DYNAMIC DATA CONSISTENCY MAINTENANCE IN PEER-TO-PEER CACHING SYSTEM

Gao Song

(*B.S., FUDAN University, China*)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgement

I would like to express my profound gratitude to my supervisor, Prof. Ooi Beng Chin, for his brilliant guidance and continuous encouragement throughout these years. The sharing of his intellectual talents and his research dedication will be the treasure in my life. Besides, he has given me much invaluable advice on many other aspects and become more than my major professor.

I would also like to thank Prof. Tan Kian-Lee, Dr. Chi Chi Hung, Dr. Ng Wee Siong and Dr. Qian Weining who have volunteered their time and great effort during the course of my thesis research. My appreciation also extends to all the members of the NUS Database Group for countless helpful suggestions. In particular, I would like to thank the following individual NUS Group members: Cai Wenyuan, Cao Xia, Cui Bin, Li Hanyu, Shu Yanfeng, Wang Qingliang, Wang wenqiang, Xia Chenyi, Yin Jianfeng, Zhang Rui, Zhou Yongluan, and others for their technical assistance and dear friendship. Further, I would like to thank the University for providing me with a scholarship for my research study.

Finally, many thanks, which are beyond words, go to my beloved parents for the love, encouragement, and understanding throughout of my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Summary

Peer-to-peer (P2P) systems have emerged as a popular way to share huge volumes of data, because of the many benefits they offer: adaptivity, self-organization, load-balancing, fault-tolerance, high availability through massive replication, and the ability to pool together and harness large amounts of resource. On-line decision making often involves significant amount of time-varying data, which can be regarded as dynamic data. Examples of such data include financial information such as stock prices and current exchange rates, real-time traffic and weather information, and data from sensors in industrial process control applications. Most of these applications are built over centralized systems due to easy management and implementation. However, centralized systems suffer from huge population and scale problems in the dynamic data applications. Due to the advantages P2P technology could offer, it is regarded as a possible solution to replace the centralized models. Unfortunately, previous P2P research has predominantly focused on the static files management. To our best knowledge, maintaining dynamic data consistency in existing P2P systems is often inefficient. We focus on the solution to maintaining dynamic data consistency in an overlay network of cooperative peers.

We present PeerCast, an adaptive framework for efficiently disseminating dynamic data in a P2P caching system. Peers maintain their cached data consistency by participating in the framework. PeerCast combines application-level multicasting techniques and demand-driven dissemination filtering techniques to provide efficiency and load balancing utilizing cooperation between peers.

We have made the following contributions. First, we have implemented PeerCast prototype layered on our P2P platform, BestPeer [58]. Second, we have provided a set of policies on PeerCast topology maintenance. They are designed to address overlay construction, recovery from peer departure or failure and network adaption problems. Third, we have proposed heuristic approaches to optimize the network resource usages and to prevent the churning problem efficiently. Fourth, we have evaluated our strategies using a combination of experiments over BestPeer infrastructure and our simulator to collect results in large-scale network scenario. Real-world traces of dynamically changing Web data are used to examine the performance of our approach. We analyze the results and examine each impact factor in our approach. Furthermore, comparison experiments are done between PeerCast and previous research, Gtk-Gnutella [53]. The results show that our approach is more efficient than Gtk-Gnutella in several aspects and achieve significant benefits.

In summary, our techniques can efficiently satisfy the consistency requirements by different peer users. Since PeerCast is simple in design and implementation, it can be easily incorporated into existing systems for immediate impact.

CHAPTER 1

Introduction

The Internet was designed with peer-to-peer (P2P) applications in mind, but as it grew, the network became increasingly asymmetric. Asymmetric bandwidth and early commercialization of the Internet disrupted the chance for network nodes to function together as peers. Consequently, the Internet has long been dominated by the client/server computing model. Client/server computing works on the basis of powerful servers providing various kinds of services in a centralized manner. The client initiates a connection to a well-known server, downloads the data, and disconnects. However, with the booming population of Internet users, the client/server model now suffers from overloaded servers and single-point failures.

