C of required chunks at the same level of granularity as the representative view and then performing the necessary selections and aggregations on them. Here, we focus on the problem of locating, accessing and caching the chunks of C, therefore we consider queries involving selections on the grouping predicates only (i.e., $\sigma \subseteq \gamma$). Furthermore, the predicates of σ are such

that the results match the boundaries of entire chunks. More general queries can be computed by post-processing the chunks of C.

We assume that the warehouses are read-only, meaning that the clients cannot issue update statements to them. If its contents have changed, a warehouse must broadcast the relevant invalidation messages. Alternatively, it can set the expiration time in each chunks it computes.

Below, we will discuss two query processing policies, an eager and a lazy one, which differ on the amount of effort they put on constructing the execution plan.

Eager Query Processing (EQP)

Assume that a user issues a query q at peer P. The EQP policy answers q by performing the following steps:

1. The query is decomposed into chunks at the same granularity as the representative view. Let C_{all} be the set of required chunks.

2. P first checks its own cache. Let C_{local} be the set of chunks that are present and C_{miss} be the remaining chunks.

3. P sends a message to its neighbors $Q_1, \ldots, Q_k$ asking for the C_{miss} set. If Q_i has a subset of C_{miss}, then it estimates the cost $T(c_i, Q \rightarrow P)$ for each of the chunks and sends these estimations to P. If a peer does not have any of the required chunks, it does not respond. In any case, Q_i propagates the request for the entire C_{miss} set to its own neighbors recursively, until the maximum allowed number of hops is reached.

4. P keeps receiving responses for a period t, after which it assumes that no more results are expected.

5. Let C_{peer} be the subset of C_{miss} that is found in the PeerOLAP network. P

constructs the execution plan for C_{peer} in a greedy manner: A chunk c_i is randomly selected from C_{peer} and is assigned to Q_i, where Q_i is the peer that can provide c_i at the lowest cost. Next, a chunk c_j is selected from the remaining chunks in C_{peer}. Let Q_j be the peer that provides c_j at the minimum cost. If Q_i also contains this chunk, the algorithm checks whether the total cost $T(\{c_i, c_j\}, Q_i)$ of acquiring both chunks from Q_i is smaller than $T(c_i, Q_i) + T(c_j, Q_j)$ in which case it assigns c_j also to Q_i. The process continues for the rest of the chunks in C_{peer}. Observe that acquiring multiple chunks simultaneously from the same peer may be cheaper, because the cost of sending messages and initializing the network connections is shared.

6. P initializes direct connections to the peers defined by the execution plan and requests for the corresponding chunks. The peers send back the chunks that have not been evicted in the meantime. Let $C_{evicted}$ be the set of evicted chunks.

7. The set C_{DW} of chunks still missing is: $C_{DW} = C_{miss} - (C_{peer} - C_{evicted})$. P gets these chunks directly from the warehouse.

8. P composes the answer and returns it to the user. The new chunks are sent to the cache control module and any necessary reconfiguration of the network is performed.

Only chunks at the same aggregation level as the query are considered. By exploiting the possibility of computing missing chunks by further aggregating the cached results, a more efficient execution plan may be constructed. However, in such cases, the number of ways to compute a chunk grows exponentially to the number of dimensional attributes, and the construction of the execution plan becomes a difficult optimization problem which is outside the scope of the discussion here. Nevertheless, the cost model is general enough to deal with aggregations if they are performed

within the scope of a single peer.

Lazy Query Processing (LQP)

The previous policy attempts to expand the search space as much as possible in order to locate the maximum number of chunks. The drawback, however, is that the system is overloaded with messages, many of which are redundant either because some of the accessed peers are irrelevant to the query or because multiple peers contain the same chunk, and their cost difference does not justify the high message overhead. Here, we present a second policy, called *Lazy Query Processing*, which tries to reduce the number of visited peers.

LQP is similar to EQP except for step 3: P sends the request to all of its neighbors $Q_1, \ldots, Q_k$, but each neighbor will propagate the request only to its most beneficial neighbor. In addition, if Q_i can answer some of the chunks, it removes them from the propagated message. As a result, if the entire query can be answered by Q_i, the message is not propagated. The process is repeated until the maximum allowed number of hops h_{max} is reached. If each peer has k neighbors the number of messages are $O(k \cdot h_{max})$ while for EQP this number becomes $O(k^{h_{max}})$.

We have already mentioned that the new chunks are forwarded to the cache control module, which decides whether it is beneficial to store some of them locally. The next paragraph explains this issue; the notion of a peer's benefit will be clarified in Section 5.4.4, where we discuss the adaptive behavior of the system. The LQP policy fits well in this concept since intuitively, we wish to form small sub-networks with similar query patterns.

5.4.3 Caching Policy

In order to define the cache control policy, a benefit metric $B()$ is assigned to every chunk c at a peer P. Naïve least recently used (LRU) or least frequently used (LFU) schemes are inapplicable for OLAP queries because the cost of computing chunks varies greatly at different levels of aggregation. [94] defines a metric, which is a function of the cost to compute a result normalized by its size and frequency. The same metric is used in [62]. [28] proposes a caching algorithm, called ClockBenefit, which is a generalization of LRU. The benefit of a chunk is measured by the fraction of the base table that it represents. Therefore, if there are n chunks in a view v, the benefit of each chunk is $\frac{|D|}{n}$, where $|D|$ is the size of the base table. Since the number of chunks at higher levels of aggregation is small, they have a higher benefit. The benefit is thus proportional to the cost of computing a chunk. The exact cost is not important in their case, since the back end always computes each chunk from the base tables and also the cache does not perform any aggregation.

Here, we define the benefit $B()$ of a chunk c in a peer P as:

$$B(c, P) = \frac{T(c, Q \rightarrow P) + a \cdot H(P \rightarrow Q)}{size(c)} \tag{5.4.3}$$

where $H(P \rightarrow Q)$ is the number of hops from peer P to Q, and a is a constant representing the overhead of sending one message. Intuitively, a high value of $H()$ denotes that it is difficult to locate a result, therefore it is more beneficial to keep it locally. Notice that the cost of locating a result is proportional to the number of hops rather than the number of peers visited, since a peer sends each request to all its neighbors in parallel. The benefit value is normalized by dividing the total cost of obtaining a chunk by its size.

Recall that $T()$ is the total cost of computing and transferring a result; its inclusion in the benefit denotes that results which are expensive to obtain, should be stored locally. Although our caching algorithm is similar to ClockBenefit, the $\frac{|D|}{n}$ metric is not suitable, since we allow pre-aggregation at the data warehouse. Therefore, the computation cost of a chunk depends on the set of materialized views.

PeerOLAP allows replication of a chunk in many peers. Replication should be performed only if it is absolutely necessary, because it consumes space that could be used for other chunks. The above mentioned benefit function facilitates the replication of objects in a controlled manner. Let c be a highly aggregated chunk that is asked for the first time and is computed from the warehouse. As the size of the chunk is small, so even though both the computation and network costs are expected to be high, the benefit will still be high. Assume that P caches c, and Q requests c from P. Since the cost of retrieving and transferring c is now lower, the probability that Q caches the same result also decreases. If Q needs c in the future, it can find it in P and its available cache space can be used for more beneficial chunks.

Admission and Replacement Algorithm

It should be obvious from the previous example that an incoming chunk is not cached by default, but only if it is beneficial enough for the peer. The admission and replacement algorithm called *Least Benefit First* (LBF) is presented below. LBF is an LRU-like algorithm which considers the benefits of the objects. It assigns a weight $W()$ to every cached chunk, which initially is equal to the chunk's benefit. $W()$ is decreased each time a new chunk is considered for admission, and is restored to its

original value whenever the chunk is accessed again. When a new chunk c_{query} arrives, LBF sorts the cached chunks in ascending weight order and marks as potential victims the first ones which, if evicted, will release enough space for c_{query}. In order to avoid accessing the entire cache index each time a new object arrives, we employ CLOCK [28]. Observe that the sorting step of line 8 requires at most $O(\log |CIndex|)$ time, where $|CIndex|$ is the number of objects in the cache, because the objects are previously sorted and in every step the position of only one object may change. The new chunk is stored only if its benefit is greater than the combined weight of the victims.

Algorithm 1: LeastBenefitFirst(c_{query})

```
1:  /* c_query is the query chunk */
2:  if c_query is already in the cache then
3:     W(c_query) := B(c_query, P) /* reset W(c_query) to its initial value */
4:  else
5:     Let c_CLOCK be the chunk corresponding to the CLOCK position
6:     W(c_CLOCK) := W(c_CLOCK) - B(c_query, P)
7:     Advance CLOCK position
8:     CIndex := List of all cached chunks sorted in ascending W(c_i) order
9:     victims := ∅
10:    next := 0
11:    while FreeCacheSpace + Σ_{v_i ∈ victims} size(v_i) < size(c_query) do
12:       victims := victims ⋃ CIndex_next
13:       next++
14:    end while
15:    W_victims := Σ_{v_i ∈ victims} W(v_i) /* the total weight of all victims */
16:    if W_victims ≤ B(c_query) then
17:       Evict victims from cache
18:       Insert c_query
19:       W(c_query) := B(c_query, P)
20:    end if
21: end if
```

LBF resembles the GD [112] algorithm which is used for caching web pages. GD,

however, will always cache a new object even if it needs to evict more beneficial ones. In our case, such behavior is contradictory with the controlled replication scheme that we aim to achieve.

The LBF algorithm controls the local cache of each peer. Next, we will present three policies, which describe the behavior of the entire system and enforce a progressively higher degree of collaboration.

Isolated Caching Policy (ICP)

The rational behind the *Isolated Caching Policy* is that a peer P is completely autonomous and will attempt to benefit from the other peers in a greedy manner. P publishes its cache contents and employs the algorithms that have been described before, but it does not count the hits on its cache by the other peers. Therefore, if a neighbor Q requests a chunk c, which is in the cache of P, P will provide c but it will not update its weight back to the original value (line 3 of LBF). If c is not important to P, it will eventually be evicted even if it is beneficial for the neighbor peers.

Although ICP disregards collaboration, it suits the philosophy of P2P systems. Recall that the peers do not necessarily belong to the same organization. Instead, they may belong to autonomous users who would like to have complete control of the resources they provide.

Hit Aware Caching Policy (HACP)

In contrast to ICP, the *Hit Aware Caching Policy* considers the hits from other peers in an effort to ensure that the caches cooperate with the aim of minimizing the total query cost. In order to comprehend this, consider again the benefit function of LBF:

If P finds a chunk c in a peer Q, then $B(c, P)$ is lower than if c were answered by the warehouse; therefore, the probability that P caches c decreases. Intuitively, LBF implements a passive way of collaboration based on an optimistic approach, since it assumes that c will still be in Q when it needs it again. In order for this to happen, HACP increases the benefit of c in Q, whenever c is used by another peer.

Voluntary Caching

The *Voluntary Caching Policy* attempts to exploit under-utilized resources that may exist in some peers while avoiding wasting any result that has been obtained from the warehouse. Assume two peers, P and Q, where P exhibits a heavy workload and has a cache full of high-benefit chunks while Q poses a few queries and has a cache that is under-utilized with low-benefit chunks. P asks for chunk c, which is found only at the warehouse. Although c has a substantial benefit, assume that P cannot admit it because the benefit of the potential victims is higher. Instead of discarding it, the voluntary caching policy will ask whether any neighbor of P can cache the result. If such a neighbor, say Q, exists, c will be forwarded to it. In case that multiple neighbors volunteer to cache c, P selects the one with the highest $B(c, Q) - B(victims, Q)$ value. Naturally, P has to pay the cost of transferring c to Q which is added to the total query cost. The intuition is that this cost will be amortized by subsequent requests for c. Note that due to the transferring cost, the benefit of caching c at Q becomes:

$$B(c, Q) = \frac{T(c, DW \rightarrow P) - T(c, P \rightarrow Q)}{size(c)} \tag{5.4.4}$$

where DW is the warehouse, P the requesting peer and Q the caching peer.

The voluntary caching policy may work either in conjunction with ICP (v-ICP) or with HACP (v-HACP).

5.4.4 Network Reorganization

The previous techniques attempt to minimize the total query cost by constructing efficient execution plans and caching query results. In this section, we try to optimize the network structure by creating virtual neighborhoods of peers with similar query patterns. The goal is to assign a set of neighbors to each peer P, so that there is a high probability for P to obtain missing chunks directly from them without having to search a large part of the network. These neighbors are the only ones that P can visit directly.

Ideally, a peer should be able to communicate with all others by direct connections, in order to have complete knowledge about the contents of all caches. This is impractical for two reasons: (i) network connections consume resources at the peer, and (ii) the entire network would be flooded with messages. Nevertheless, as we shall show, experimentally good results can be obtained even with a limited set of *beneficial* neighbors. Note that the initial neighbors that a peer connects to when entering the network are nothing more than starting points and they are by no means optimal. Additionally, even if a good set of neighbors is known at connection time, query patterns may change or some of the neighbors may leave the network.

Motivated by the above, we formulate the problem as a special case of caching. Each peer has a number of available network resources, which are the equivalent of cache cells, and the objects that are cached are the direct connections to other peers. Each connection is assigned a benefit value and the most beneficial connections are

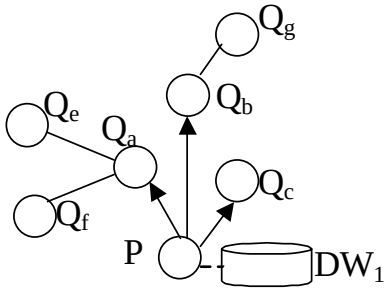


Figure 5.4: A Sample Network Structure

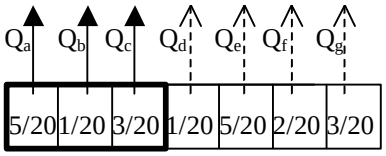


Figure 5.5: The LFU Connection Cache at Peer P. (Numbers represent hit ratios.)

selected to be the peer's neighbors.

Similar to the LBF policy, we follow an optimistic approach assuming that if a peer is contacted once, it can be found again later. From this assumption, and given that the cached objects cannot be further aggregated, it is clear that a hit to any peer is of equal benefit, regardless of the chunk that is retrieved. Recall that in any case, the results are sent back to the peer that initiated the query via a direct connection that is opened for the transfer. Therefore, in Figure 5.4, if Q_f provides a very beneficial object to P while Q_e provides a less beneficial one, each connection is charged with one hit. For these reasons, we use a simple LFU policy for caching network connections.

Since the number of allowed network connections nc_{max} is expected to be small, we can maintain accurate statistics for more than nc_{max} connections. For instance, in Figure 5.5, $nc_{max} = 3$ and the neighbors of P are $Q_{a,b,c}$, but we maintain a cache of seven connections. The set of neighbors is not altered every time there is a change in

Table 5.1: Parameters Derived from the Prototype

Parameter	Value	Comments
TR_R	3.68891 KB/sec	Average transfer rate between remote peers (WAN)
TR_L	594.9347 KB/sec	Average transfer rate between local peers (LAN)
TR_D	4675.945 KB/sec	Average transfer rate from the disk
AMT_R	1.2975 sec/mes	Average time per message between remote peers (WAN)
AMT_L	0.3765 sec/mes	Average time per message between local peers (LAN)
ICT_R	3.68 sec/con	Average time to initiate a remote connection (WAN)
ICT_L	0.36 sec/con	Average time to initiate a local connection (LAN)

the LFU cache in order to avoid frequent reconfigurations of the network. Rather, the system waits until k requests are served (where k is a system parameter), and then selects as neighbors the nc_{max} more beneficial connections. In the previous example, if it is already time for reorganization, Q_b will be evicted and it will be replaced by Q_e.

Notice that the network connections considered here are virtual and differ from physical network connections. Consequently, the "neighbor" relation is asymmetrical: if P has Q as a neighbor, the opposite may not necessarily be true. In cases where the relations are symmetrical, the two sides can connect with each other using the same physical connection, thus saving the cost of initializing new connections. Here, we do not consider the minimization of this cost.

Table 5.2: The Schema of the APB Dataset. The values represent the size of the domain in each dimension at the corresponding level of hierarchy.

	Product	Customer	Channel	Time
L_0	1	1	1	1
L_1	4	99	9	2
L_2	15	900	-	8
L_3	75	-	-	24
L_4	300	-	-	-
L_5	605	-	-	-
L_6	9000	-	-	-

5.5 Experimental Evaluation

We evaluated the performance of PeerOLAP using two implementations. The first one was an actual prototype consisting of a data warehouse server in Hong Kong and 10 peers in Singapore. It was used to test the fundamental aspects of the architecture, and to derive real-life parameters that were subsequently used by a simulator in the second implementation to evaluate the behavior of PeerOLAP in various situations. Table 5.1 illustrates this set of parameters, which will be used in this section.

We employed the dataset from the APB benchmark [82] in addition to a synthetic dataset (SYNTH) which was also used by [28] (see Tables 5.2 and 5.3). The total space of the entire cube was around 3.5G tuples for APB and 69M tuples for SYNTH. The total space was divided in chunks in a way that the chunk dimension range at any level was kept proportional to the number of distinct values at that level. The size of the largest chunk was 1M tuples.

The *Detailed Cost Saving Ratio* (DCSR) [62] was employed to measure the results. DCSR is defined as:

$$DCSR = \frac{\sum_i wcost(q_i) - \sum_i cost(q_i)}{\sum_i wcost(q_i)} \tag{5.5.1}$$

Table 5.3: The Schema of the SYNTH Dataset

	D_1	D_2	D_3	D_4
L_0	1	1	1	1
L_1	25	25	5	10
L_2	50	50	25	50
L_3	100	-	50	-

where $wcost(q_i)$ is the total cost of answering the query q_i in the worst case, and $cost(q_i)$ is the cost achieved by the system. For the worst-case scenario, we assumed that the peers did not have any cache[4], so all the queries must be answered by the w

co

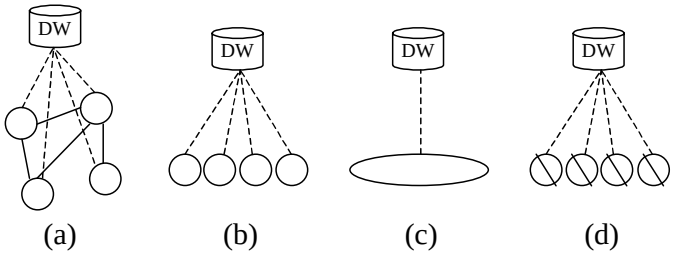


Figure 5.6: Configurations with One Data Warehouse. Dashed lines represent remote connections, and solid lines local ones: (a) PeerOLAP, (b) client-side cache, (c) one large cache, and (d) clients without cache

The tested configurations consisted of one data warehouse at a remote location (i.e., the transfer rate of the connection was TR_R) and a set of one to 100 local peers. The speed of all local connections was set to TR_L.

[4]Theoretically, the worst-case cost can be higher, due to messages. This is not significant for our results, since we are interested in the relative performance of different policies.