Napster [7] is the first system to recognize that popular content need not be requested from central servers but could be downloaded from peers that already possess the content. With the dismissal of the assumption of asymmetry upon which traditional ADSL and cable modem providers rely, and the increasing use of broadband connections, decentralized P2P systems have also spread out across the

Internet. The success and popularity of Gnutella [5] and Freenet [27] bring a good start to further research on P2P technology. Indeed, P2P technology has become a hot research topic. Because of the advantages and benefits of P2P technology, some tough problems may eventually be resolved with the deployment of the technology. More and more applications, such as Web caching (like Squirrel [44] and Buddy-Web [76]), multimedia sharing (like P2Cast [37], music retrieval [80]) and database applications (like PeerOLAP [47], PIER [43], PeerDB [59], BuddyCQ [57], PeerCQ [34], range queries [38, 70]) are being deployed on P2P systems. P2P technology is an emerging paradigm that is now viewed as a potential technology that could re-shape distributed architecture. We will further discuss the current P2P technology developments in Chapter 2 as background knowledge to our study.

The applications of online dynamic data processing have exponentially grown in recent years. They are different from the traditional applications, since dynamic data assume the form of continuously changing value with the variation of time. Data change frequently and unexpectedly in applications such as network measurement monitoring, stock prices and current exchange rates, real-time traffic and weather information, and data from sensors in industrial process controls. Effective handling of dynamic data becomes a very important task in recent years. Due to cost effective implementation and convenient data management, most of dynamic data applications have been built over centralized systems. However, centralized systems suffer from the problems of single-point failures, extensibility and scalability.

P2P technology has many advantages over the centralized systems, including alleviating the single-point failure problem and reducing the workload of centralized servers. However, current existing P2P systems are ill-equipped to handle dynamic data.

As important strategies in P2P systems, caching and replication techniques are well-studied topics in the context of distributed systems as a means to achieve easy data access, minimize query response time and raise system performance. For instance, in the Web scenario, Web proxy caching and content distributed networks, can scale up the origin server[1] by reducing overall load on the server. Likewise, caching and replication techniques have also been widely studied in the P2P environment in recent years [28, 78, 13, 79, 29, 22]. Quite a few of replication strategies have been proposed to increase the performance of data search and access, such as "owner replication" and "path replication" [28]. Freenet [27], OceanStore [51], PAST [69], etc., are global persistent data stores designed to scale to billions of users. They provide a consistent, highly-available and durable storage utility over an infrastructure comprised of peer nodes. Caching and replication create numerous copies of data objects scattered throughout the P2P overlay network. They bring benefits only when the cached object rarely changes. If data objects are updated frequently, the cache hierarchy becomes virtually worthless. The origin server becomes heavily loaded with new document requests, updates and missed requests that could not be met by the lower level caches. Consequently, the benefits of cache and replication decrease. These problems become the obstacles for P2P technology to further develop in the applications of online dynamic data processing. Maintaining dynamic data consistency is difficult in P2P and it has not been well addressed.

[1]The Web server where a Web object ultimately resides is called the origin server for that object. Origin servers act as authoritative sources of Web content.

1.1 Motivation

A key issue in deploying P2P technology in managing dynamic data is the data consistency problem. The current techniques used to solve the data inconsistency problem in P2P systems are often inefficient [36]. For example, measured activity in Gnutella and Napster indicates that the median up-time for a node is 60 minutes [21]. Peers join the overlay and leave at will, and network links disconnect sometimes. Moreover, the message disseminating scope is limited by the TTL scope, which leads to a large number of peers not being reachable. Maintaining dynamic data consistency in a P2P environment is challenging. So, the goal of our work is to design a high-scalable, fault-tolerant and efficient framework to maintain cached data consistency on P2P systems. In this thesis, we focus on maintaining the consistency of dynamic data in an overlay network using cooperative peers.

1.2 Contributions

In order to handle the dynamic data applications deploying P2P technology, we aim to provide a framework to maintain dynamic data consistency in P2P systems. In summary, we seek to make the following research contributions:

- Implement an adaptive data consistency framework prototype PeerCast layered on the P2P infrastructure, BestPeer [58]. PeerCast provides graphic user-interface for peer users to set their data of interest and the associated consistency requirements. Working on the basis of peer heterogeneity in data consistency requirements, peers in PeerCast cooperate with each other to maintain cached data consistency by pushing updates.

- Provide a set of policies on PeerCast topology maintenance. For overlay construction, we provide three dissemination tree construction policies: randomized, round-robin and locality-biased. In order to address peer departure or failure problems, PeerCast provides failure detection and robust recovery techniques. Once peer users change their accessing behaviors, PeerCast also provides self-adaptive procedures to adjust the network.

- Propose heuristic approaches to optimize network resource usages by network reconfiguration and prevent the churning problem by characterizing unstable peers.

- Evaluate the above proposed methods by conducting experiments over the BestPeer infrastructure, and by simulation of a large-scale network scenario. We will evaluate our approach using real-world traces of dynamic Web data. We will analyze the impact factors of PeerCast, and we will compare PeerCast with an existing approach: Gtk-Gnutella [53]. We will demonstrate that PeerCast not only increases efficiency but also reduces the overhead for maintaining the data consistency in P2P environment.

1.3 Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents a literature review on related work. We give an overview of the P2P development history and discuss issues in current P2P research. We classify and generalize popular data consistency techniques, and present recent research results.

- Chapter 3 describes the BestPeer infrastructure, the platform on which our framework is built. We also briefly review application-level multicasting techniques. We further present distributed cooperative consistency maintenance techniques. These techniques are the building blocks of PeerCast.

- Chapter 4 provides the design issues of the PeerCast framework. This is the main part of our research. We introduce the three different policies for dissemination tree construction, and the recovery mechanisms to handle peer departure and failure. We also discuss the self-adaptive mechanism of the PeerCast, and present the source peer recovery policy.

- Chapter 5 provides heuristic policies to improve the performance and efficiency in PeerCast.

- Chapter 6 presents the methodology for conducting the experiments. We report our experiment results and analyze every impact factor to our approach in details. In the first part, we mainly examine the impact factors to our approach. In the second part, we also set up the current cache consistency protocol in P2P, Gtk-Gnutella, for comparison with our framework. We present the advantages and disadvantages of both approaches. Results of simulation experiments indicate that our approach outperforms previous research.

- Chapter 7 summarizes the contributions of our research and discusses future research.

CHAPTER 2

Background and Related Work

The Internet as originally conceived in the late 1960s was a peer-to-peer (P2P) system. Because of clients' poor bandwidth and limited computation ability, P2P models could not be developed until recent years. Hardware performance has improved greatly. Even personal computers can accomplish some heavy computing tasks. Meanwhile, broad-bandwidth connections are widely established. Current P2P applications generally would benefit from the Internet like the original network. In a P2P computing model, peers can be regarded as servants, which act as servers and clients at the same time. Peer nodes can cooperate with each other to undertake huge computation tasks by pooling resources together such as SET@home [8], or share their storage such as Napster [7], Gnutella [5] and Freenet [27]. The first generation of P2P systems is for media resources sharing among thousands of nodes. MP3 and video clips are such examples. The second generation of P2P systems is based on structured overlay which provides more powerful query routing techniques. Since P2P computing architectures provide data-centric model, which is superior to the traditional network placement model [74], it is considered to replace the client/server model in the near future. Indeed, the advent of large-scale ubiquitous computing makes P2P a natural model for interaction between devices. Over the last few years, all kinds of distributed applications, especially database applications and web services, are developed and deployed on the P2P environment.

2.1 P2P System Architectures

According to research achievements so far, P2P systems can be classified into three different categories [28]. Some P2P systems, such as Napster [7], are centralized as illustrated in Figure 2.1 (a). Centralized P2P systems have central servers playing the query routing and maintaining all the peer information. The centralized P2P systems suffer from the central server failure and the problem of scalability. Other P2P systems are decentralized and have no centralized server. Of these decentralized designs, some are structured in that they have tight coupling between the P2P network topology and the location of data, such as Chord [75], CAN [62], Pastry [68], etc. The design of these systems are based on *distributed hash table (DHT)*, which maps the data objects and peer nodes into the same identifier space. As illustrated in Figure 2.1 (c), each data object is assigned to a specific node. Other decentralized P2P systems, such as Gnutella [5] and Freenet [27], and hybrid system KaZaA [9], are unstructured with loose coupling between topology and the location of data. In these systems, peers are more autonomous and querying are normally depending on message flooding because of lack of routing information.