5.5.1 PeerOLAP vs. Client-Side Cache Architecture

In the first set of experiments, we compared PeerOLAP against a traditional client-server architecture with client-side caching (C-S) (Figure 5.6b). First, we considered the best case for PeerOLAP, where all peers are connected to each other (i.e., clique network). We used 10 peers and varied the cache size of each from 0.001% to 10% of the total data cube size. The query set consisted of 20,000 queries following the 80-20 rule (i.e., 80% of the queries accessed a hot region representing 20% of the entire data cube). Each peer initiated the same number of queries. For fairness of comparison with C-S, PeerOLAP used its most naïve configuration: the optimizer employed the lazy policy (LQP) and the cache policy was ICP.

The results are shown in Figures 5.7 and 5.8. In the same figures, we plot in the results of a hypothetical one-peer system (Figure 5.6c) having a cache size equal to the sum of the caches of all peers. This configuration, called CentralCache, represents the optimal case of the system. It is clear that in a clique configuration, PeerOLAP achieves near-optimal performance. The cost difference from CentralCache is due to the replication of some objects, which is difficult to avoid completely, and the cost of the messages. PeerOLAP easily outperforms C-S as expected. The results from both APB and SYNTH dataset are similar, although the absolute values differ. Since the trends were the same for all our experiments, in the following, we only present the results from SYNTH.

Next, we tested a more realistic configuration: each peer was connected to four others only, and the maximum hops allowed for search was set to three. The cache size of each peer was set to 1% of the total cube size, while all policies remained the same as before. The number of peers varied from 10 to 100. The query set was generated

as follows: The peers were divided into groups of 10. For each group, we provided a separate query set following a 90-10 distribution, and there was no intersection among the hot regions of different groups. Again, 20,000 queries were generated and each peer initiated the same number of queries. The results are presented in Figure 5.9.

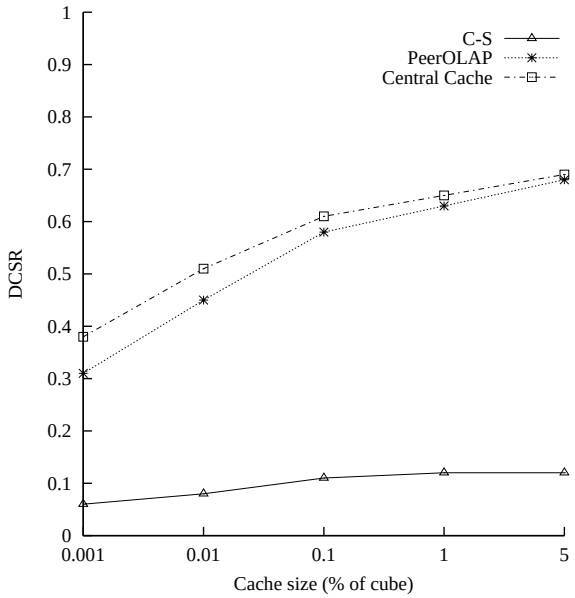


Figure 5.7: PeerOLAP vs. Client-Side Cache System: (APB Dataset)

The performance of C-S is almost constant since, irrespective of the number of peers, the size of an individual cache remains the same. As expected, CentralCache also improves when the size of the total cache increases. The behavior of PeerOLAP is more complicated: for 10 peers, there is only one group, and the system attempts to exploit the contents of neighbor caches as before. However, its performance is now not very close to optimal, because the number of neighbors and the number of hops are limited. Although in the best case, each peer can reach 12 others (i.e., the number of neighbors times the number of hops), the structure of the network may contain loops, so the actual number of peers that are explored is lower. Due to the limited knowledge

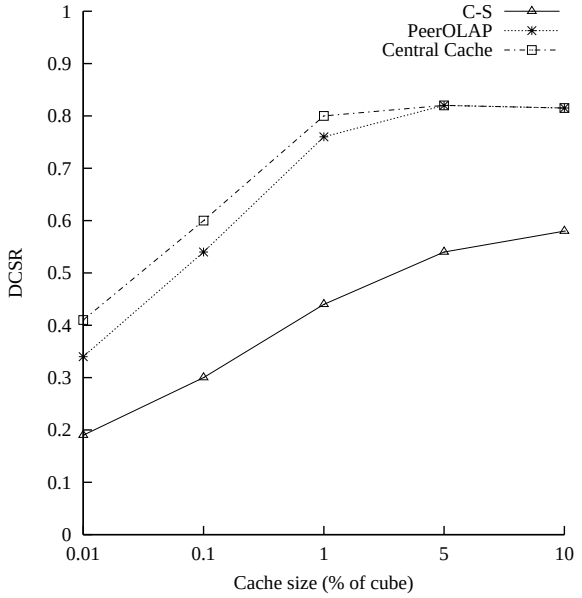


Figure 5.8: PeerOLAP vs. Client-Side Cache System: (SYNTH dataset)

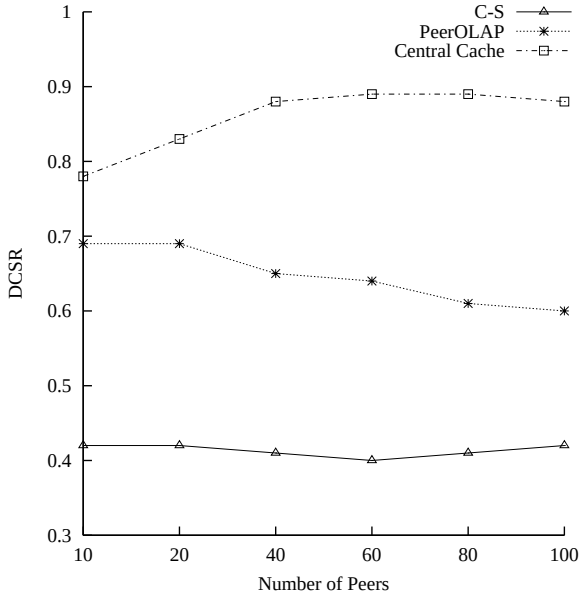


Figure 5.9: Groups of 10 Peers Accessing the Same Hot Region (Four Neighbors per Peer, Three Hops Allowed)

of the contents of other caches, the performance drops. This is more obvious when the number of peers increases. More peers with irrelevant data are inserted; therefore, it is more difficult for a peer to find others with similar workload. Nevertheless, even when there are 100 peers in the network, PeerOLAP is still considerably better than C-S, partially because it can locate peers in the same group, and also because it takes advantage of similarities in the "cold" part of the workload.

Notice that the performance of PeerOLAP drops because we add peers with different workload. If more peers with similar workload are inserted, the performance typically increases, or remains the same in the worst case.

5.5.2 Evaluation of the Query Optimization Strategies

The next experiment evaluated the performance of the eager (EQP) and the lazy (LQP) query optimization strategies. We used a network of 100 peers, each equipped with a cache space equal to 1% of the data cube space. The caching policy was set to ICP and network reorganization was disabled. The query set consisted of 10 groups with 10 peers each, and every group accessed a different hot region, as before. First we set the maximum number of hops to three and we varied the number of neighbors per peer. The results are shown in Figure 5.10.

Naturally, when there are zero neighbors, PeerOLAP is equivalent to C-S. When the number of neighbors increases, the knowledge of other peers' contents also improves, leading to better performance. The performance gain is almost linear for LQP since the maximum number of peers it can search is also a linear function of the number of peers. EQP, on the other hand, can explore up to $O(n^{hops})$ peers, where n is the number of neighbors. For example, when $n = 6$, EQP may potentially contact all

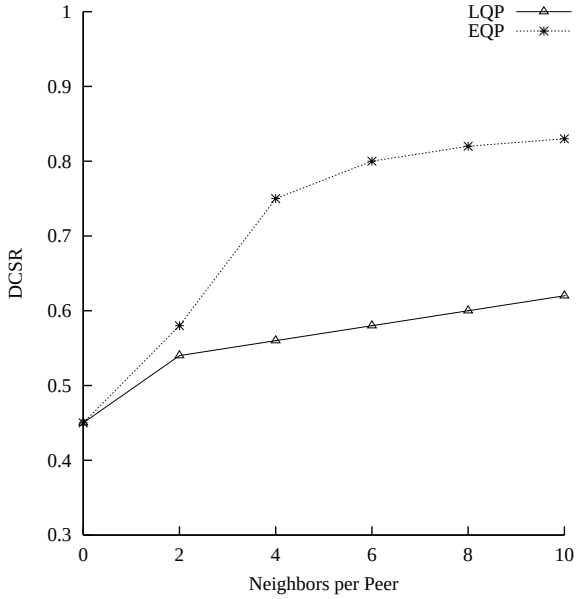


Figure 5.10: Query Optimization for a Network of 100 Peers and Three Hops

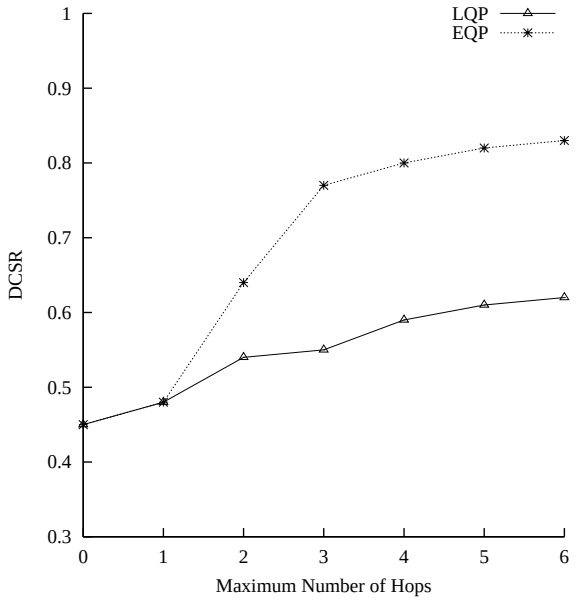


Figure 5.11: Query Optimization for a Network of 100 Peers and Four Neighbors Per Peer

the nodes (depending of course on the network structure) since $6^3 = 216$. Therefore the performance improves fast until it is almost equal to optimal. Similar results are shown in Figure 5.11, where the number of neighbors is four and the number of hops is varied. Notice that when the number of hops is one, the two policies are equivalent, since LQP always searches all its direct neighbors.

From these results, one might suggest that it is always preferable to follow the EQP strategy. However, the performance metric we have used here is based on the total execution cost and does not provide any information about the response time. EQP transmits a large amount of messages in the network. If all these messages need to be simultaneously processed, the response time will be affected considerably; such behavior contradicts user requirements.

5.5.3 Evaluation of the Caching Policies

In this set of experiments, we evaluated the performance of the caching policies. We used a clique network consisting of 10 peers and we generated query sets consisting of 20,000 queries following a 90-10 distribution. In contrast with the previous experiments, here, the number of queries each peer initiated was not the same for all peers. In each dataset Q_k, one of the peers received k queries and the rest were divided equally among the remaining nine peers. For instance, in the Q_{90} query set, 90% of the queries would be assigned to one peer, and the rest would receive 10/9=1.1% of the queries. In this way, we sought to simulate situations where some peers use their resources heavily while others under-utilize theirs.

We started by evaluating the LBF and LRU algorithms which were aimed at controlling the local cache of each peer. The cache size of each peer varied from

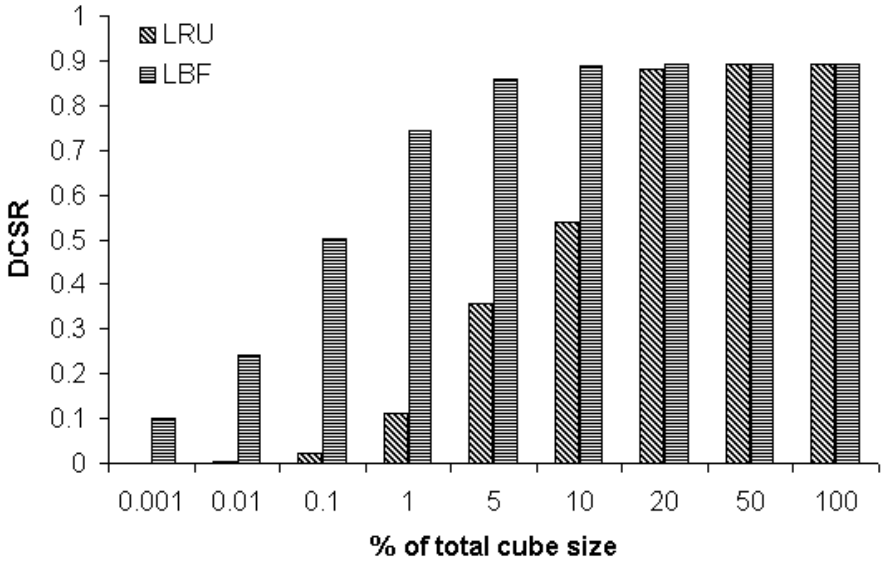


Figure 5.12: Comparison of the LRU and LBF

0.001% to 100% of the total cube size. The ICP policy was used in the experiment. The results are presented in Figure 5.12. Obviously, as the total cache size contributed by each peers increases, the performance of LBF and LRU improves, due to having more space for storing additional chunk objects. However, LBF outperforms LRU in all the cases; its performance gains in the small cache environments (i.e., 0.001% to 10%) are especially remarkable. When the cache size is small, the replacement of cache objects becomes frequent, therefore choosing a right object to be replaced is crucial for system performance. LRU replaces a chunk without considering the computation cost of the chunks at different levels of aggregation, causing poor performance when compared to the LBF scheme. In contrast, the LBF scheme defines a beneficial matric for objects evaluation in the replacement process. It considers the computing cost at the different levels of aggregation of the chunk, the number of hop to obtain the chunk, and the cost to transfer the chunk. In other words, if it is difficult to locate

the result, it is more beneficial to keep it locally.

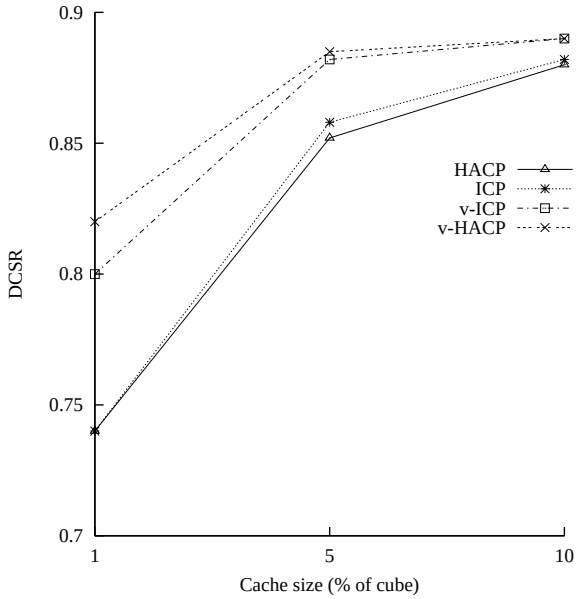


Figure 5.13: Comparison of Caching Policies

Next, we investigated the performance differences of the four global caching policies, i.e., ICP, HACP, vICP and vHACP. Figure 5.13 compares the four combinations of caching policies for the Q_{90} query set, with cache sizes varying from 1 to 10%. The experiments reveal that ICP and HACP have negligible performance differences. Moreover, there are cases where HACP performs slightly worse than ICP. This can be explained by the following example: Assume that P fetches c_Q and Q fetches c_P from the warehouse and store them in their local cache with high benefit values. Then P and Q request c_P and c_Q respectively. c_P is not cached in P (neither is c_Q cached in Q) because the benefit of such caching is low as the result is already stored in the neighbor peer. At the same time, because of the HACP policy, c_P is forced to remain in Q (and c_Q in P). The result of this kind of "deadlock" is that both peers must pay the network cost of fetching the result from their neighbor, in contrast with the ICP

policy which would eventually enable each peer to cache the correct chunk. Assigning a lower weight to the remote accesses, compared with the local ones, only reduces but does not solve the problem.

Although HACP is not very beneficial itself, it combines well with the voluntary caching approach. In the Q_{90} dataset, there are nine nodes which are under-utilized. Voluntary caching allows some of the data from the heavily loaded peer to use the available resources of its neighbors. Therefore, both v-ICP and v-HACP perform better than ICP and HACP. v-HACP is better than v-ICP because it allows the heavily loaded peer to inform the others that the chunks it has previously provided are still useful. Nevertheless, again the performance difference is not significant; the major performance gain comes from voluntary caching.

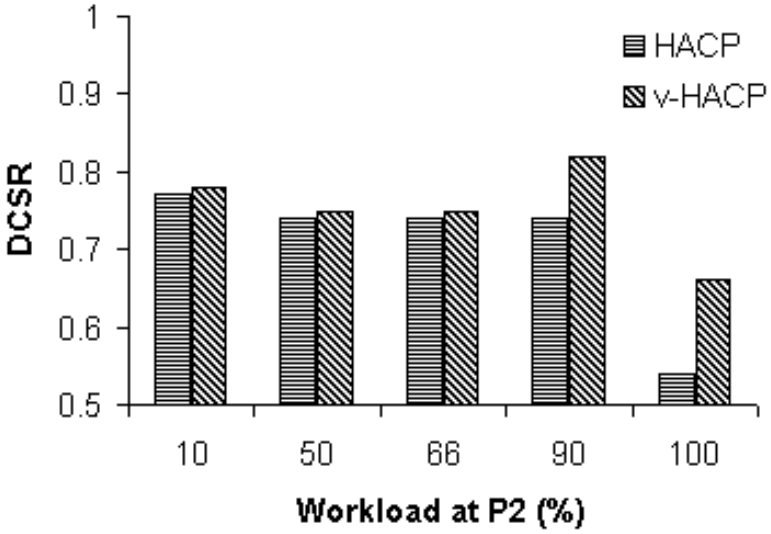


Figure 5.14: HACP vs. v-HACP for $Q_{10}, Q_{50}, \ldots, Q_{100}$ Query Sets

In Figure 5.14 we further investigate this issue: we compare HACP and v-HACP for workloads with different skew. We set the cache size to 1% and used query sets

varying from Q_{10} to Q_{100} (i.e., all the queries are initiated by the same peer). v-HACP is better in all cases; however, when all peers are significantly loaded, the difference between the two policies is not large. Nevertheless, when some peers are under-utilized, v-HACP is clearly better. This is obvious in the extreme case, where all the queries are asked by the same peer, while the rest just share their caches. If voluntary caching is not used in this case, the caches of all nine peers are always empty; this explains the substantial performance difference when v-HACP is employed. Note that the results among different query sets are not comparable.

Figure 5.15 presents the performance of each individual peer for the Q_{90} set with the cache size set to 1%. Obviously, P_2 is the peer which initiates 90% of the queries. We have shown before that the overall performance of the system improves due to voluntary caching. Figure 5.15 reveals that in addition to the heavily loaded peer, other peers may also benefit, but some may exhibit worse performance. This is true for both v-ICP and v-HACP, although the peers are affected in different ways.

In the previous experiments, we assumed all the caches were cleared before the experiments started. Although this environment setting is useful for measuring the benefits of different caching policies, it does not capture the "*new-join*" situation, where a new peer joins a network which has been operating for a long period. We studied the "*new-join*" situation in another set of experiments by feeding the system with training sets, thus allowing the peers to fill up their caches before the experiments began. In Figure 5.16(a), the cache size is 1% and the training dataset varies from 0% to 10%, 20%, 50% and 80% of the total data cube size. Similarly, the cache size in Figure 5.16(b) and Figure 5.16(c) is 5% and 10% respectively.

Several interesting observations may be made from the results. First, all four

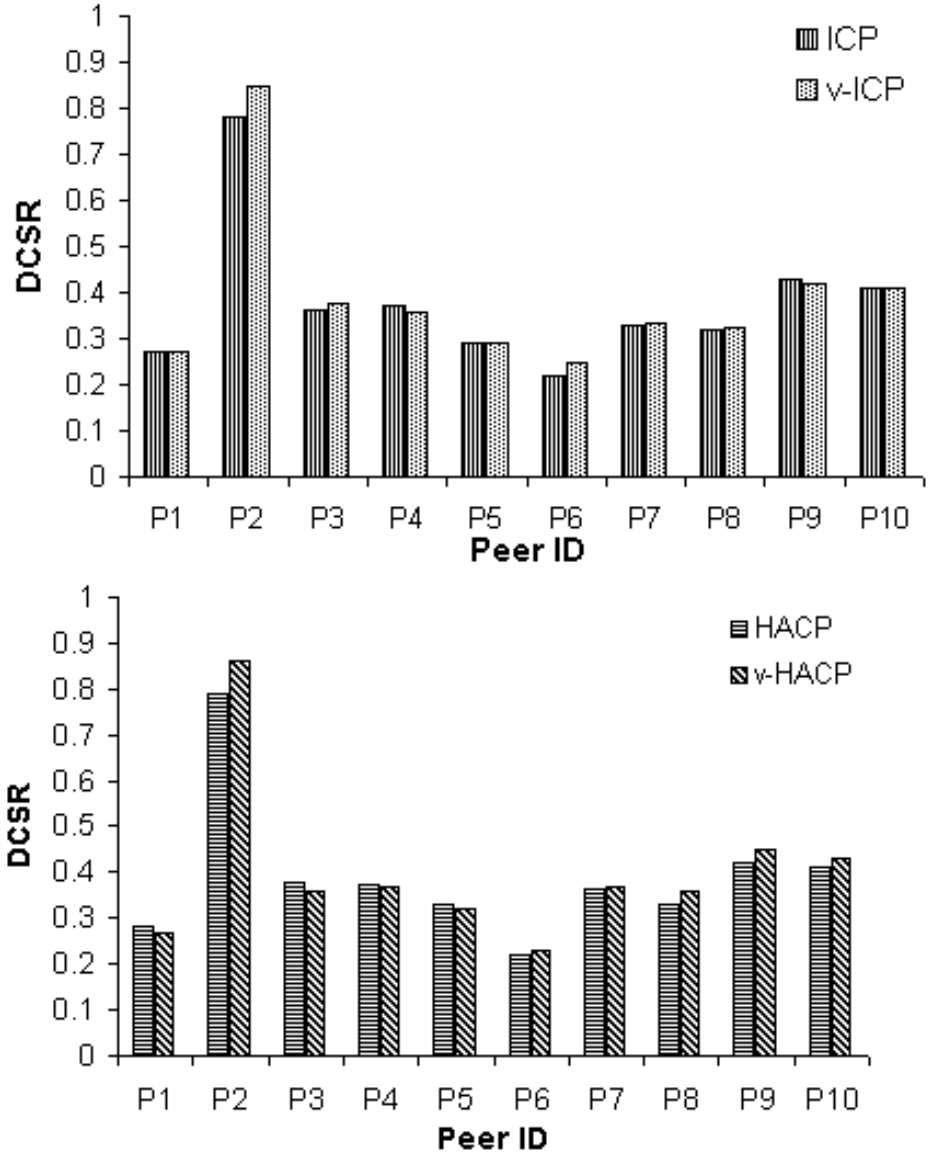
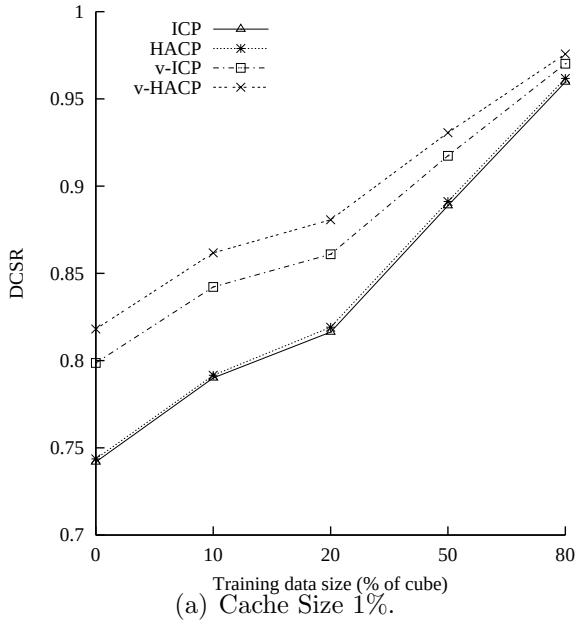
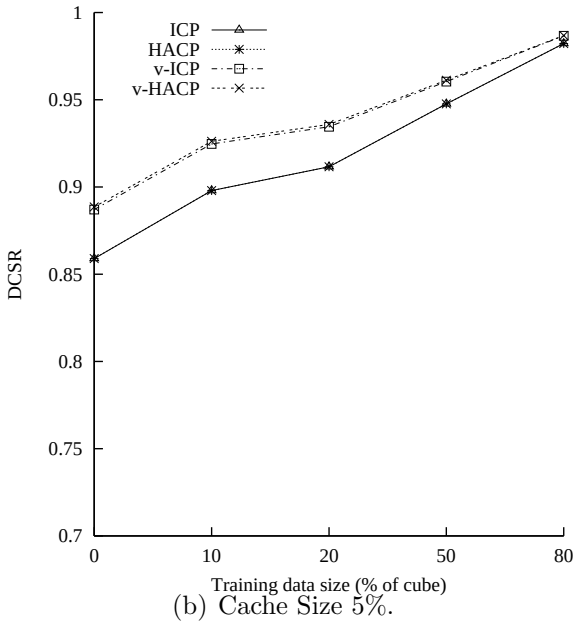


Figure 5.15: DCSR Achieved by Each Individual Peer for Q_{90} with a Cache Size of 1%: (top) Isolated Caching Policy, (bottom) Hit Aware Caching Policy

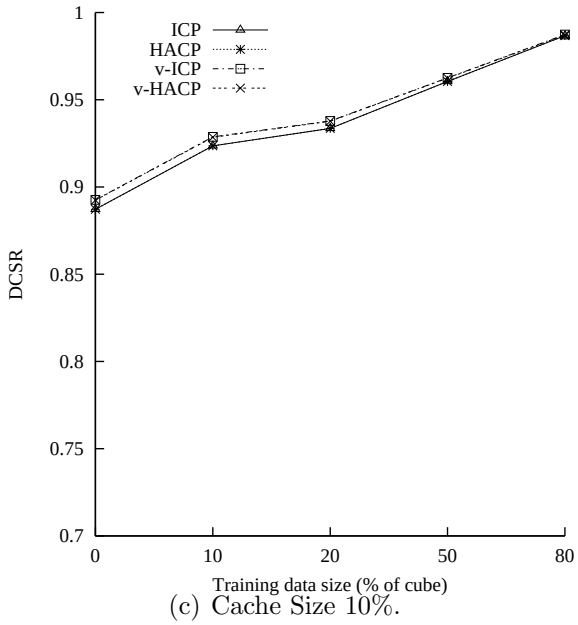
caching policies achieve performance gain when the training size increases. As the training size increases, more *useful* chunks are stored in the local peer's cache, since the LBF algorithm is able to evaluate correctly the degree of benefit in keeping a chunk in the local cache correctly. As a net result, it reduces the number of costly requests to the central warehouse. This suggests that in practice, a peer will enjoy better performance gain by joining the PeerOLAP network since the event of empty caches will happen only once – during the time of system initialization. When the system matures, all the caches are filled with objects. As a result, collaboration in a PeerOLAP network benefits all peers in the community. Second, in contrast with voluntary caching policies, ICP and HACP show little difference in their performance, regardless of the training data size in all three cache-size environments: small (1%), medium (5%) and big cache size (10%). This is due to a lack of global knowledge of the environment in the two policies, unlike the situation with the vICP and vHACP policies. The explanation for Figure 5.13 can similarly be applied here. Third, when the medium cache size is used, the different voluntary caching approaches, i.e., vICP and vHACP, have negligible performance differences regardless of the training data size. However, in general, voluntary caching approaches are still superior to ICP and HACP. Finally, all caching policies have negligible performance differences regardless of the training data size in the large cache size environment. When the cache size increases as in Figure 5.16(b) and Figure 5.16(c), the performance differences are not significant for all policies due to the large cache pool in the community, which neglects the needs of the advance replacement and collaboration policies, since most of the chunk objects can be stored locally.



(a) Cache Size 1%.



(b) Cache Size 5%.



(c) Cache Size 10%.

Figure 5.16: Effect of Training Data Size

5.5.4 Effect of Network Reorganization

In the last set of experiments, we evaluated the adaptive behavior of PeerOLAP. We employed a network of 100 peers and we set the cache size of each to 1% of the data cube. The query optimization strategy was LQP and the caching policy was ICP. We used the same query set as in Section 5.5.2 (i.e., 10 groups with 10 peers each; every group accessing a different hot region). The maximum number of hops was set to 5. The period T_{reorg} that a peer reorganized its neighbors was set to 40 (i.e., each time it had made 40 queries).

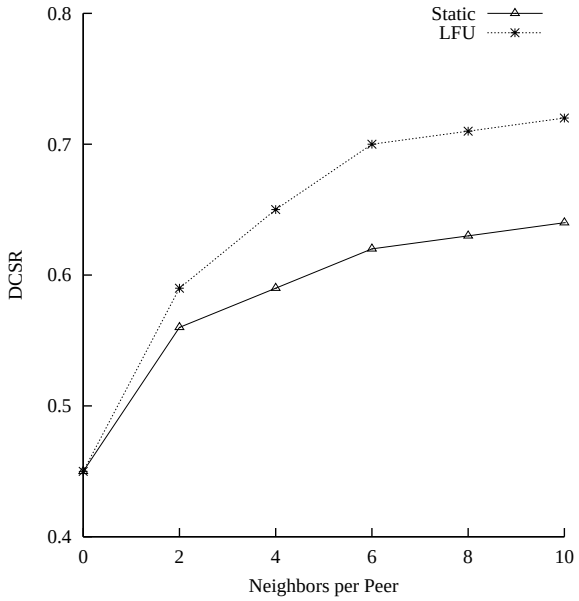


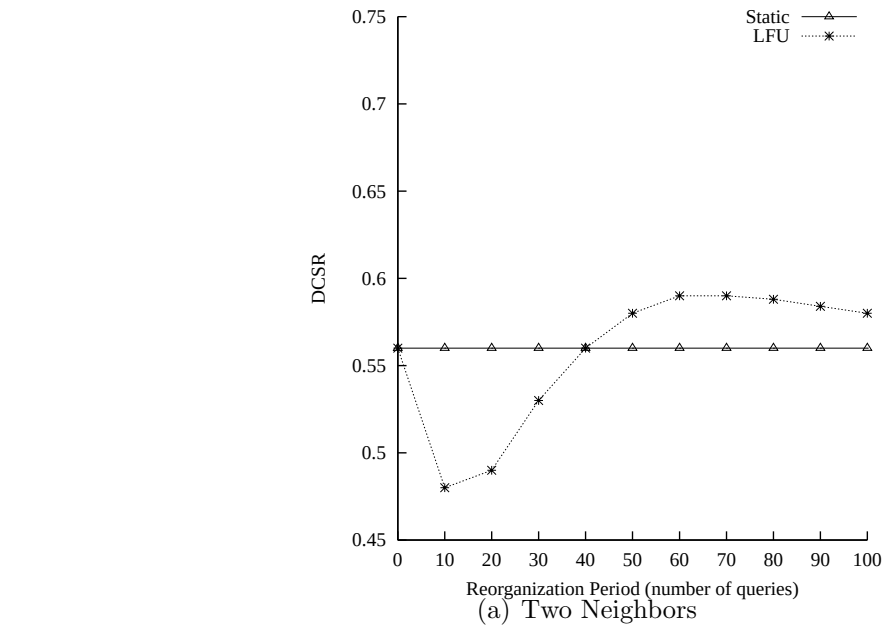
Figure 5.17: Effect of Network Reorganization

In Figure 5.17, we vary the number of neighbors per peer and compare our adaptive strategy against a static network. As the number of neighbors increases, the performance of the static system improves, because of the better knowledge about the contents of other peers. By rearranging the neighbors of a peer P, there are two possible benefits: (i) the cost of searching for chunks decreases because some distant

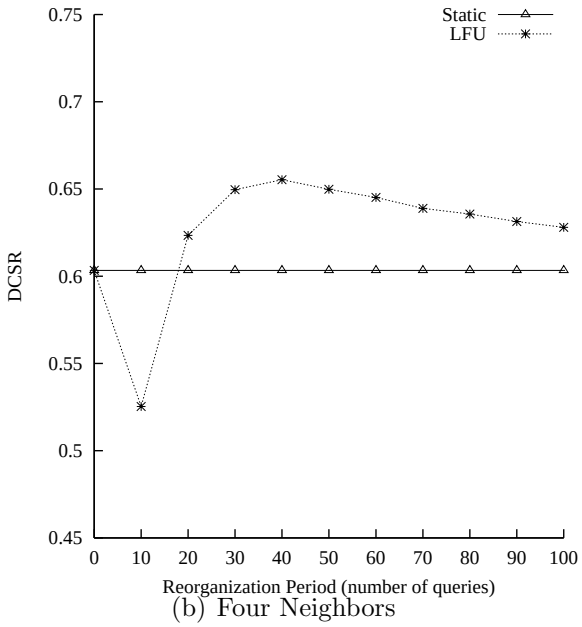
beneficial relevant nodes are becoming direct neighbors, and (ii) with high probability, the neighbors of a beneficial peer are also beneficial to P; therefore, larger groups are constructed incrementally.

In Figure 5.18, we set the number of neighbors per peer to two, four and 10, and we vary the reorganization period T_{reorg} from zero to 100. Consider Figure 5.18(b), when $T_{reorg} = 0$, the network is static. When T_{reorg} becomes 10, the performance drops significantly. This is due to the fact that there has not been enough time to gather accurate statistics; the initial network structure happened to be quite beneficial and the new structure is worse. However, if we allow the system to collect more information, the resulting network structure will be better and the performance increases. Observe that for values of T_{reorg} greater than 40, DCSR drops again slowly. The reason now is different: reorganization is performed so infrequently that it cannot follow the changes of the workload. In the extreme case, if T_{reorg} approaches infinity (practically if it is larger than the number of queries), the network becomes identical to static again.

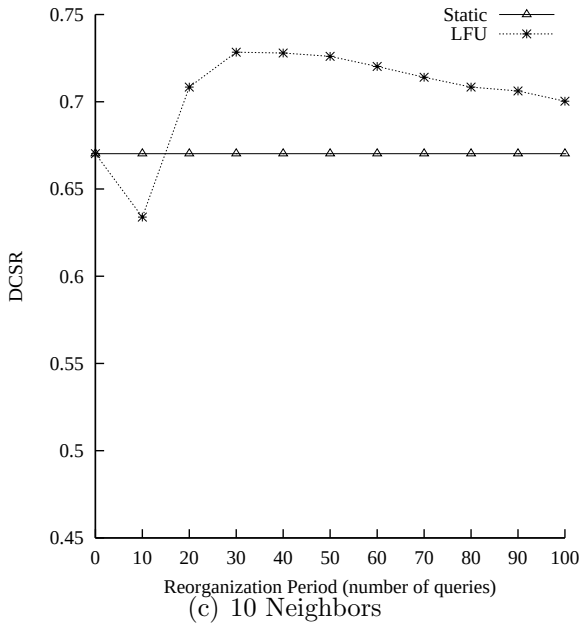
Notice from Figure 5.18(a), Figure 5.18(b) and Figure 5.18(c), that after the first performance drops, by extending the reorganization period, at a certain point, the reorganization approach becomes better than the static approach; we call the first frequency that makes such a success the performance horizon (PH). The PH is a reorganization period where the reorganization approach performs better than the static approach after the first performance drops with the number of neighbors per peer as its constraint. Figure 5.19 presents the PH versus number of neighbors per peer. When the number of neighbors per peer is small (e.g., two neighbor peers), theoretically, the dropping of any existing neighbor and subsequently the connection



(a) Two Neighbors



(b) Four Neighbors



(c) 10 Neighbors

Figure 5.18: Frequency of Network Reorganization

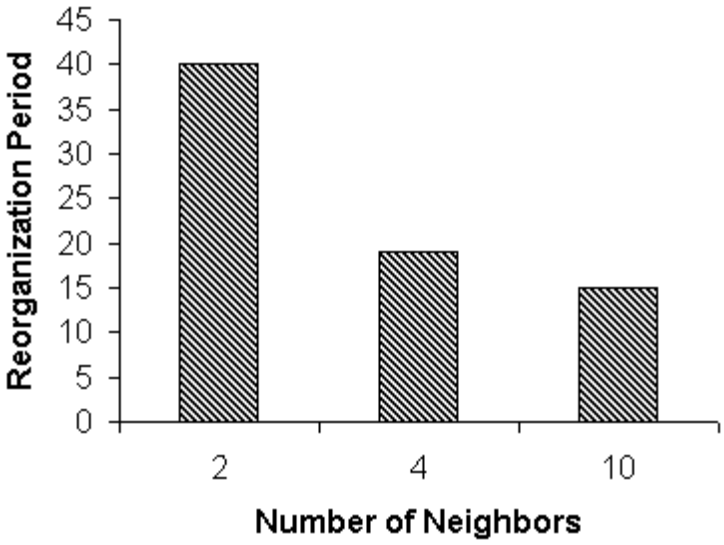


Figure 5.19: Performance Horizon of Two, Four and 10 Neighbors

to a new peer will affect the system significantly. Therefore, it requires a longer period in order to make a good decision on who the best peers are. In contrast, a peer with a large number of neighbors is more consistent in its performance even when the environment is in a high reorganization period.

5.6 Summary

In this chapter, we have presented PeerOLAP, a distributed caching system for OLAP results. In a typical client-server architecture, isolated remote clients access data warehouses and maintain previous results in their local caches. By sharing the contents of the individual caches, PeerOLAP constructs a large virtual cache which can benefit all peers. The system is fully distributed and highly scalable as there is no centralized administration point and no central catalogue. The network does not have any

specific structure, and peer participation does not have to be predictable.

As shown in the experimental evaluation, PeerOLAP achieves significant performance gains when compared to traditional systems. This is accomplished by (i) query optimization techniques that determine which chunks should be requested form the warehouse, and which should be retrieved from the peers; (ii) caching policies that enable cooperation among caches and eliminate unnecessary replication of objects; and (iii) re-configuration mechanisms that create virtual neighbors of peers with similar access patterns.

Chapter 6

FuzzyPeer: Answering Similarity Queries in P2P Networks

6.1 Introduction

In this chapter, we will focus on data acquisition problems, especially on retrieving information from P2P networks without limiting itself to only exact key matching queries.