Both structured and unstructured P2P systems have their own advantages and shortcomings. First, since *DHT*-based structured P2P systems keep the routing table to facilitate key search, they outperform unstructured P2P systems in terms of object searching efficiency. Unstructured P2P systems have to use messages flooding to search, which leads to lowering search efficiency and wasting huge network traffic. Second, churning problem [21], which is referred to peers frequently coming in and going out, does cause more significant overhead for structured systems than unstructured P2P system does. In order to preserve the efficiency and correctness of routing, most *DHT*s require *O(logn)* repair operations after each failure (e.g., Chord and CAN). In contrast, churn causes little problem for Gnutella and other P2P systems that employ unstructured overlay networks as long as peer node doesn't become disconnected by the loss of all of its neighbors. Third, *DHT* searching techniques use the exact search key, which always map a data object into a key, consequently, peer uses get the exact data object. On the other hand, unstructured P2P systems use keywords and often can get many answers related to the keywords, which is more preferred by Internet users. Recently, some popular P2P-based sharing softwares use unstructured P2P system, such as *eDoneky* [4] and *BT* [3], etc. PeerCast is built on the decentralized unstructured P2P systems.

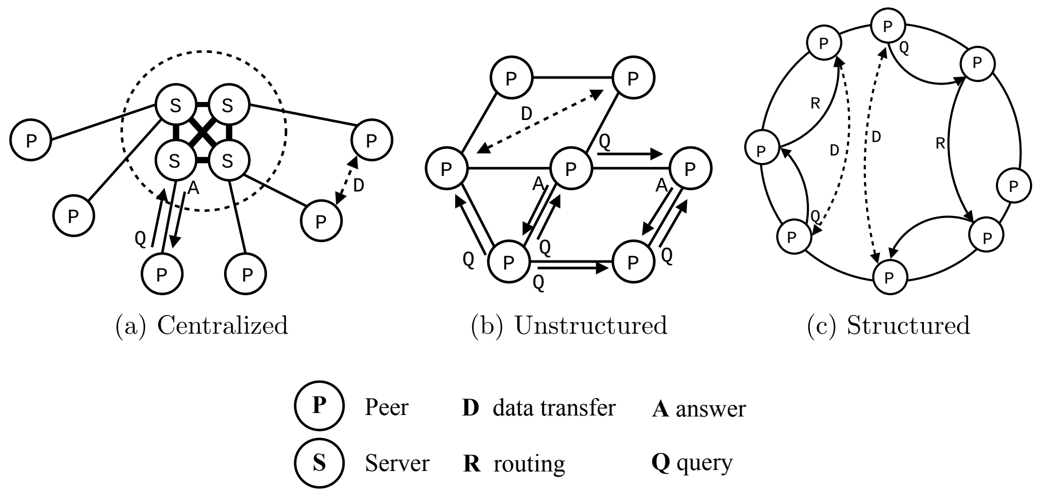


Figure 2.1: P2P Systems Classification

2.2 P2P Open Problems from Data Management Perspective

In this section, we review P2P systems from the perspective of data management. Despite of many benefits, P2P systems present several challenges that are currently obstacles to their widespread acceptance and usage. 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.

Due to the particular nature of P2P, many techniques previously developed for distributed systems of tens or hundred of servers may no longer apply. We describe some open problems of P2P research in perspective of data management, and discuss some current solutions and tough issues that need to be addressed in the near future.

- **Data Placement and Query Routing**: *Data placement* and *query routing* are two challenges to be resolved for sharing objects on any P2P systems. Data placement is the assignment of a set of objects to be stored at each peer in the network. It defines how data or metadata is distributed across the network of peers. Given the name of an object, after finding the corresponding object's location, query routing is to address how to route the query to the location.

  Napster [7] uses a centralized design to resolve these issues. A central server maintains the index for all objects in the system. New peers joining the system know the identity of the central server while the server keeps information about all the nodes and objects. After it sends the request (e.g., name of the object) to the central server, the server returns the IP addresses of the peers. The requesting peer then uses IP routing to pass the request to one of the returned peers and downloads the object directly from that peer.

Gnutella [5] follows a different approach in order to get around the problem of the centralized design. There is no centralized server in the system. Each peer in the Gnutella network knows only about its neighbors. A flooding model is used for both locating an object and routing the request through the peer network. Peers flood their requests to their neighbors, which causes a high overhead on the network as a result of flooding and the possibility of missing some requests even if the requested objects are in the system.

Peer node stores only its own collection of data either in Napster or in Gnutella. However, in another group P2P systems such as Chord [75], CAN [62], Pastry [68], data or metadata is carefully placed across nodes in a deterministic fashion. These systems are based on implementing a distributed data structure called *DHT*, which supports a hash-table-like interface for storing and retrieving objects. CAN uses a d-dimensional virtual address space for data location and routing. Each peer in the system owns a zone of the virtual space and stores the objects that are mapped into its zone. Each peer stores routing information about O(d) other peers, which is independent of the number of peers, N, in the system. Likewise, Chord assigns unqiue identifiers to both objects and peers in the system. Given the key of an object, it uses these identifiers to determine the peer responsible for storing that object. Each peer keeps routing information about O($logN$) other peers, and resolves all lookups via O($logN$) messages, where N is the number of peers in the system.

- **Schema Mediation and Data Integration**: Since peers pool their storage together, varieties of data may exist with in each peer's data repository, e.g., images library, music files, document collections or relational database tuples. In order to exchange information efficiently in a semantically meaningful way, data management and data integration tools should be provided. Although conventional schema mediation techniques have been studied for decades, unfortunately, they suffer from two significant problems. First, they require a comprehensive schema design before they can be used to store or share information. Second, they are difficult to extend because schema evolution is heavyweight and may break backwards compatibility. Due to the dynamics and large-scale nature of P2P systems, the assumption of traditional schema mediation may not be feasible in a P2P data management scenarios. Research efforts such as Piazza [40, 41], Hyperion [48] and PeerDB [59] address the problem of schema mediation in P2P data sharing systems.

Piazza [40, 41] has provided a flexible formalism, called Peer-Programming Language (PPL) for mediating between peer schemas, which deploys two commonly used formalisms: global-as-view (GAV) and local-as-view (LAV) to specify local mappings. Reformulation takes as input a peer's query and the formulas describing semantic relationships between peers, and it outputs a query that refers only to stored relations at the peers. Bernstein *et al.* [16] introduce the Local Rational 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 acquaintance, i.e., other peers. The set of acquaintances changes often due to site availability and changing usage pattern. 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 of formulas define semantic dependencies between the two databases. In Hyperion Project [48], mapping tables are proposed for data mapping in the P2P environment. Kementsietsidis *et al.* extend [16] by providing domain relation management through capabilities of inferring new mapping tables and determining consistency of mapping constraints. PeerDB [59] tackles the semantic gap by providing both Local Dictionary and Export Dictionary without a shared global schema. Export Dictionary 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. Mapping procedure is based on meta-data keywords matching by information retrieval techniques.

- **Search**: Search mechanism is a core component in P2P systems. Good search mechanism allows users to effectively locate desired data in a resource-efficient manner. To some extent, search mechanism decides the topology, data placement and message routing. Designing such a mechanism is difficult in P2P systems for several reasons: scale of the system, unreliability of individual peers, etc.

  In order for P2P systems to be useful in a wide range of applications, search mechanism must be able to support query languages of varying levels of expressiveness. The simplest form of query is an object lookup by key or identifier. Much research focuses on search techniques for keyword queries, where a few keywords can usually uniquely identify the desired file such as music or video files. If many results are returned for comprehensive keyword search, users may need the results to be ranked and filtered by relevance. Ranked search can be built on top of regular search by retrieving all results and sorting locally. Users may sometimes be interested in knowing aggregate properties of the system or data collection as a whole (e.g., COUNT, MEDIAN, etc.), rather than locating specific data. Furthermore, SQL defined over the P2P data storage could also be needed.