Peer-to-Peer (P2P) technology has recently attracted a lot of attention since it allows the implementation of large distributed repositories of digital information. In a P2P system, numerous nodes of equal roles are connected through an arbitrary network and exchange data or services directly with each other. Many P2P systems follow a semi-centralized (or hybrid [25]) architecture (e.g., Napster [15]), where queries are posed to a centralized index, although the data and services are distributed. Despite their advantages, hybrid systems suffer from several drawbacks: (i) they cannot follow high-frequency changes in the source data, (ii) they require expensive dedicated infrastructure (i.e., high-end server farms, fast network connections, etc.), (iii) they exhibit a single point of failure, and (iv) legal reasons may prevent the accumulation

of information at a central location (e.g., the case of Napster).

As an alternative, several fully distributed (or pure P2P) systems have been proposed. In these systems, there are no centralized catalogues or functionalities; instead, peers are individually contacted and return the results they contain. There are two major sub-categories of pure P2P systems: (i) Hash-based systems (e.g., Chord [100], CAN [92] and Pastry[31]), which assign a unique key to each file and forward queries to specific nodes based on a hash function. Although they guarantee the location of content within a bounded number of hops, they require tight control of the data placement and topology of the network. (ii) Broadcast-based systems (e.g., Gnutella[42]), which use message-flooding to propagate queries. There is no specific destination; hence every neighbor peer is contacted and forwards the message to its own neighbors until the message lifetime expires. Such systems have been successfully employed in practice to form large-scale ad hoc networks. Here, we assume a pure P2P, broadcast-based architecture.

Most existing systems support only boolean query evaluation. Each file is characterized by its meta-data (i.e., a set of keywords), and queries ask for combinations of keywords. Consider, for instance, a music sharing system. Users ask for a song title, or a combination of an artist and an album name. Such queries can be unambiguously evaluated as "found" or "not found" by searching the meta-data for matching keywords.

In this chapter, we investigate a different problem: Users ask fuzzy queries like: "Find the top-k images which are similar to a given sample." Such queries are common in image retrieval systems (e.g., QBIC [79]) because it is difficult for humans to express precisely an image's content in keywords. Since there is no centralized index, each

peer within the query horizon is contacted and returns k results (i.e., the top-k local images) to the initiator, which, in turn, computes the global result. Unfortunately, the extremely low selectivity of such queries floods the network with useless messages. An alternative solution is to set a threshold similarity and accept answers only above this value. The issue in this case is how to select the query-dependant threshold, given that the interpretation of an image depends on the user's perception; if the threshold is too low, there is no benefit in terms of transmitted messages while, if it is too high, there is the risk of wasting the query messages without locating any satisfactory answer. Moreover, this method would not reduce the number of query messages which grows exponentially with the number of hops.

Observe, however, that due to the fuzzy nature of the queries, the answers are always approximations. As a result, if two queries are similar, the top-k answers for the first one may contain (with high probability) some of the answers for the second query. In addition, in P2P networks, each peer can examine the messages that pass through it. Motivated by these observations, we developed FuzzyPeer, a generic P2P system that supports similarity queries. In FuzzyPeer, some of the queries are stopped (i.e., they are not propagated further) and stay resident inside a set of peers. We use the term frozen for such queries. The frozen queries are answered by the stream of results that passes through the peers and which have been initiated by the remaining running queries. By carefully selecting the set of queries that will be answered by the streaming data, the quality of the results and the response time remain at acceptable levels even when the system is overloaded. Additionally, the number of messages drops considerably, thus improving the scalability and the throughput of the network. Moreover, our optimization algorithms are fully distributed and pose no additional

overhead due to synchronization messages.

Although throughout this paper we focus on image retrieval, our methods are applicable to other domains where similarity search is performed in P2P networks. As an example, consider the case of text retrieval for documents that reside in autonomous interconnected workstations, where the network topology may not be flat. Given a two-level super-peer organization (e.g., Morpheus [74]), we can apply the same techniques at the upper level which contains the index of its clients, rendering the entire system more scalable.

6.2 System Description

The FuzzyPeer system consists of a set of peers which are connected through an ad hoc unstructured network and implement a mechanism for searching their data. In contrast with other P2P systems which characterize every shared object only by a set of keywords, FuzzyPeer allows content-based fuzzy queries. We will describe the functionalities of our system through a case study of an image retrieval application.

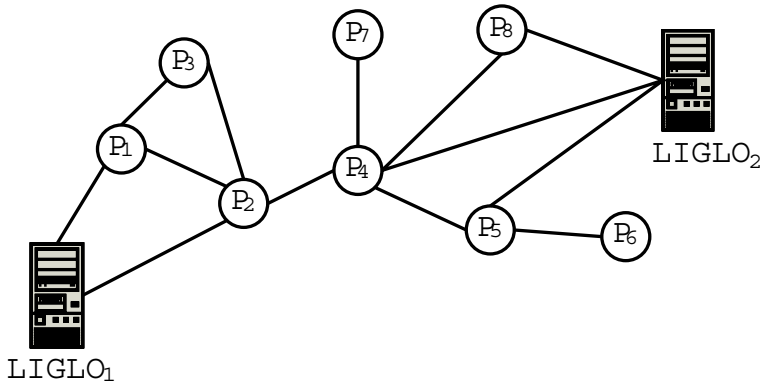


Figure 6.1: A Typical FuzzyPeer Network

Figure 6.1 depicts a typical FuzzyPeer network. It consists of eight peers which are connected through a set of links. Each link represents an active TCP/IP connection and is independent of the structure of the physical network layer. Connections are symmetrical, meaning that if a peer P_i is a neighbor of P_j, then P_j is also a neighbor of P_i. Ideally, each peer P should be connected to all others since this would maximize its search space. However, there is a trade-off because connections consume system resources and also cause more messages to be processed by P. Since the system is heterogeneous both in terms of bandwidth and processing power, powerful peers with fast links are typically connected to more neighbors. This is common in most P2P systems. For example, Gnutella implementations allow up to four neighbors for peers with slow connections, while powerful peers may have tens of neighbors.

Participation in the network is dynamic; each peer can join or leave the system at any time. When a peer enters the network, it contacts a location-independent global name lookup (LIGLO) server [77] to get a set of potential neighbors; then it employs a Gnutella-like protocol [42] to connect. Except for the LIGLO servers, the system is fully distributed. Furthermore, the LIGLO servers are not involved in the query processing and can be completely eliminated if the set of initial neighbors can be otherwise determined; for instance, peers on the same segment of a LAN may connect to each other.

Let the user of P1 ask a query q: "Find the top 10 images which are similar to a given sketch." P_1 will broadcast q to P_2 and P_3. The receiving nodes will search their databases and return the IDs of the top 10 most similar images together with a similarity measure to P_1. At the same time, they will broadcast q to their neighbors. For example, P_2 will send q to P_3 and P_4. P_3 notices the duplicate message and

rejects it. P_4, on the other hand, computes the local result and returns it to P_2 which, in turn, will send the result back to P_1. P_4 also propagates q to P_5, P_7 and P_8. All the results will be returned to P_1 via P_4 and P_2. Queries can propagate for up to a maximum number of hops d. Assuming that $d = 3$, P_5 will not propagate the query any further; therefore P_1 cannot reach P_6. P_1 waits for up to $MaxWaitTime$; during this interval, it receives the answers and continuously refines the result. After $MaxWaitTime$ expires, any answer message that reaches P_1 is rejected.

Assume now that soon after P_1, P_3 also submits a query $q\prime$ which is similar to q. In a traditional P2P system, $q\prime$ propagates through P_2 the same way as q. $q\prime$ causes many messages to pass through P_2 almost simultaneously with the messages generated by q. Therefore, P_2 is overloaded and all messages are delayed. If the delay is long enough, $MaxWaitTime$ expires, causing q and $q\prime$ to terminate before they receive enough useful results. Notice, however, that we can do better: When q passes through P_2, it initiates an answer stream $Stream_q$. All the answers from P_4, P_5, P_7 and P_8 will go through $Stream_q$. When $q\prime$ reaches P_2, the system can identify that q and $q\prime$ are similar, so instead of being propagated, $q\prime$ will freeze inside P_2 and will be attached to $Stream_q$. P_2 will afterwards duplicate and send to P_3 all answers that reach $Stream_q$. The intuition is that since q and $q\prime$ are similar, their answers are also similar, and it is preferable to get some approximate answer than not getting any answer at all. The rest of the paper presents the freezing technique in detail.

6.2.1 Prototype Implementation

The basic components of a FuzzyPeer node are depicted in Figure 6.2; the low-level network functionalities, such as connecting to other nodes, message handling, etc., are

provided by BestPeer [77]. This is a JAVA-based generic platform which simplifies the development of P2P systems. A part of BestPeer provides the LIGLO functionality. All the FuzzyPeer-specific code resides in the Query Processing Applet. This module coordinates the entire system and connects to the local database. The details of the database are irrelevant to FuzzyPeer. The only requirement from the database is the support of a top-k operator, although it is also desirable to provide a cost estimation function. In our case study, the database consisted of flat files of original high-resolution images, together with pre-calculated feature vectors. We used Daubechies wavelets [107] to represent the visual features of the images. In our experiments, the number of images in each peer was relatively small, allowing us to keep all the feature vectors in the memory and find the top-k queries by performing sequential search. In practical situations, where the image database is expected to be larger, we can employ a high-dimensional index for k-nearest-neighbor search like [113].

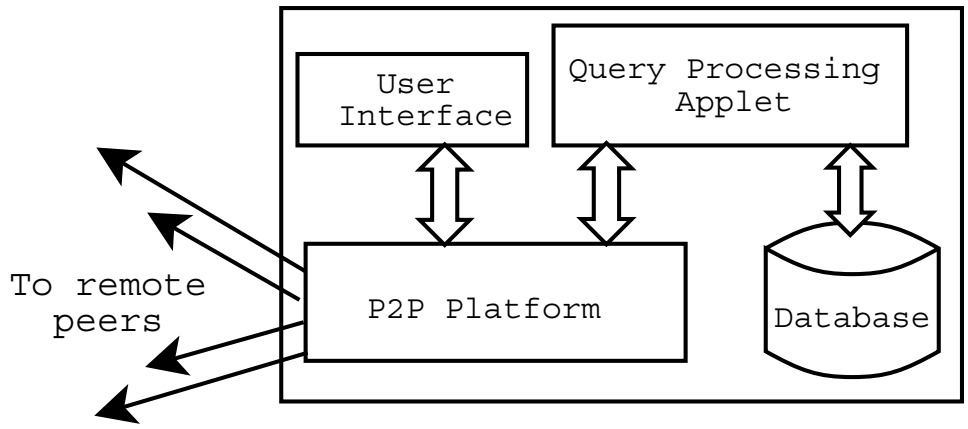


Figure 6.2: Peer Components

Note that for the rest of the work, we shall only consider the optimization of the

search process. We assume that the downloading of the original image or a thumbnail is performed outside the search network (i.e., as a separate http connection).

6.3 Query Processing

In this section, we analyze our freezing techniques. Although we focus on the image retrieval application and we employ the Euclidian distance as a similarity metric, the translation to other domains is straightforward.

Users pose queries by means of a sample image img_{user}. We apply the Daubechies wavelet transformation $DT(img_{user})$ on the sample image and produce an m-D feature vector $f_1, ..., f_m$. This vector is the query q. The size of q is typically much smaller than the size of the image. The similarity $S(q, img)$ between a query q and an image img is defined as the Euclidian distance between the vector q and $DT(img)$. In the same way, we define the similarity $S(q, q')$ between two queries q and q' as the Euclidian distance of the corresponding vectors. Obviously, when $S(\cdot) = 0$, the vectors are identical.

When a peer receives a query q, it computes the k most similar images $img_1, ..., img_k$ from the local database, and returns a set of k pairs $(id_i, S(q, img_i))$, $i = 1..k$. Note that the answers do not contain the feature vectors of the images, but only the image IDs and the similarity measures. This is done in order to reduce the size of the answer messages, since in some applications, each feature vector may be several KBytes.

In a traditional broadcast-based P2P system, when a query q is initiated at peer P, it is propagated through all its neighbors until a maximum number of hops d is reached. The peers that receives the query, send their answers back to P via the reverse path. We call this the $Non - Freezing$ algorithm (nf).

When a query q is propagated through a peer P, it creates an answer stream $Stream_q$. All the subsequent answers for q will go through $Stream_q$. If P is the peer which initiated q, the answers that arrive in $Stream_q$ are forwarded to the User Interface module; else they are propagated to the previous peer (i.e., the peer from which the query arrived). Therefore, an answer is propagated to at most one peer. Obviously, there can be multiple streams simultaneously active at P. For every stream, P maintains a data structure containing the feature vector q together with several statistics.

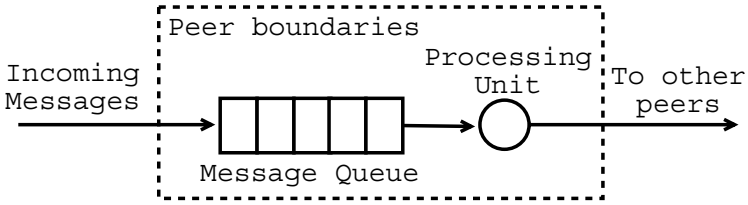


Figure 6.3: Message Propagation Model

There are several sources of delays in the path of a message, including the network cost and the processing time. To facilitate our study, we use a simplified model (Figure 6.3): Each peer P has a processing unit with a FIFO queue attached to it. All incoming messages M_0, M_1, M_2,... enter the queue. When the processing unit is ready, it removes the message M_0 from the head of the queue and processes it for time $T(M_0,P)$. After processing, the message is transmitted to the next peer. The total time a message M_i spends at P is:

$$T_{total}\left(M_i, P\right) = \sum_{j=0}^{i-1} T\left(M_j, P\right) + T\left(M_i, P\right) \tag{6.3.1}$$

where the first factor of the equation is the waiting time in the queue and the second factor is the actual processing time.

For a given number of online peers, assume that the query rate is low. Then the queues in all peers are empty, and from Equation 6.3.1, it follows that the only delay of a query message is its own processing time. However, when the query rate increases, the message queues become longer; therefore the delays for the messages also increase due to the queue waiting time. Recall that users abort queries after $MaxWaitTime$. If the delays are long, there is not enough time for the query messages to reach many nodes before $MaxWaitTime$ expires. The number of answers that arrive at the initiating node decreases rapidly; therefore, the probability of obtaining useful answers (i.e., the precision of the results) also drops. This resembles the thrashing effect in time-sharing systems. The freezing algorithm that we describe below, attempts to minimize the problem by decreasing the number of concurrent messages in the system.

6.3.1 Static Query Freezing (SQF)

The intuition behind the Static Query Freezing (SQF) algorithm is simple: some queries are *frozen* (i.e., paused) inside the system in order to reduce the total workload. The result is that the waiting time in the queues decreases for the remaining running queries, so they can retrieve enough answers before $MaxWaitTime$ expires. The frozen queries attach to the streams of similar running queries and receive the same results. There are several benefits in this approach: (i) Thrashing is avoided (if enough queries are frozen). Instead of not answering any query at all, with SQF, a considerable percentage of the queries can locate accurate answers. (ii) Excess queries are frozen instead of aborted. Since all answers are approximations, there is a high probability for a frozen query to receive accurate results if it attaches to a similar stream. This is different from other systems (e.g., web search engines) where the

probability of finding a concurrent similar query is low. In such systems, queries ask for certain keywords and run in the server for a few msec, while in P2P systems, queries run for around three orders of magnitude more time (i.e., 100's of sec). (iii) Even if the results for the frozen queries are not accurate, users can utilize them to refine their original query.

When a query q is propagated through a peer P, it initializes an answer stream $Stream_q$. All the subsequent answers for q will go through $Stream_q$. For every stream that is active at P, we maintain a data structure containing the feature vector q together with several statistics. The streams are used by the freezing algorithms.

Figure 6.4 presents the SQF algorithm which takes two parameters: the probability p_f to freeze a query and the number of hops f_{hops} that q must travel before it freezes. The initiator peer decides with probability p_f if the new query is going to freeze. Then q is propagated as usual. Assume that q has been selected to freeze, and after f_{hops}, reaches peer P'. SQF pauses q (i.e., q will not propagate further through that path), checks all the active streams at P' and attaches q to the most beneficial one. If no stream exists at P', q is just paused. Observe that q will freeze in all peers which are f_{hops} hops away from P. Also notice that the Non-Freezing algorithm (nf) is a special case of SQF where $p_f = 0$.

When an answer comes for a stream, SQF searches whether there are any attached queries to it. For every attached query q, a duplicate answer is generated and returned to the query initiator. Notice that since the answer messages do not contain any feature vectors, we cannot perform any filtering for the attached queries.

The freezing technique has the additional benefit of increasing the query horizon of the frozen queries. Consider again the example of Figure 6.1, assuming that there

```
1.  On UserQuery(q)
2.      With probability p_f set q.frozen = true, q.f_hops = h_f
3.      Initiate an answer stream A_q
4.      Broadcast q
5.
6.  On QueryReceived (q) // query received from neighbor
7.      IF q.frozen==true AND q.f_hops == q.traveled_hops THEN
8.          Freeze q
9.          Attach q to a beneficial answer stream, if such stream exists
10.     else
11.         IF q.frozen == false THEN calculate answer and send it back
12.         IF q.traveled_hops < max_Hops THEN broadcast q
13.
14. On ResultReceived(r_q) // result received from neighbor
15.     FOR EACH query q' attached to q DO
16.         IF there is no cycle due to frozen queries THEN
17.             Change r_q to r_q'
18.             Propagate r_q' backwards as a result for q'
19.     IF current peer is the initiator of q THEN
20.         Add r_q to answer stream A_q
21.     ELSE propagate r_q backwards
```

Figure 6.4: Static Query Freezing Algorithm

is no link between P2 and P3. P1 initiates a query q which propagates to all nodes except P6 (recall that the maximum number of hops d = 3). P3 also initiates q' which, if not frozen, will reach only P1, P2 and P4. On the other hand, if q' freezes in P1 and attaches to q, the answers from P5, P7 and P8 will also be forwarded to P3. Therefore, the probability of locating an accurate result for q' increases.

The method of selecting a beneficial stream needs further clarification. Each stream *st* has a lifetime *lt* which is the same as the expiration time of the query that created it. A query queue will benefit by attaching to *st* only if there is enough time

left for many results to propagate through it. Therefore, recent streams are more beneficial. Also the benefit is proportional to the similarity of q with the query that initiated st. Notice that we can calculate this similarity, since we have the feature vectors for both queries. In our implementation, we use a combined benefit, giving more weight to the similarity criterion.

In the experimental section, we shall show that SQF improves significantly the throughput and the scalability of the system. Its applicability, however, is limited in practice since the user must provide for each query an appropriate set of parameters for the current condition of the network. Below, we describe an alternative freezing algorithm which adapts dynamically to the workload of the system.