Thus far, research in search mechanism has focused on answering simple queries, such as key lookups. Current research on supporting complex query in P2P systems is very preliminary. PIER project [43] has supported a subset of SQL over a P2P framework, but they reported significant performance "hotspots" in their preliminary implementation. Further research is needed to extend these techniques into more expressive aggregates.

- **Replication and Caching**: Replication and caching are well-understood techniques deployed in distributed systems. The objective of them is to minimize the overall query execution time in a huge pooling data storage. In a P2P scenario, the objective can either be achieved through minimizing the number of routing hops or maximizing the replication of objects.

  Cohen *et al.* [28] have evaluated different replication strategies and revealed the optimal strategy in unstructured P2P network from a theoretical perspective. The problem statement of replication policy in P2P network is as follows. The network consists of n nodes, each with capacity p which is the number of copies/keys that the node can hold. Let $R = np$ denote the total capacity of the system. There are m distinct data items in the system. The normalized vector of query rates takes the form $\mathbf{q} = q_1 \geqslant q_2 \geqslant \ldots \geqslant q_m$ with $\sum q_i = 1$. The query rate q_i is the fraction of all queries that are issued for the ith item. An *allocation* is a mapping of items to the number of copies of that item. Let r_i denote the number of copies of the ith item, and let $p_i = r_i/R$ be the fraction of the total system capacity allotted to item i: $\sum_{i=1}^{m} r_i = R$. The allocation is represented by the vector $\mathbf{p} = (p_1, p_2, \ldots, p_m)$. A replication *strategy* is a mapping from the query rate distribution $\mathbf{q}$ to the allocation $\mathbf{p}$. In Gnutella [5], when a search is successful, the object is stored at the requester node only. The replication strategy is called "owner replication". Freenet [27] provides a different replication strategy. When a search succeeds, the object is stored at all nodes along the path from the requester node to the provider node. So, they can reply immediately to any further request for that particular object. This strategy is named as "path replication". Each Freenet node maintains a stack. Objects that are requested more often are moved up in the stack, displacing the less requested ones. PAST [69] has been designed to store multiple replicas of files and cache additional copies of popular data objects. PAST controls the distribution of per-node storage capacities by comparing the advertised storage capacity of a newly joining node with the average storage capacity of nodes. It maintains the invariant that k copies of each inserted file are maintained on different nodes. Highly popular files may demand many more than k replicas in order to sustain its lookup load while minimizing client latency and network traffic. In order to balance the remaining free storage space among the nodes, PAST provides replica diversion policy. If node A cannot accommodate a copy locally, it considers replica diversion. For this purpose, Node A chooses node B in its leaf set[1] that is not among the k closest and does not already hold a diverted replica of the file. Node A asks node B to store a copy on its behalf, then enters an entry for the file in its table with a pointer to node B. OceanStore [51] is a utility infrastructure designed to provide continuous access to data storage scaled to billions of users. Objects are replicated and stored on mul-

[1]In addition to the routing table, each node in PAST maintains IP addresses for the nodes in its leaf set. The leaf set is the set of nodes with nodeIDs partially similar to present node's nodeID.

tiple servers. A given replica is independent of the server on which it resides at any one time; thus they are referred as floating replicas. OceanStore provides two location policies. A fast, probabilistic algorithm attempts to find the object near the requesting machine. If the probabilistic algorithm fails, location is left to a slower, deterministic algorithm.

Existing P2P systems often utilize the replication and caching techniques to promise availability in the presence of network partitions and durability against failure and attack. However, high degree of replication makes update much harder, and increases the retrieval complexity. Maintaining consistency over replicated objects is a difficult problem in P2P network. A typical solution, which is quite acceptable for P2P scenario, is to have each object be owned by a single master, which is solely responsible for its freshness [36].