6.3.2 Adaptive Query Freezing (AQF)

The drawback of SQF is the need to set accurately the parameters. If the freezing probability p_f is too low, the system will enter the thrashing region. On the other hand, if p_f is too high, there are not enough running queries; consequently, the probability of a frozen query to locate a similar answer stream will decrease and the precision of the results will drop.

p_f depends on two major factors : (i) The number $|Q|$ of concurrently active queries (i.e., those that are not yet aborted) in the system. Obviously, the load of the system is proportional to $|Q|$, therefore when there are more active queries, p_f must increase. $|Q|$ can be analyzed as $|Q| = Q_{us} \cdot |P|$, where Q_{us} is the number of queries per user per second and $|P|$ is the number of active peers (i.e., users). (ii) The topology of the network. P2P systems typically exhibit power-law topology. Some hub nodes (e.g., P4 in Figure 6.1) receive more messages and become the bottleneck,

even if the average load of the system is moderate. Since the system is dynamic with no centralized administration point, it is difficult to gather information about these factors. Notice, however, that the effect of varying $|Q|$ or altering the topology is that the waiting time in the queues changes.

```
1.  On UserQuery(q)
2.      Initiate an answer stream A_q
3.      Broadcast q
4.  On QueryReceived (q) // query received from neighbor
5.      q.traveled_hops++
6.      calculate answer and send it to the previous peer
7.      IF q.traveled_hops < max_Hops THEN
8.          IF checkFreezeCriteria(q) == true THEN
9.              q' is a running query which is similar to q
10.             // chekFreezeCriteria ensures that such q' exists
11.             Freeze q
12.             Attach q to q'
13.         ELSE broadcast q
```

Figure 6.5: Adaptive Query Freezing Algorithm

The Adaptive Query Freezing (AQF) algorithm controls the waiting time in the queues. Intuitively, if the waiting time is such that there is no time for a query to be forwarded or for the answer to come back before the query is aborted, there is no benefit from propagating the query message. Instead, there is a penalty, since the message will put additional load on the subsequent peers. In such cases, it is better to freeze the query in order to prevent thrashing. On the other hand, if the queues are short, it is beneficial to propagate the query in order to retrieve accurate results. Figure 6.5 presents the details of the algorithm. The OnResultReceived method is

the same as in SQF. In contrast to SQF, however, the initiator peer does not need to decide if the new query will freeze. The query is propagated inside the network and each receiving peer decides independently. Notice that AQF is general and versatile since it does not depend on any application-specific criterion.

For an incoming query message MQ at P, the checkFreezeCriteria() function returns true, if:

$$T_{total}(MQ, P) > a_q \cdot MaxWaitTime \tag{6.3.2}$$

where T_{total} is defined by Equation 6.3.1 and a_q is a system-wide parameter. The accurate evaluation of T_{total} is difficult. The reason is that $T(\cdot)$ is a function of the message type. For example, a query message needs more processing time than an answer, since the former requires an expensive search in the local database while the latter just needs to be propagated. Even if we have an accurate estimation for each message type, we still cannot evaluate T_{total}; the exact processing time will be known when all messages in the queue enter the processing unit, since some of them may get frozen.

$MaxWaitTime$ is also an estimation, since each user decides independently when to abort a query. In practice, we expect to get quite an accurate estimation with a small variance of this parameter by observing the behavior of users over a period of time (i.e., most users would wait for a couple of minutes before aborting and refining their queries).

The value of the parameter a_q should be such that it allows enough time for query processing and for answer messages to return to the initiator before the query is

aborted. Formally, a_q depends on the query path:

$$a_q(P_0, ..., P_i) = \frac{1}{MaxWaitTime} \cdot \sum_{j=0}^{i} T_{total}\left(MQ, P_j\right) \tag{6.3.3}$$

where P_0 is the initiating peer and P_i is the current peer. Nevertheless, this formula assumes that every peer has knowledge about the queue waiting time at the other peers in the query path, which is unrealistic. In practice, however, the exact value of a_q is not critical, as long as it prevents the queues from growing exponentially. By gathering statistics of the queue lengths over a period of time, a_q can be set as a system-wide constant. For our settings, we found that $a_q = 1$ provided the best results; however varying it for almost an order of magnitude did not affect considerably the performance, indicating the robustness of AQF.

6.3.3 Similarity Query Freezing (simQF)

AQF bases its decisions solely on the size of the message queues. It is a very general algorithm, and it does not employ any application-specific knowledge to further improve its performance. Since our queries ask for similar images, we developed the simQF freezing algorithm which uses query similarity as the freezing criterion. The algorithm is the same as AQF, except that a query q is frozen at a peer P if there is an answer stream whose distance to q is less than the threshold ρ.

Our experiments revealed that simQF produces good results if the threshold ρ is set correctly; else the behavior resembles the Static Query Freezing technique.

6.3.4 Multiple-feature Queries

One of the major challenges in multimedia retrieval systems is that the similarity between objects cannot be defined precisely with a simple description. For instance, the image of an orange can be best described by a combination of color feature: '*similar to yellowish*', shape: '*round*', and possible content text description: '*fruit or orange*'. Figure 6.6 illustrates the idea of combining multiple features in a graphical manner. Each point in the figure represents an image in the existing feature space. The feature space is separated into three well-defined clusters: color (yellowish), shape (round) and text-description (fruit). A user query distribution might be best described in the combination of color, shape and text-description. However, with these existing features representation, it might not be easy to model the query distribution. Alternatively, a query can be divided into multiple independent sub-queries, each targeting a specific feature cluster. A complex multimedia query system that combines multiple atomic sub-queries is called a multiple-feature query system [34, 35].

FuzzyPeer supports multiple-feature queries and integrates them into the query freezing framework. Assume that an object is characterized by n feature vectors $v_1, ..., v_n$. Then there are $2^n - 1$ possible query types for every combination of features. We support two methods for processing such queries: serial and random.

Multiple-attribute Random processing (maR)

This algorithm is inspired by Fagin's Algorithm (FA) [34]. FA computes multiple-feature queries at a middleware by combining sequential access to single-feature sorted lists with random access to the original data provider. In our case, although we cannot have sorted streams, we can access a remote peer directly.

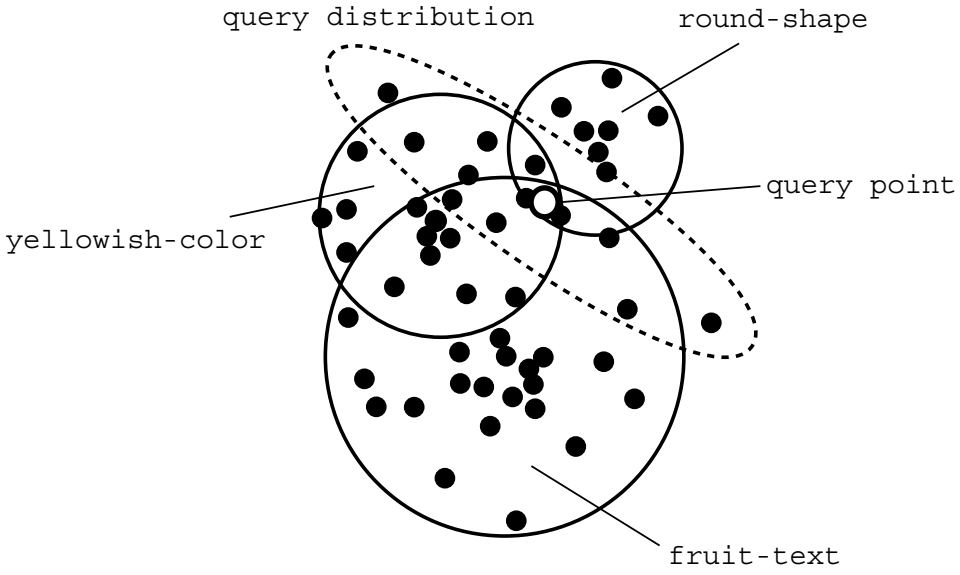


Figure 6.6: Query Distribution across Multiple Feature Clusters

Queries that have some common subset of features are considered compatible. For example the query "*Color is yellowish and Shape is round*" is compatible with "*Color = orange and Type = fruit*" because they contain the same feature "Color". The algorithm works in two phases. In the first phase, it runs exactly like AQF with the additional characteristic that compatible queries can attach to each other. Because of this, some of the results that arrive from frozen queries are not complete (i.e., they answer only some of the features). In the second phase, the algorithm sorts the incomplete results and selects the top-k according to the similarity metric. For these k objects, it performs direct access to the remote peers that contain them, and receives the complete answers. As we shall show in the experimental section, this algorithm is beneficial when the data is clustered. In case of random data, the performance deteriorates due to the remote connection overhead.

Multiple-attribute Sequential processing (maS)

In contrast to the previous method, this algorithm considers a different query type as incompatible, even if they share some features. Therefore, it never attaches a query to an answer stream of a different type. The result is that there are no incomplete answers so there is no need for a second phase. Since there is no remote connection overhead, this algorithm is more suitable for uniform datasets.

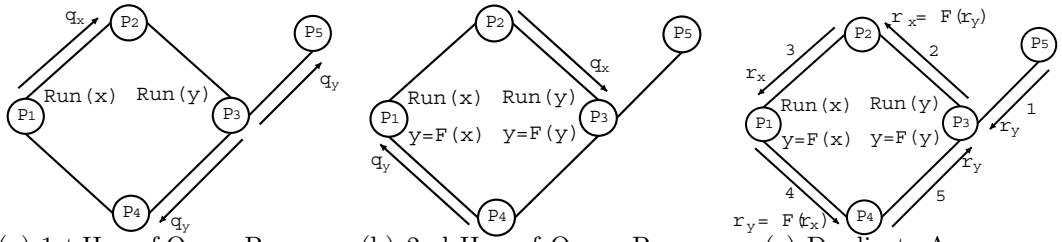
6.3.5 Dealing with Cycles

Connections among peers are arbitrary, resulting in network graphs that contain cycles. The existence of cycles generates unnecessary messages both during query propagation and in the process of answering frozen queries. The first case is easier to manage: If the cycles are longer than the maximum number of query hops d, there is no overhead. In the other case, if a peer P_i receives a query message q that has passed before [1], it simply drops q. The overhead is one extra message per cycle.

The effect of cycles on frozen queries is more complex. To illustrate this, assume the network topology of Figure 6.7(a), where P_1 initiates query q_x and P_3 initiates q_y almost simultaneously. Let q_x reach P_3 faster through P_2, and q_y reach P_1 through P_4, as shown in Figure 6.7(b) (the exact route is not important). P_3 realizes that q_x is similar to q_y, which is already running, so it freezes q_x and attaches it to q_y. In the same way, P_1 freezes q_y since it is similar to q_x. Now, assume that P_3 receives a result r_y from P_5 that answers q_y. Since q_x is attached to q_y, P_3 labels the result as $r_x = F(r_y)$ and propagates it to P_1. There, r_x answers q_x that has q_y attached, so it

[1] Peers maintain a list of the IDs of messages that have recently passed through them. Each message has a unique ID, consisting of the IP address of the initiating peer and a unique local key.

changes again the label to $r_y = F(r_x)$ and sends it to P_3 (Figure 6.7(c)). Obviously, P_3 detects the duplicate answer and rejects it. This kind of cycle, however, creates considerable overhead; up to $O\left((c-1) \cdot N^d\right)$ unnecessary messages are propagated, where c is the length of the cycle and N is the maximum degree of the nodes that receive q_y.



(a) 1st Hop of Query Propagation (b) 2nd Hop of Query Propagation (c) Duplicate Answers

Figure 6.7: Cycles due to Frozen Queries

In order to break such cycles, we append each answer with information about the transformations that have taken place in the return path. In our example, r_x carries a tag indicating that its original label was r_y; therefore, P_1 will not use r_x to answer q_y. Notice that there may exist cycles with up to l transformations, where l is not bounded by d. Nevertheless, in practice l is expected to be moderate; long cycles are rare, because the original queries expire, causing their attached queries to expire also, before the messages manage to travel all the hops around the cycle. In our experiments, for example, queries expire within 60sec in networks with 1000 nodes; the resulting cycles contain at most three transformations.

More sophisticated solutions are also possible. For instance, the system can implement a cycle avoidance algorithm (refer to [6] for a survey) when propagating the queries, in order to prevent the forming of cycles. Alternatively, we could run a cycle

Table 6.1: Parameters Derived from the Prototype

Parameter	Value	Comments
TR_R	3.69 KB/sec	Average transfer rate between remote peers (WAN)
TR_L	594.93 KB/sec	Average transfer rate between local peers (LAN)
APT_O	1.19 sec/mes	Average processing time for query message
APT_M	0.01 sec/mes	Average processing time for other types of query messages
ICT_R	3.68 sec/con	Average time to initiate a remote connection (WAN)
ICT_L	0.36 sec/con	Average time to initiate a local connection (LAN)

detection and recovery algorithm to un-freeze some of the queries after a cycle is formed. However, both methods would add complexity to the system and introduce overhead due to control messages, while the simple solution that we have described above works acceptably well.

6.4 Experimental Evaluation

We employed two implementations to evaluate our methods. The first one is a JAVA prototype based on our BP platform which run on Pentium III PCs with 256MB RAM and Windows 2000; it was used to derive the basic parameters of the system (see Table 6.1). LAN denotes that a pair of nodes was physically connected to a 10Mbps local network, while WAN means that one node was in Singapore and the other in Hong Kong. The parameters were used in the second implementation, which was a simulator running on a 2-CPU Ultra-SRARC III server with 4GB RAM. We employed a simulator since it would be impractical to set up large networks, while the benefits of our methods could only become significant when there are many participating nodes.

Since there is no similar system in use, we did not have any information about the network topology or user behavior. As an approximation, we adopted the parameters of existing broadcast-based content-sharing systems, which are presented in [110]. We generated two network topologies for the simulation:

1. $Uniform$, where the average number of neighbors per node is 3.2.

2. $PowerLaw$ [36], which is a for simulating the behavior of the Internet.

We employed the PLOD algorithm [16] and set $\alpha \in [0.85, 0.99]$, $\beta \in [96, 355]$, resulting in 3.2 neighbors per node on average. The nodes were connected either through a slow (WAN) or a fast (LAN) line to the network. A node which was connected through a slow link could support up to four concurrent neighbors (this is the default value in Gnutella). The number of nodes which were simultaneously online varied from 100 to 1000. Since in practice, only around 5% of the users are active at any given time [109], our results are representative for populations of up to 20,000 users.

We used three image datasets in our experiments. The first one, REAL48, consisted of a library of 10,504 high resolution images. We used wavelet coefficients to represent their visual features. In order to achieve a uniform spacing of colors, we first transformed the images from the original RGB to the CIE L*U*V color space. Then, we rescaled each image to 128 $\times$ 128 pixels and we calculated a five-level wavelet decomposition by using the Daubechies wavelet transformation [107], resulting in 16 sub-bands. We computed the average and the standard deviation for each sub-band and got 16 (μ, σ) pairs. These 2·16 coefficients, together with the upper left 4 $\times$ 4 corner of the transformation matrix, formed a 48-D vector $F = f_1, f_2, ..., f_{48}$

which encoded the visual features of the image. To ensure that each dimension of the feature vector received equal emphasis, we computed the normalized vector $F' = \{f'_1, f'_2, ..., f'_{48}\}$ as:

$$f'_i = \frac{f_i - \mu_i}{\sigma_i}$$

where μ and σ are the mean and the standard deviation of the ith dimension of the entire collection's feature vectors. The normalization was performed to express the similarity between two images by the Euclidian distance of their feature vectors.

The second dataset, $REAL_{191}$, consisted of the same set of images, but for each one, we extracted two feature vectors. The first was a 32-D vector with the 16 (μ, σ) pairs of the Daubehies transformation. The second vector had 159 dimensions and encoded information about the texture. Our third dataset was $SYNTH_{200}$. It consisted of 10,000 pairs of feature vectors, one with 32 and the other with 168 dimensions, and was generated synthetically. There were 100 clusters of vectors around 100 random points. The vectors inside each cluster followed a Gaussian distribution, where $\sigma = 0.2$.

6.4.1 Static Query Freezing

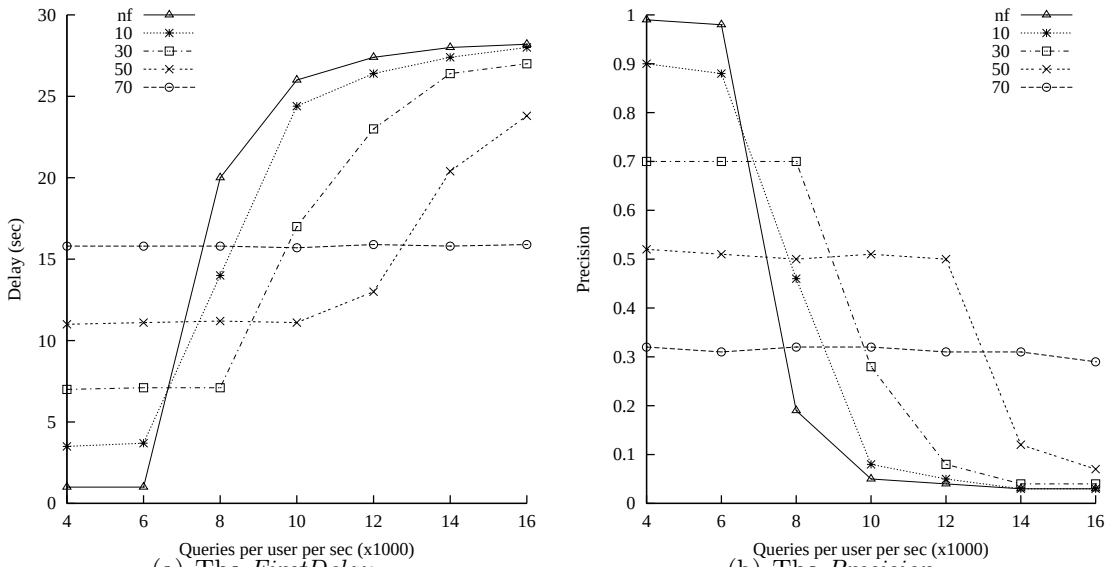
In the first set of experiments, we compared a broadcast-based system that did not support frozen queries (denoted as nf) against our static freezing algorithm. There were 100 peers simultaneously online, and for each setting, we calculated the average results over 1000 queries; these were images selected from our dataset with uniform distribution. The peer which initiated each query was also randomly chosen. For every possible query, we pre-calculated the *Global-top-k* which was the set of the top-k images from the entire dataset. k was fixed to 10 for all the experiments in this

section. The performance results of our algorithms are presented in Figure 6.8, 6.9 and 6.10. In the left side of each figure row, we draw the $FirstDelay$ which is the average delay in seconds in obtaining the first image which belongs to the *Global-top-k*. Intuitively, this measure indicates how long the user must wait until the arrival of the first useful result. Users may wait for up to $MaxWaitTime$ seconds before they abort the query. If no useful result has arrived during this interval, $FirstDelay$ is set to $MaxWaitTime$. The right side of each Figure 6.8, 6.9 and 6.10 presents the average precision of the results. $Precision$ is defined as the number of obtained images that belong to *Global-top-k*, divided by k. Notice that this metric does not consider the rank of each image. We also computed a $WeightedPrecision$ metric by assigning the highest weight to the *Global-top-1* image, and decreasing it linearly for the subsequent ranks. Here, we present only the *Precision* metric; the trend for the $WeighedPrecision$ was the same.

second row the *Precision* as a function of Q_{us} (Queries per user per second).