- **Data Consistency**: Replicating and caching create numerous copies of data objects scattered throughout the P2P overlay network. They promise high data availability in P2P systems, minimize response latency to query and reduce the network traffic. Unfortunately, they introduce data inconsistency problem. To achieve data freshness and update consistency in distributed systems, there are many possible ways of propagating updates from the data origins to intermediate nodes that have materialized views of this data. Most of the previous consistency work has focused on conventional distributed systems, such as Web proxy caching, Content Delivery Networks (CDNs), and mobile computing environment. Existing approaches are inefficient in P2P systems due to the unreliable nature of peers, high autonomy and large-scale of the P2P network.

  Some possible solutions would be invalidation messages pushed by the server or client-initiated validation messages; however, both of these incur overhead that limits scalability. Another approach is a timeout/expiration-based protocol, as employed by DNS and web caches. This approach has lower overhead, however, it only guarantees looser freshness and consistency. Data consistency maintenance techniques in current P2P systems are inefficient [36]. We will present a detailed survey in this topic in next section.

We have generalized current open problems in P2P research area. Our work focuses on presenting a potential solution to data consistency maintenance in P2P caching systems. In the following section, we survey various approaches for data consistency in distributed systems and classify them by their dominant way of solving.

2.3 Data Consistency Schemes Taxonomies

Data consistency problems exist in any system that uses some form of cache to speed up accesses. Data consistency protocols have been studied in computer architecture, distributed file systems, network and distributed database systems. The consistency problems are slightly different in the four contexts. In particular, data consistency is a tradeoff between performance and precision in distributed systems. When data is replicated or cached, system performance benefits. However, the multiple copies of the same information maintained at the different sites can become inconsistent and stale if the objects are updated at the origin servers. Without special mechanisms to enforce the freshness of the cached data management, distributed systems would continue using the stale cached copy of objects to query results.

2.3.1 Consistency Models

Traditionally, consistency has been discussed in the context of read and write operations on shared data, available by means of distributed shared memory, a distributed shared database, or a distributed file system. Replicas may be physically distributed across multiple machines. *Strong consistency* is defined as a model in which after a write operation completes, no stale copy of the modified document will ever be returned to user [54]. On the other hand, *weak consistency* is defined as the consistency model in which a stale document might be returned to the user. In such a manner, data freshness can not be guaranteed. For strong consistency, it is unnecessarily restrictive for many applications. In some cases, providing strong consistency imposes performance overheads and limits system availability. Although queries executed over cached data can get an answer very quickly in weak consistency, usually no guarantees are given as to exactly how imprecise the answer is. So, the user is left to guess the degree of imprecision based on knowledge of data stability or how recently caches were updated. Weak consistency is not always satisfactory.

Thus, a variety of optimistic consistency models have been proposed for applications that can tolerate relaxed consistency. In TRAPP [60], users supply a quantitative precision constraint to balance the tradeoff between precision and performance. Yu *et al.* have designed a system, TACT [83], that can support application-specific consistency models. The need for differentiating models stems from tradeoffs among performance, availability, and consistency. Consistency is defined in terms of three continuous parameters: the number of writes that a replica can permit not to have seen, the number of writes that can be performed locally before update propagation takes place, and the time allowed to delay update propagation. Deolasee *et al.* [30] propose dissemination of dynamic Web data techniques tailor dissemination of data from servers to clients based on clients' coherency requirement. Each user specifies a temporal coherency requirement for each cached item of interest.

2.3.2 Update Propagation

In this subsection, we discuss different ways of propagating updates to replicas, which are independent of the consistency model that is to be supported. There are three design issues about update propagation.

The first design issue concerns what is actually to be propagated. Basically, there are three possibilities:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.

Propagating a notification is what *invalidation protocols* [77] do. In an invalidation protocol, other copies are informed that an update has taken place and that the data they contain are no longer valid. Since no more than a notification is propagated, whenever an operation on an invalidated copy is requested, that copy generally needs to be updated first. The main advantage of invalidation protocols is that they use little network bandwidth. The only information that needs to be transferred is a specification of which data are no longer valid. Such protocols generally work best when there are many update operations compared to read operations.

Transferring the modified data among replicas is the second alternative, and is useful when the read-to-write ratio is relatively high. In that case, the probability that an update will be effective is high in the sense that the modified data will be read before the next update takes place.