Q_{us} is the number of queries that each user initiates per second. It follows a Gaussian distribution, where the mean $\mu \in [4 \cdot 10^{-3}, 16 \cdot 10^{-3}]$, the standard deviation $\sigma = 5\% \cdot \mu$, and values are restricted in the interval $\mu \pm 3\sigma$. The x-axis in the graphs represents the mean value of Q_{us}; the maximum and minimum values correspond to one query per user every 63 and 250sec, respectively. For the static freezing algorithms 10, 30, 50 and 70% of the queries are selected randomly to freeze one hop away from their origins.

In Figure 6.8, we consider a power-law network where $MaxWaitTime$ is 30sec and allow each query to propagate for up to seven hops. When Q_{us} is low, the best results both in terms of $FirstDelay$ and $Precision$, are achieved if there are no frozen queries



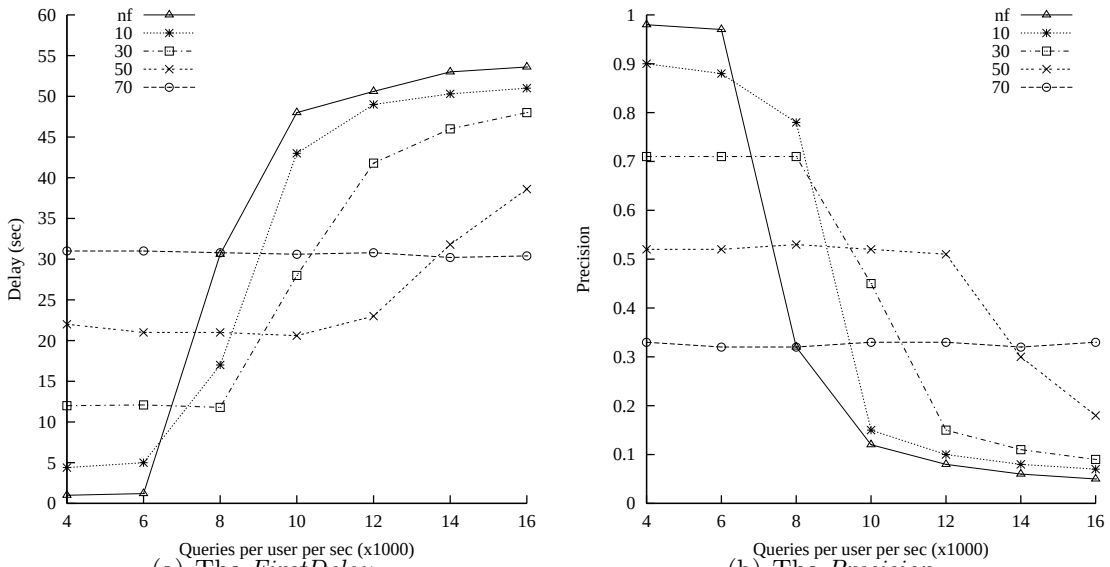
(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.8: Non-frozen(nf) vs. 10, 30, 50, 70% Statically Frozen Queries. MaxWait-Time = 30sec, Power Law Network.

(nf). This is due to the low number of queries that are propagated simultaneously inside the network. When there is an attempt to freeze a query q at a peer P, it is possible that there is no other query q' running at P at the same time, so there will be no results for q via that path. Even if there is a q' available, there is only a low probability that q and q' are similar enough, so most of the results will be useless.

While a low query rate does not benefit the freezing algorithms, it is definitely desirable in P2P systems. The load of each peer is kept low and the links among nodes are not congested. Therefore q is propagated fast, and there is enough time to exploit a large part of the network before $MaxWaitTime$ expires. This fact explains the good results achieved by the traditional method (nf).

When the query rate Q_{us} increases, however, the performance of nf deteriorates rapidly. This is due to the overloading of peers and network connections. Messages



(a) The *FirstDelay*. (b) The *Precision*.

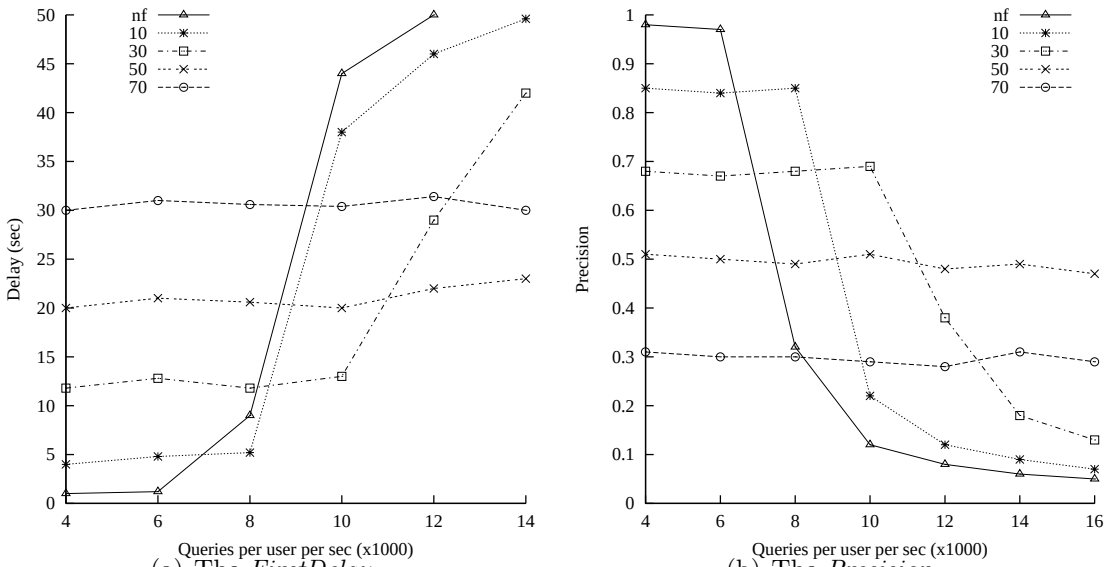
Figure 6.9: Non-frozen(nf) vs. 10, 30, 50, 70% Statically Frozen Queries. MaxWait-Time = 60sec, Power Law Network.

take longer to propagate; therefore $FirstDelay$ increases. Moreover, since there is not enough time to contact many nodes before the queries expire, $Precision$ decreases. On the other hand, by freezing 10% of the queries, the number of concurrent messages inside the network decreases. Thus, although both $FirstDelay$ and $Precision$ deteriorate, this occurs at a slower rate than the nf case. The result is that for large values of Q_{us} (greater than $8 \cdot 10^{-3}$ for this setting), the $Frozen_{10\%}$ case performs better than nf.

The performance can be further improved by freezing more queries. For example, freezing 70% of the queries produces better results than the 10% case, for $Q_{us} \geq 10_{-2}$. The tradeoff is that for smaller values of Q_{us} the $Frozen_{70\%}$ case performs considerably worse. Summarizing, in order to achieve good results, the number of frozen queries should increase when Q_{us} increases, and vice versa.

Figure 6.9 depicts the results for the same settings except that $MaxWaitTime$ is fixed to 60sec. Note that the $FirstDelay$ measure is not comparable for different values of $MaxWaitTime$ because of the way it is calculated (i.e., it is set to $MaxWaitTime$ if a query does not return any useful result). For $Precision$, on the other hand, the comparison is meaningful. Observe that while the trend is the same as in the previous case, the absolute values are higher. This is due to the fact that queries are allowed more time to propagate; therefore, they explore a larger part of the network and return more results. As a consequence, the relative performance of the algorithms changes. For example, $Frozen_{10\%}$ is now better than $Frozen_{30\%}$ and $Frozen_{50\%}$ for $Q_{us} = 10^{-2}$, because even if there are congested points in the network, there is enough time for the messages to pass through. In theory, if infinite $MaxWaitTime$ were allowed, nf would achieve the best performance for any value of Q_{us}; the quality of results would decrease monotonically if more queries were frozen. By combining this with the previous observations, we conclude that the percentage of frozen queries should increase when Q_{us} increases or $MaxWaitTime$ decreases.

In the previous experiments, there was a point beyond which the results improved by freezing more queries. In general, however, this is not always true as one can see in Figure 6.10. For this experiment, we again set $MaxWaitTime = 60$sec but we changed the network structure from Power-Law to Uniform. If the percentage of frozen queries is kept below 50%, the behavior is similar to the previous cases. Nevertheless, the performance of $Frozen_{70\%}$ is always worse. This is justified as follows: In a power-law network there are some nodes that receive exponentially more messages that the others. By freezing more queries, such nodes benefit because they become less congested. In a uniform network, on the other hand, the variation of the

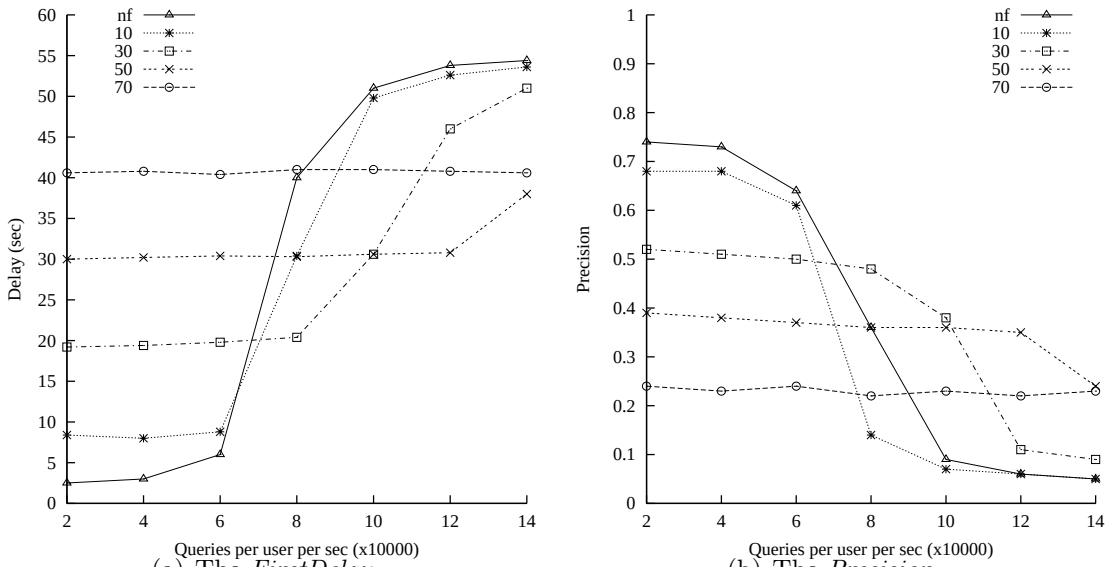


(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.10: Non-frozen(nf) vs. 10, 30, 50, 70% Statically Frozen Queries. MaxWaitTime = 60sec, Uniform Network.

peers' workload is not so significant. By freezing 50% of the queries, almost no node is overloaded anymore. If more queries are frozen, there do not remain enough running queries to provide answer streams; therefore, the performance does not improve.

In this experiment, $Q_{us} = 16 \cdot 10^{-3}$ corresponded to one query per user every 63sec. Since $MaxWaitTime$ = 60sec, this was almost the highest allowed value for Q_{us}, assuming that a single user could not have more than one query running at the same time. By altering this assumption, $Frozen_{70\%}$ could perform better than $Frozen_{50\%}$ beyond some threshold value of Q_{us}. In theory, this is always possible if the number of simultaneous queries per user is unbounded. In practice, however, this parameter is limited. Summarizing, the network structure also affects the performance of the freezing algorithms.



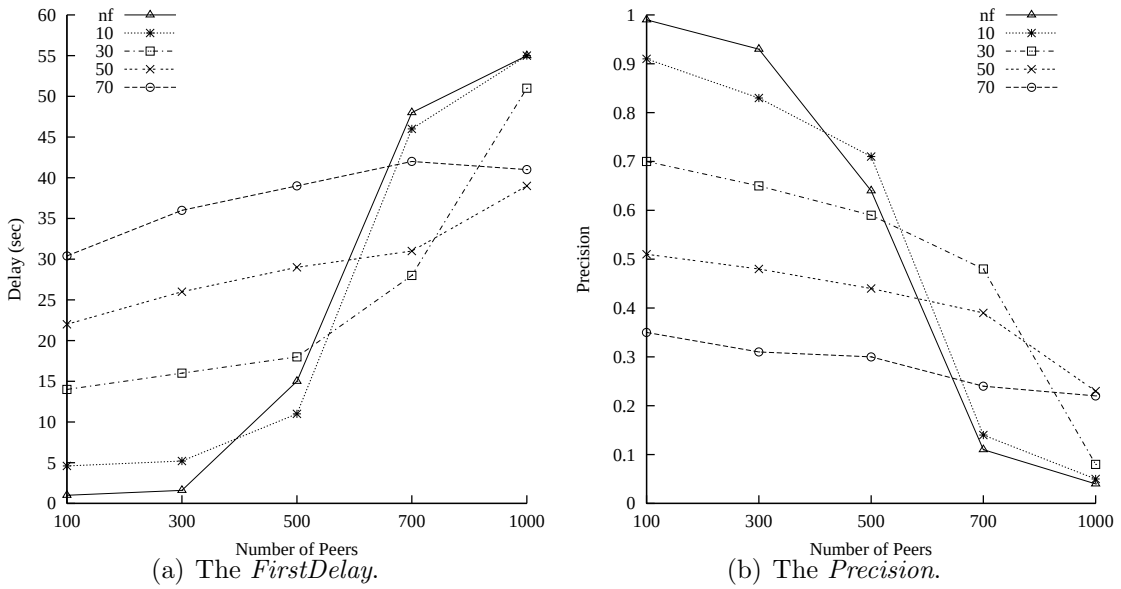
(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.11: Non-frozen vs. Statically Frozen Queries. 1000 peers, $MaxWaitTime$ = 60sec, Power Law Network.

We also tested the behavior of the system for larger user populations. In Figure 6.11, we show the results for a power-law network with 1000 simultaneously active users; $MaxWaitTime$ is 60sec. The trend of the algorithms is the same, although the results were obtained for lower values of Q_{us}. The maximum and minimum values of the x-axis correspond to a range of one query per user every 12 to 83min. Notice that the absolute number of concurrent queries that the network supported, remains roughly the same as in the previous experiments (i.e., the number of nodes is increased and Q_{us} decreased by one order of magnitude). Also observe that the best value for *Precision* drops to 0.75 compared to 0.98 in the previous experiments. This happens because the maximum number of hops a query message can travel is still seven. When the number of nodes is small, seven hops are enough to cover the entire network, but for 1000 nodes there exist parts of the network outside the query

radius.

The effect of the active user population size is further investigated in the experiments of Figure 6.12. Q_{us} is fixed to $14 \cdot 10^{-4}$ and the number of peers varies from 100 to 1000. The results verify that increasing the number of online peers has the same effect of increasing the query rate. Therefore, our freezing technique improves the scalability of the system both in terms of throughput and number of active users.



(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.12: Non-frozen vs. Statically Frozen Queries. $Q_{us} = 14 \cdot 10^{-4}$, MaxWaitTime = 60sec, Power Law Network.

Recall that in all cases so far, the frozen query was attached to the most beneficial stream at the freezing peer ($Stream_{BEST}$ method). Here, we investigate whether there is any performance gain by attaching a frozen query to multiple streams. We tested several combinations and present here the extreme case where the frozen query is attached to every available stream ($Stream_{ALL}$). We employed a power-law network with 100 peers and $MaxWaitTime$ = 60sec. In Table 6.2, we show the difference

of the $FirstDelay(Stream_{BEST}) - FirstDelay(Stream_{ALL})$, and in Table 6.3, the difference of $Precision(Stream_{ALL}) - Precision(Stream_{BEST})$. Therefore, in both tables a positive value indicates that $Stream_{ALL}$ is better. In some cases $Stream_{ALL}$ improves the performance; this happens because each frozen query receives more answers, so there is a higher probability to locate useful results. In other cases, however, $Stream_{ALL}$ is worse; this is due to network overloading from the excessive amount of answer messages that are returned for every frozen query. Nevertheless, in both cases, the difference is not significant. We conclude that our heuristic for selecting the most beneficial stream, performs reasonably well in practice.

Table 6.2: FirstDelay($Stream_{BEST}$) – FisrtDelay($Stream_{ALL}$)

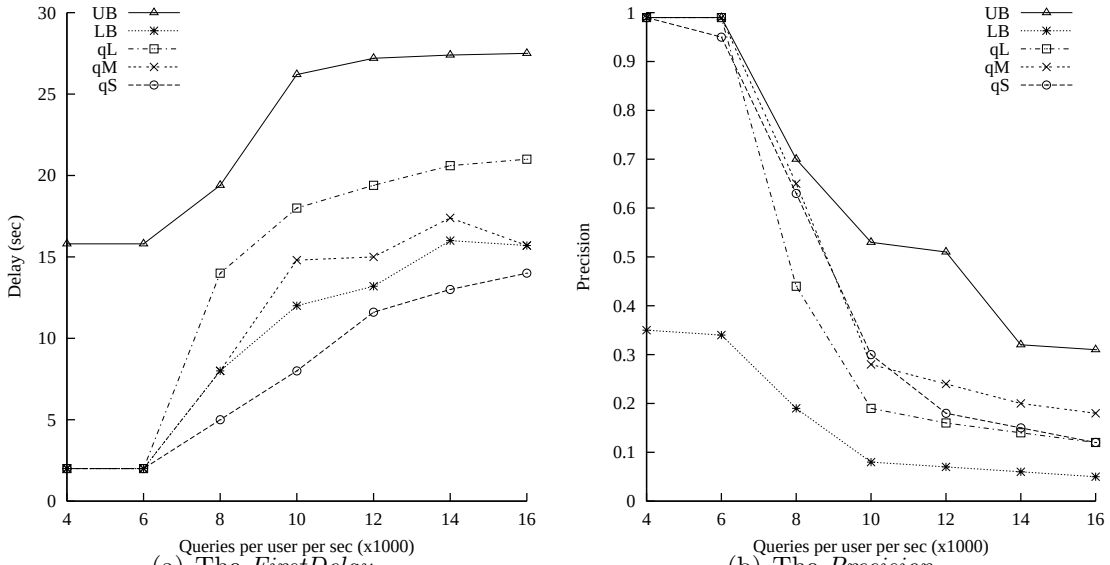
		Queries / user per sec ($\times 10^{-3}$)			
		4	**8**	**12**	**16**
% frozen	**10**	0	101	-172	-166
	30	0	627	-685	-468
	50	0	462	-81	-1345
	70	0	522	-168	-1039

Table 6.3: Precision($Stream_{ALL}$) – Precision($Stream_{BEST}$)

		Queries per user per sec ($\times 10^{-3}$)			
		4	**8**	**12**	**16**
% frozen	**10**	0.0000	-0.0015	0.0004	0.0000
	30	0.0000	0.0005	0.0013	0.0003
	50	0.0000	0.0017	0.0055	0.0028
	70	0.0000	0.0013	0.0310	0.0532

6.4.2 Adaptive Query Freezing

In the previous section, we showed that by freezing some queries, the performance of the entire system can be greatly improved. Static freezing, however, is not a practical solution since it does not consider the parameters that affect performance, such as the query rate, the number of users, *MaxWaitTime* and the network structure. In the following, we shall evaluate our adaptive freezing algorithm AQF, which takes into account these factors.



(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.13: 100 peers, MaxWaitTime = 30sec, Power Law Network

Figure 6.13 presents the results for a power-law network with 100 peers and $MaxWaitTime$ = 30sec. The settings are the same as for the experiment of Figure 6.8. For every value of Q_{us}, we compute the maximum (Upper Bound UB) and the minimum (Lower Bound LB) of the metrics from all the algorithms presented in Figure 6.8 (including nf). The adaptive algorithm is compared against the best and worst performance achieved by the static methods.

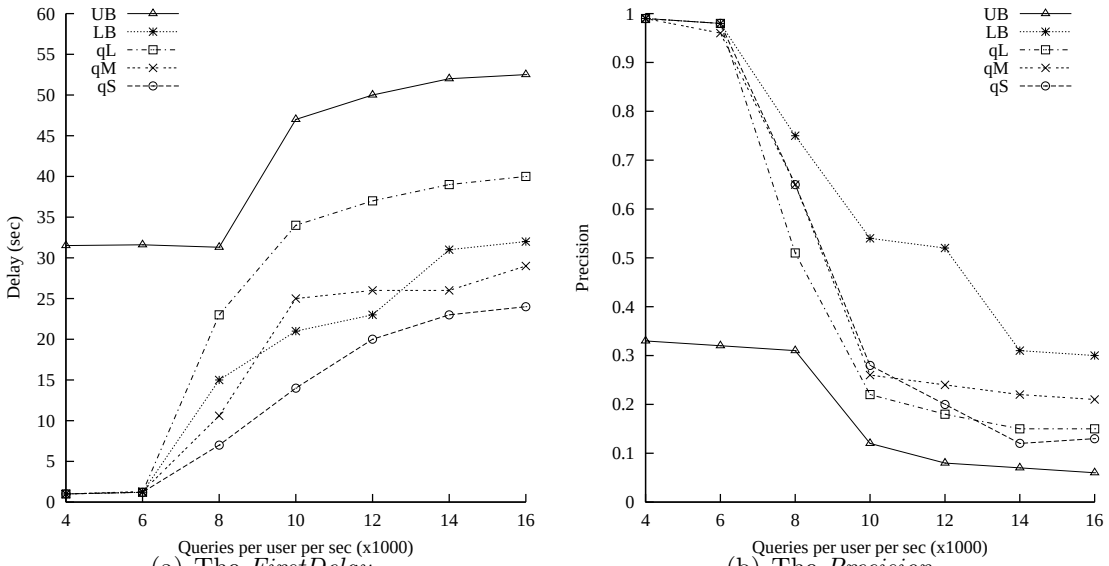
Recall that AQF has only one system-wide parameter a_q: A query q will freeze at peer P only if the average waiting time at the message queue of P is greater than $a_q \cdot MaxWaitTime$. We present the results for three values of a_q: (i) $a_q = 4$, which produces long message queues qL, (ii) $a_q = 1$, which corresponds to medium queues qM, and (iii) $a_q = 1/16$ for short queues qS.

Consider the qL case first. For $Q_{us} \leq 6 \cdot 10^{-3}$, there are relatively few queries propagated simultaneously in the network. Consequently, the waiting time at the message queues at most peers is less than $4 \cdot MaxWaitTime$, so AQF does not freeze any queries; thus AQF behaves like nf. When Q_{us} increases, longer message queues appear. AQF starts freezing some queries and outperforms nf. Nevertheless, the results are still worse than the best static freezing algorithm.

The qM case, on the other hand, prohibits the generation of long queues by freezing more queries. Notice that qM also behaves like nf for $Q_{us} \leq 6 \cdot 10^{-3}$, and beyond this, it follows closely the best static result in terms of $FirstDelay$. $Precision$ also improves compared to qL, but it is still not as good as UB. We investigated further this issue and observed that for uniform networks qM was closer to the best static results. The problem with the power-law network is that the length of the queues may be much larger in some peers (i.e., they may differ up to two orders of magnitude for our settings). In such peers, most of the queries are frozen even if the similarity with the attached streams is low, leading to low $Precision$. Improving this aspect of AQF is part of our on-going work.

We also tested the qS case which results in shorter message queues. An interesting observation is that the results for $FirstDelay$ are better than the best case of all static alternatives. This illustrates that the static algorithms are not optimal for any

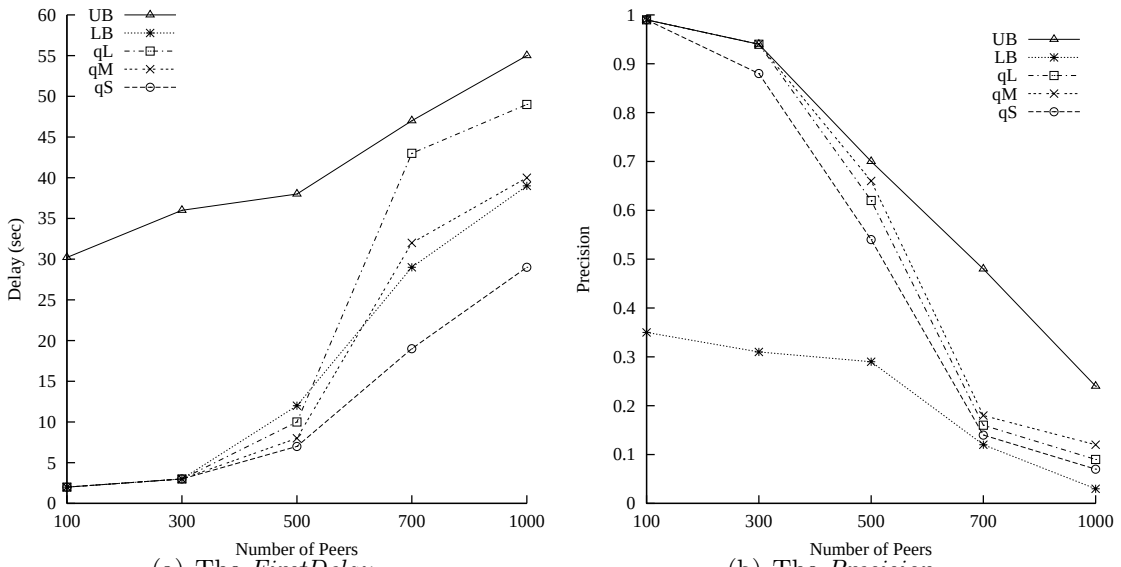
percentage of frozen queries, since they do not consider the different conditions at each of the peers that freezes the query. Notice that qS outperforms qM in terms of $FirstDelay$ because the waiting time of the messages in the queues is shorter. However, qS freezes too many queries, so the number of useful answers that reach the initiating peers drops; therefore, qS is worse than qM in terms of $Precision$. We also experimented with smaller values of a_q. In those cases, both $FirstDelay$ and $Precision$ deteriorated since many queries were expiring prior to receiving any useful answer.



(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.14: 100 peers, MaxWaitTime = 60sec, Power Law Network.

The above results were also verified by the experiment of Figure 6.14, where $MaxWaitTime$ is set to 60sec. UB and LB correspond to the upper and lower bound of the static algorithms of Figure 6.9. A point to note here, is that qS is worse in terms of $Precision$ than both qM and qL, for high query rates. Again this is due to the excessive number of queries that are frozen by qS.



(a) The *FirstDelay*. (b) The *Precision*.

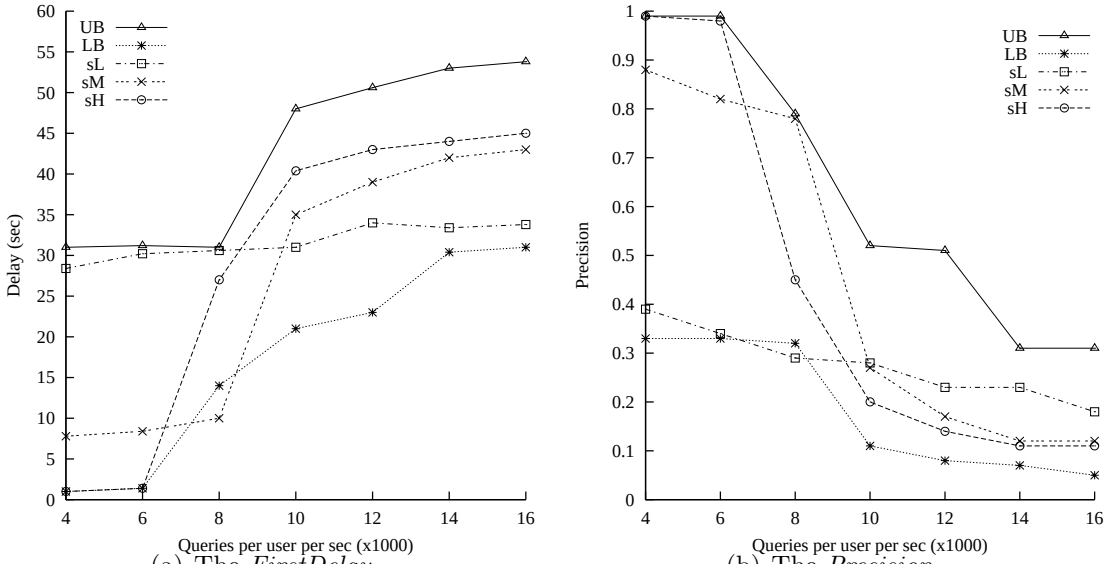
Figure 6.15: $Q_{us} = 14 \cdot 10^{-4}$, MaxWaitTime = 60sec, Power Law Network.

We also tested AQF for larger user populations. In Figure 6.15, we set $Q_{us} = 14 \cdot 10^{-4}$ and varied the number of users from 100 to 1000 in a power-law network. UB and LB are calculated from the values of Figure 6.12. The results also support our previous observations. Summarizing, AQF is a practically useful algorithm since it achieves good performance with minimal parameter tuning. Our best results were obtained for $a_q = 1$ which means that the average delay in the message queues must be kept close to $MaxWaitTime$. Our experiments, however, revealed that AQF is robust so the exact value of a_q is not critical.

6.4.3 Similarity Query Freezing Algorithm

This section evaluates the performance of our alternative freezing algorithm simQF. Recall that while AQF decides whether to freeze a query based on the message queue waiting time, simQF employs a similarity criterion; a query q is frozen at peer P if

there exists an answer stream at P whose distance from q is less than ρ.



(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.16: Similarity Query Freezing. 100 peers, MaxWaitTime = 60sec, Power Law Network.

In Figure 6.16, we present the results for three similarity thresholds: (i) a low similarity case sL where $\rho = 4$, (ii) medium similarity sM with $\rho = 3$, and (iii) high similarity sH, where $\rho = 1$. There are 100 peers in a power-law network; UB and LB are copied from Figure 6.13 and 6.14. For low query rates, the sH case performs better since it does not freeze many queries due to the tight similarity threshold. For the same reason, however, sH deteriorates fast as Q_{us} increases. Notice that this behavior is similar to nf. Also observe that sM and sL freeze gradually fewer queries; therefore their relative performance compared to sH is low for slow query rates and improves as Q_{us} increases. This behavior is identical to static query freezing, where a tight similarity threshold ρ corresponds to a low percentage of frozen queries.

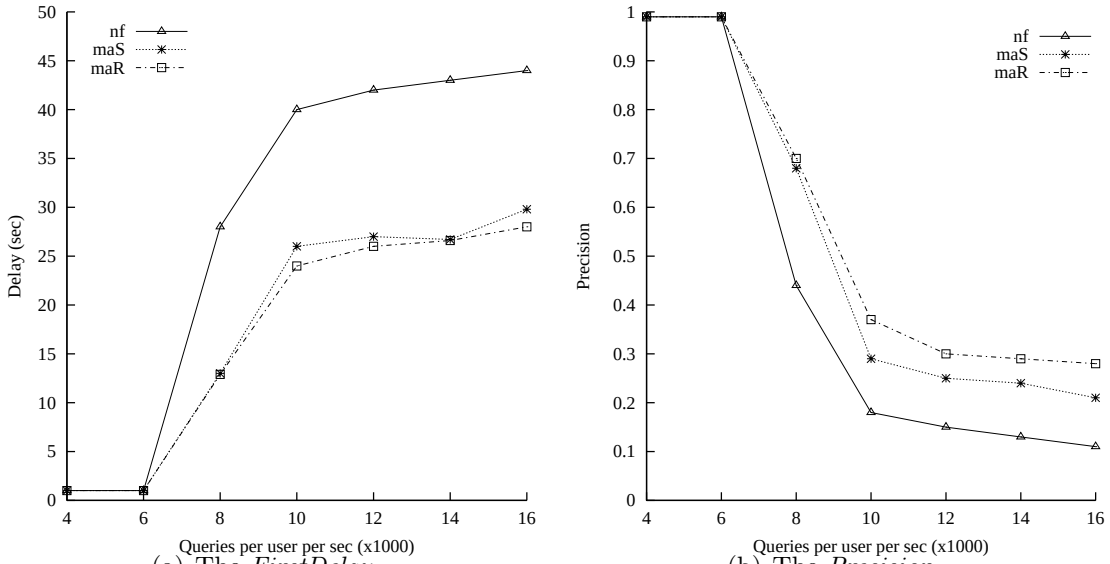
Comparing simQF with AQF, we should note that simQF does not adapt to the

workload of the system, therefore its applicability is limited in practice. The similarity threshold of simQF may be considered a more natural alternative to the static freezing algorithms which require an ad-hoc parameter (i.e., the percentage of frozen queries). Even from this perspective, the similarity threshold is difficult to be defined, since it depends on the application, the user's perception and the query itself. In contrast, the parameter of AQF can be easily derived and it is application independent.

6.4.4 Multiple-feature Queries

In our final set of experiments, we tested the performance of AQF for multiple-feature queries. We employed the SYNTH_{200} dataset in a power-law network with 100 peers. For our adaptive algorithms we set $a_q = 1$. As in the previous experiments, we submitted 1000 queries chosen randomly from our dataset. Since SYNTH_{200} had two feature vectors, there were three possible types for every query (i.e., q_{f1}, q_{f2} and $q_{f1,f2}$). The query type was selected with uniform distribution.

The results are presented in Figure 6.17. nf is the traditional non-freezing broadcast-based algorithm. It does not consider the similarities among query types; q_{f1} and $q_{f1,f2}$) for instance, are treated as different queries. Our multiple-feature serial algorithm maS also considers such queries as different. Therefore, the improved performance of maS is due to adaptive query freezing. The trends of the results are the same as these presented in Figure 6.13 and 6.14, although the absolute values fluctuate due to the different datasets. The results of the multiple-feature random algorithm maR are more interesting. Recall that maR exploits the similarities among query types and includes a refinement step, where it performs direct accesses to non-neighbor peers in order to retrieve better answers. Compared to maS, $FirstDelay$ does not improve significantly. This is expected since the refinement step of maR is



(a) The *FirstDelay*. (b) The *Precision*.

Figure 6.17: Multiple-feature Queries. 100 peers, MaxWaitTime = 60sec, Power Law Network, $a_q = 1$, SYNTH_{200} dataset.

initiated after 80% of $MaxWaitTime$ has passed. During this interval, most of the queries have already received their first useful answer. On the other hand, there is a more obvious improvement in terms of *Precision*, since direct accesses can fetch useful results which would be otherwise inaccessible.

The good performance of maR is partially due to the fact that SYNTH_{200} was clustered. We also ran the same experiments with the unclustered REAL_{181}. In this case, maR performed almost identically to maS. Actually, for some settings maR was slightly worse that maS due to the overhead of initiating new connections. Nevertheless, in practice, this overhead is not significant, and since we do not assume any knowledge about the properties of the dataset properties, the possible benefits of maR justify its employment.

6.5 Summary

In this chapter, we have dealt with the problem of retrieving information from large repositories built on top of ad hoc P2P networks. While most existing approaches are limited to exact key matching, we have developed FuzzyPeer which supports content-based similarity queries. Due to the absence of centralized indexing, such queries are challenging; the difficulty of defining an application-independent terminating criterion in addition to the extremely low selectivity of the queries, overloads the system with useless messages and causes thrashing. We have addressed this issue by introducing the freezing technique: some queries are paused and attached to answer streams from similar concurrently running ones, since the answers for both queries are expected to overlap. We proposed AQF, a simple yet efficient distributed optimization algorithm which improves the scalability and the throughput of the system. Numerous applications, including full-text search in large archives or fuzzy queries in distributed multimedia repositories, can benefit from our techniques. We have demonstrated this by a case study of an image retrieval application.

Chapter 7

Conclusion

The objective of this research is to investigate and propose heuristic approaches of data sharing and system management in ad hoc P2P systems without strong control over the topology of the network and the contents of each peer. We have addressed several common problems in the database community but with specific requirements for P2P data sharing and management systems.

We have proposed several query processing techniques for the P2P environment without relying on any global schemes or knowledge. Chapter 3 discusses a simple methodology where every BestPeer node maintains a statistics log of its environment. The logs are updated each time after some query results are obtained. Based on the statistics, optimization such as self-reconfiguring the network to achieve better performance for subsequent queries is applied. PeerOLAP (Chapter 5) process queries in a fashion similar to BestPeer, where queries are broadcast to the P2P network. However, in contrast to BestPeer, PeerOLAP employs a set of heuristics in order to limit the number of peers that are accessed. The decision-making of FuzzyPeer (Chapter 6) is achieved by monitoring the results streaming through remote peers that are closer to the ideal P2P system. Such an approach eliminates the need of obtaining

the status of each peer in its environment, while facilitating a clearer picture of its environment for decision-making.

The issues of the heterogeneity of data sources are extensively studied in Chapter 4. PeerDB is proposed for such purposes, where IR techniques are used for solving the aforementioned tasks. Each peer is allowed to define its schema without any global constraints. Meta-data is used to resolve the conflict of different semantic objects with different syntactic presentations.

We have studied the consequences of data placement problems in a dynamic environment and reported our findings in Chapter 5. In particular, we have focused on data placing problems for OLAP applications. As shown in the experimental evaluation, with proper selection placement strategies, even though with ad hoc participants, it is possible to achieve significant performance gains over traditional systems.

With regard to the above multiple data granularity access problems, we have designed the BestPeer platform, which integrates with mobile agent technology (details in Chapter 3). Mobile agent offers several advantages as compared to traditional static data access methodologies. It allows extensibility to existing systems and finer granularity of data sharing where partial content of a file or data may be shared.

There exist several topologies such as Chord [100], CAN[92] and Pastry[31] that allow queries to be answered within a bounded number of hops, since search is guided by a hash function. However, we are interested in P2P systems like Gnutella, where search is distributed in an overlay network. When a new peer PN wishes to join the network, it first acquires the address of an arbitrary peer with an empty slot. A peer P broadcasts a query to all its neighbors, which propagates it recursively. If any of the visited peers contain a result, it sends it back to P directly. A peer can also broadcast

exploration messages, when some of its neighbors abandon it (i.e., go off-line). This topology has served as a basic design guideline for the implementation of the BestPeer network architecture. In addition, data replication may improve the performance and responsiveness of P2P data sharing and management systems. However, it makes the updates much harder, and maintaining consistency over replicated objects is a well-known database problem. In this thesis, we have applied a limited degree of data replication for P2P applications, where data updates are infrequent, such as OLAP applications.

In this work, we have presented some preliminary fundamental results, and described our initial work in the construction of an adaptive P2P data sharing and management system. The results of this study have confirmed our contribution in P2P-like distributed data sharing systems that support dynamic data and dynamic workloads.

7.1 Future Scope of Work

We plan to extend PeerDB in several directions. First, we plan to make a node more intelligent by allowing it to determine at runtime which strategy to adopt – code-shipping or data-shipping. Second, we have focused on looking for "similar" schemas. More recently, the keyword-based search engine for relational databases has been developed [12]. We plan to see how such features can be integrated into our system to facilitate keyword-based search in PeerDB. Third, we are continuing the work on joining the relations from multiple nodes. Joining relations from a single node can be done by MySQL. However, we need to implement our own algorithm to join relations from multiple nodes. We plan to use AJoin [102] as the joining algorithm as it can

provide continuous answers to the user as soon as data arrives. Unlike traditional query processing techniques, AJoin blocks only when all available data have been examined. As a result, AJoin delivers its response to the user as soon as possible.

We will investigate the option of developing more sophisticated algorithms for network reconfiguration in PeerOLAP. Identifying the neighborhoods of peers with similar access patterns is essentially a clustering problem, which however, is difficult to solve because: (i) there is no complete knowledge about the whole network at any site; thus, each peer must make decisions using only partial information, and (ii) the available information constantly changes as the caches get updated, and peers enter/leave the network.

We are working on incorporating dynamic network reconfiguration to FuzzyPeer. The idea is to alter dynamically the set of neighbors of some peers in order to minimize the required number of query hops. In the future, we are also planning to support general database queries through the use of XML.

Bibliography

[1] *BestPeer Project Home Page*, http://xena1.ddns.comp.nus.edu.sg/p2p/.

[2] *FURI*, http://www.jps.net/williamw/furi.

[3] *MySql Home Page*, http://www.mysql.com/.

[4] *Visibroker*, http://info.borland.com/techpubs/visibroker/.

[5] *WebSphere*, http://www-3.ibm.com/software/info1/websphere/index.jsp.

[6] A. Tanenbaum and A. Woodhull, *Operating systems design and implementation*, Prentice-Hall Inc, 1999.

[7] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, and J. Wu, *Advanced peer-to-peer networking: The P-Grid System and its Applications*, PIK Journal - Praxis der Informationsverarbeitung und Kommunikation, Special Issue on P2P Systems (2003).

[8] K. Aberer, P. Cudré-Mauroux, and M. Hauswirth, *A framework for semantic gossiping*, SIGMOD Record, 31(4) (2002).

[9] K. Aberer and M. Hauswirth, *Peer-to-peer information systems: concepts and models, state-of-the art, and future systems*, Tutorial at International Conference on Data Engineering (ICDE), 2002.

[10] Karl Aberer, *P-Grid: A self-organizing access structure for P2P information systems*, Lecture Notes in Computer Science **2172** (2001).

[11] S. Abiteboul and O. Duschka, *Complexity of answering queries using materialized views*, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 1998, pp. 254–263.

[12] S. Agrawal, S. Chaudhuri, and G. Das, *Dbxplorer: A system for keyword-based search over relational databases*, Proceedings of the 18th International Conference on Data Engineering (San Jose, CA), April 2002.

[13] J. Albrecht and W. Lehner, *On-line analytical processing in distributed data warehouses*, IDEAS, 1998, pp. 78–85.

[14] A. Andrzejak and Z. Xu, *Scalable, efficient range queries for grid information services*, The Second IEEE International Conference on Peer-to-Peer Computing (P2P2002), 2002.

[15] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier, *Space filling curves and their use in geometric data structure*, Theoretical Computer Science, 1997, pp. 3–15.

[16] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern information retrieval*, ACM Press/Addison-Wesley, 1999.

[17] S. Bergamaschi, S. Castano, D. Beneventano, and M. Vincini, *Semantic integration of heterogeneous information sources*, Special Issue on Intelligent Information Integration, Data & Knowledge Engineering **36** (2001), no. 1, 215–249.

[18] S. Bressan, C.L. Goh, B.C. Ooi, and K.L. Tan, *Supporting extensible buffer replacement strategies in database systems*, ACM SIGMOD International Conference on Management of Data, 1999.

[19] J. Byers, J. Considine, and M. Mitzenmacher, *Simple load balancing for distributed hash tables*, 2nd International Workshop on Peer-to-Peer Systems (IPTPS), February 2003.

[20] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, *Answering regular path queries using views*, International Conference on Data Engineering (ICDE), 2000, pp. 389–398.

[21] P. Cao, J. Zhang, and P. B. Beach, *Active cache: Caching dynamic contents on the web*, Middleware Conference, 1998.

[22] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, *Towards heterogeneous multimedia information systems: The garlic approach*, International Workshop on Research Issues in Data Engineering(RIDE): Distributed Object Management, 1996.

[23] A. Castillo, M. Kawaguchi, N. Paciorek, and D. Wong, *Concordia as enabling technology for cooperative information gathering*, Proceedings of the 31th Annual Hawaii International Conference on System Sciences 1998 (HICSS31), 1998.

[24] C.C.K. Chang and H. García-Molina, *Mind your vocabulary: query mapping across heterogeneous information sources*, ACM SIGMOD International Conference on Management of Data, 1999, pp. 335–346.

[25] A. Crespo and H. García-Molina, *Routing indices for peer-to-peer systems*, International Conference on Distributed Computing Systems (ICDCS), 2002.

[26] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan, *Semantic data caching and replacement*, VLDB, 1996, pp. 330–341.

[27] P. Deshpande and J. F. Naughton, *Aggregate aware caching for multi-dimensional queries*, International Conference on Extending Database Technology (EDBT), 2000, pp. 167–182.

[28] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton, *Caching multidimensional queries using chunks*, ACM SIGMOD International Conference on Management of Data, 1998, pp. 259–270.

[29] A. Doan, P. Domingos, and A. Y. Halevy, *Reconciling schemas of disparate data sources: A machine-learning approach*, ACM SIGMOD International Conference on Management of Data, 2001.

[30] A. Doan, J. Madhavan, P. Domingos, and A. Halevy, *Learning to map between ontologies on the semantic web*, World-Wide Web Conference, 2002.

[31] P. Druschel and A. Rowstron, *Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems*, IFIP/ACM International Conference on Distributed systems platforms (Middle ware), 2001, pp. 329–350.

[32] D. W. Embley, D. Jackman, and L. Xu, *Multifaceted exploitation of metadata for attribute match discovery in information integration*, Workshop on Information Integration on the Web, 2001, pp. 110–117.

[33] Entropia Home Page, *http://www.entropia.com*.

[34] R. Fagin, *Combining fuzzy information from multiple systems*, ACM Symp. on Principles of Database Systems (PODS), 1996, pp. 216–226.

[35] R. Fagin, A. Lotem, and M. Naor, *Optimal aggregation algorithms for middleware*, ACM Symp. on Principles of Database Systems (PODS), 2001.

[36] M. Faloutsos, P. Faloutsos, and C. Faloutsos, *On power-law relationships of the internet topology*, ACM SIGCOMM, 1999, pp. 251–262.

[37] T. Finin, R. Fritzson, D. McKay, and R. McEntire, *KQML as an Agent Communication Language*, 3rd International Conference on Information and Knowledge Management (CIKM), 1994, pp. 456–463.

[38] D. Florescu, A. Y. Levy, and A. O. Mendelzon, *Database techniques for the world-wide web: A survey*, SIGMOD Record **27** (1998), no. 3, 59–74.

[39] Freenet Home Page, *http://freenet.sourceforge.com/*.

[40] H. García-Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge, *Distributed and parallel computing issues in data warehousing*, ACM Symposium on Principles of Distributed Computing, 1998.

[41] Graham Glass, *Overview of voyager: Objectspace's product family for stateof-the-art distributed computing.*, White paper, ObjectSpace, http://www.objectspace.com/products /documentation /VoyagerOverview.pdf, 1999.

[42] Gnutella Development Home Page, *http://gnutella.wego.com/*.

[43] C. L. Goh, S. Bressan, B. C. Ooi, and M. Anirban, *Storm: A 100% java persistent storage manager*, OOPSLA Workshop on Java and Object, 1999.

[44] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu, *What can databases do for peer-to-peer?*, WebDB Workshop on Databases and the Web, 2001.

[45] L. M. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz, and E. L. Wimmers, *Transforming heterogeneous data with database middleware: Beyond integration*, IEEE Data Engineering Bulletin **22** (1999), no. 1, 31–36.

[46] A. Halevy, O. Etzioni, A.H. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov, *Crossing the structure chasm*, 2003.

[47] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov, *Piazza: Data Management Infrastructure for Semantic Web Applications*, The 12th International World Wide Web Conference, 2003.

[48] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov, *Schema Mediation in Peer Data Management Systems*, International Conference on Data Engineering (ICDE), 2003.

[49] V. Harinarayan, A. Rajaraman, and J. D. Ullman, *Implementing data cubes efficiently*, ACM SIGMOD International Conference on Management of Data, 1996, pp. 205–216.

[50] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica, *Complex queries in dht-based peer-to-peer networks*, International Workshop on Peer-to-Peer Systems (IPTPS02), 2002.

[51] R. Hull and G. Zhou, *A framework for supporting data integration using the materialized and virtual approaches*, ACM SIGMOD International Conference on Management of Data, 1996, pp. 481–492.

[52] ICQ Home Page, *http://www.icq.com/*.

[53] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K. L. Tan, *An adaptive peer-to-peer network for distributed caching of olap results*, ACM SIGMOD International Conference on Management of Data, 2002, pp. 25–36.

[54] P. Kalnis and D. Papadias, *Proxy-server architectures for olap*, ACM SIGMOD International Conference on Management of Data, 2001, pp. 367–378.

[55] G. Karjoth, D.B. Lange, and M. Oshima, *A Security Model for Aglets*, IEEE Internet Computing **1** (1997), no. 4.

[56] N. Karnik and A. Tripathi, *Agent Server Architecture for the Ajanta Mobile-Agent Systems*, International Conference on Parallel and Distributed Processing Techniques and Applications, 1998.

[57] KDD Cup 2001, *http://www.cs.wisc.edu/ dpage/kddcup2001/*.

[58] A. M. Keller and J. Basu, *A predicate-based caching scheme for client-server database architectures*, VLDB Journal **5** (1996), no. 1, 35–47.

[59] A. Kementsietsidis, M. Arenas, and R. J. Miller, *Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues*, ACM SIGMOD International Conference on Management of Data, 2003.

[60] J. Kleinberg, *Small-world phenomena and the dynamics of information*, Advances in Neural Information Processing Systems (NIPS), 2001.

[61] D. Kossmann, *The state of the art in distributed query processing*, ACM Computing Surveys **32** (2000), no. 4, 422–469.

[62] Y. Kotidis and N. Roussopoulos, *Dynamat: A dynamic view management system for data warehouses*, ACM SIGMOD International Conference on Management of Data, 1999, pp. 371–382.

[63] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, *Oceanstore: An architecture for global-scale persistent storage*, Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), 2000.

[64] D. Ursino L. Palopoli, G. Terracina, *The system dike: Towards the semi-automatic synthesis of cooperative information systems and data warehouses*, ADBIS-DASFAA, 2000, pp. 108–117.

[65] D. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.

[66] M. Lenzerini, *Data integration: A theoretical perspective*, ACM Symp. on Principles of Database Systems (PODS), 2002, pp. 233–246.

[67] LOCKSS Home Page, *http://lockss.stanford.edu/*.

[68] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias, *Active caching of on-line-analytical-processing queries in www proxies*, International Conference On Parallel Processing, 2001, pp. 419–426.

[69] J. Madhavan, P. A. Bernstein, and E. Rahm, *Generic schema matching with cupid*, International Conference on Very Large Data Bases (VLDB), 2001, pp. 49–58.

[70] R. J. Miller, M. A. Hernandez, L. M. Haas, L. Yan, C. T. Howard Ho, R. Fagin, and L. Popa, *The clio project: Managing heterogeneity*, **30** (2001), no. 1, 78.

[71] T. Milo and S. Zohar, *Using schema matching to simplify heterogeneous data translation*, International Conference on Very Large Data Bases (VLDB), 1998, pp. 122–133.

[72] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, *Peer-to-peer computing*, Technical Report HPL-2002-57, HP Laboratories Palo Alto, March 2002.

[73] Mitsubishi Electric, *Concordia: An infrastructure for collaborating mobile agents*, Proceedings of the 1st International Workshop on Mobile Agents (MA '97), April 1997.

[74] Morpheus Home Page, *http://www.morpheus-os.com/*.

[75] Napster Home Page, *http://www.napster.com/*.

[76] NFS Version 4 Home Page, *http://www.nfsv4.org/*.

[77] W. S. Ng, B. C. Ooi, and K. L. Tan, *BestPeer: A Self-Configurable Peer-to-Peer System*, Poster in International Conference on Data Engineering (ICDE), 2002, p. 272.

[78] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Y. Zhou, *PeerDB: A P2P-based System for Distributed Data Sharing*, International Conference on Data Engineering (ICDE), 2003, pp. 633–644.

[79] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin, *The qbic project: Querying images by content using color, texture and shape.*, In Storage and Retrieval for Image and Video Databases (SPIE), 1993, pp. 173–187.

[80] H. S. Nwana, D. T. Ndumu, L. C. Lee, and J. C. Collis, *ZEUS: A Toolkit and Approach for Building Distributed Multi-Agent Systems*, International Conference on Autonomous Agents (Agents) (Seattle, WA, USA), 1999, pp. 360–361.

[81] Object Management Group, *http://www.omg.org/*.

[82] *Olap council apb-1 olap benchmark r-ii*, http://www.olapcouncil.org.

[83] B. C. Ooi, K. L. Tan, H. J. Lu, and A. Y. Zhou, *P2P: Harnessing and Riding on Peers*, The 19th National Conference on Data Bases, August 2002.

[84] B. C. Ooi, K. L. Tan, A. Y. Zhou, C. H. Goh, Y. G. Li, C. Y. Liau, B. Ling, W. S. Ng, Y. F. Shu, X. Y. Wang, and M. Zhang, *PeerDB: Peering into Personal Databases*, ACM SIGMOD International Conference on Management of Data (Demo), 2003.

[85] A. Oram, *Peer-to-peer : Harnessing the power of disruptive technologies*, 2001.

[86] M. T. Ozsu and P. Valduriez, *Principles of distributed database systems*, Prentice Hall, 1999.

[87] Panos Kalnis, *Static and dynamic view selection in distributed data warehouse systems.*, PhD Thesis (Computer Science Dept., University of Science and Technology, Hong Kong.), 2002.

[88] Parabon Computation Home Page, *http://www.parabon.com/*.

[89] C. Parent and S. Spaccapietra, *Database integration: an overview of issues and approaches*, Communications of the ACM **41** (1998), no. 5, 166–178.

[90] A. B. Philip, G. Fausto, K. Anastasios, M. John, S. Luciano, and Z. Ilya, *Data management for peer-to-peer computing: A vision*, WebDB Workshop on Databases and the Web, 2002.

[91] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, *Load balancing in structured p2p systems*, 2nd International Workshop on Peer-to-Peer Systems (IPTPS), February 2003.

[92] S. Ratnasamy, R. Francis, M. Handley, R. Krap, J. Padye, and S. Shenker, *A Scalable Content-Addressable Network*, ACM SIGCOMM, 2001.

[93] A. Rowstron and P. Druschel, *Past: A large scale persistent peer-to-peer storage utility*, Workshop on Hot Topics in Operating Systems (HotOS), November 2001.

[94] P. Scheuermann, J. Shim, and R. Vingralek, *Watchman : A data warehouse intelligent cache manager*, VLDB, 1996, pp. 51–62.

[95] SETI@home Home Page, *http://setiathome.ssl.berkely.edu/*.

[96] A. Shukla, P. Deshpande, and J. F. Naughton, *Materialized view selection for multidimensional datasets*, International Conference on Very Large Data Bases (VLDB), 1998, pp. 488–499.

[97] A. Shukla, P. Deshpande, and J. F. Naughton, *Materialized view selection for multi-cube data models*, International Conference on Extending Database Technology (EDBT), 2000, pp. 269–284.

[98] I. A. Smith and P. R. Cohen, *Toward a Semantics for an Agent Communications Language Based on Speech-Acts*, 13th National Conference Artificial Intelligence, (AAAI Press), 1996.

[99] Squid Web Proxy Cache, *http://www.squid-cache.org/*.

[100] I. Stocia, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, ACM SIGCOMM, 2001.

[101] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, *Mariposa: A wide-area distributed database system*, VLDB Journal **5** (1996), no. 1, 48–63.

[102] K.L. Tan, P.K. Eng, B.C. Ooi, and M. Zhang, *Join and multi-join processing in data integration systems*, Data and Knowledge Engineering **40** (2002), no. 2, 217–239.

[103] A. Tomasic, L. Raschid, and P. Valduriez, *Scaling heterogeneous databases and the design of disco*, International Conference on Distributed Computing Systems, 1996, pp. 449–457.

[104] J. D. Ullman., *Information integration using logical views*, International Conference on Database Theory (ICDT), 1997, pp. 19–40.

[105] United Devices Home Page, *http://www.ud.com/*.

[106] R. Vincent, B. Horling, and V. Lesser, *An agent infrastructure to build and evaluate multi-agent systems: The java agent framework and multi-agent system simulator*, Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems., vol. 1887, Wagner & Rana (eds.), Springer,, January 2001.

[107] J. Z. Wang, G. Wiederhold, O. Firschein, and S. X. Wei, *Content-based image indexing and searching using daubechies' wavelets*, International Journal on Digital Libraries (1), 1997, pp. 311–328.

[108] X. Y. Wang, W. S. Ng, B. C. Ooi, K. L. Tan, and A. Y. Zhou, *BuddyWeb: A P2P-based Collaborative Web Caching System*, Position Paper in Peer to Peer Computing Workshop (Networking), 2002.

[109] B. Yang and H. García-Molina, *Comparing hybrid peer-to-peer systems*, International Conference on Very Large Data Bases (VLDB), 2001, pp. 561–570.

[110] B. Yang and H. GarcíaMolina, *Efficient search in peer-to-peer networks*, International Conference on Distributed Computing Systems (ICDCS), 2002.

[111] B. Yang and H. García-Molina, *Designing a super-peer network*, International Conference on Data Engineering (ICDE), 2003.

[112] N. Young, *On-line caching as cache size varies*, Symposium on Discrete Algorithms, 1991.

[113] C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish, *Indexing the distance: An efficient method to knn processing*, International Conference on Very Large Data Bases (VLDB), 2001.

[114] B. Y. Zhao, J. Kubiatowicz, and A. Joseph, *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing.*, Technical report, UCB/CSD-01-1141, University of California, Berkeley, 2001.