The third approach is not to transfer any data modifications at all, but to tell each replica which update operation it should perform. This approach assumes that each replica is represented by a process capable of "actively" keeping its associated data up to date by performing operations [71]. The main benefit of active replication is that updates can often be propagated at minimal bandwidth costs, provided the size of the parameters associated with an operation are relatively small. On the other hand, more processing power may be required by each replica, especially when operations are relatively complex.

The second design issue is whether updates are pulled or pushed. In a *push-based approach*, updates are propagated to other replicas without those replicas even asking for the updates. Push-based approaches are often used between permanent and server-initiated replicas, but can also be used to push updates to client caches. Server-based protocols are applied when replicas generally need to maintain a relatively high degree of consistency. Push-based protocols are efficient in the sense that every pushed update can be expected to be of use for one or more readers. In addition, push-based protocols make consistent data immediately available when asked for.

In contrast, in a *pull-based approach*, a server or client requests another server to send it any updates it has at that moment. Pull-based protocols, also called client-initiated protocols, are often used by client caches. For example, a common strategy applied to Web caches is first to check whether cached data items are still up to date. When a cache receives a request for items that are still locally available, the cache checks with the original Web server whether those data items have been modified since they were cached. In the case of a modification, the modified data

Table 2.1: Main Issues Comparison between Push and Pull

Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

are first transferred to the cache, and then returned to the requesting client. If no modifications take place, the cached data are returned. In other words, the client polls the server to see whether an update is needed. Pull-based approach is efficient when the read-to-update ratio is relatively low. This is often the case with client caches, which have only one client. The main drawback of a pull-based strategy in comparison to a push-based approach is that the response time increases in the case of a cache miss.

When comparing push-based and pull-based solutions, there are a number of tradeoffs to be made, as shown in Table 2.1. For push-based protocols, apart from the fact that stateful servers are often less fault tolerant, the server needs to keep tracks of all client caches. Keeping track of all client caches may introduce a considerable overhead at the server. For example, in a push-based approach, a Web server may easily need to keep track of tens of thousands of client caches. Each time a Web page is updated, the server will need to go through its list of client caches holding a copy of that page, and subsequently propagate the update.

In addition, the messages that need to be sent between a client and the server also differ. In a push-based approach, the only communication is that the server sends updates to each client. When updates are only informed by invalidations, additional communication is needed by a client to fetch the modified data. In a pull-based approach, a client will have to poll the server, and, if necessary, fetch the modified data.

Finally, the response time at the client is also different. When a server pushes modified data to the client caches, it is clear that the response time at the client side is zero. When invalidations are pushed, the response time is the same as in the pull-based approach, and is determined by the time it takes to fetch the modified data from the server.

A hybrid form of pull and push propagation is lease. Leases are originally introduced by Gray and Cheriton [35]. They provide a convenient mechanism for dynamically switching between a push-based and pull-based strategy. A lease is a promise by the server that it will push updates to the client for a specified time. When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary.

The third design issue is to decide whether unicasting or multicasting should be used. In unicast communication, when a server sends its update to N other servers, it does so by sending N separate messages, one to each server. With multicasting, the underlying network takes care of sending a message efficiently to multiple receivers. In many cases, it is cheaper to use available multicasting facilities. An extreme situation is when all replicas are located in the same local-area network and that hardware broadcasting is available. In that case, broadcasting or multicasting a message is no more expensive than a single point-to-point message. Unicasting updates would then be less efficient.

Multicasting can often be efficiently combined with a push-based approach to propagating updates. In that case, a server that decides to push its updates to a number of other servers simply uses a single multicasting group to send its updates. In contrast, with a pull-based approach, it is generally only a single client or server that requests its copy to be updated. In that case, unicasting may be the most efficient solution.

Table 2.2: Approaches Classification by Consistency Model

Consistency degree	Approaches
strong	Invalidation, Continuous Multicast Push, Leases
weak	Time-To-Live, Validation
demand-driven	Heuristic Approaches, Data Recharging

Table 2.3: Approaches Classification by Update Propagation Way

Dissemination way	Approaches
push	Continuous Multicast Push, Data Recharging, Invalidation
pull	Time-To-Live, Validation
hybrid	Heuristic Approaches, Leases

2.3.3 Data Consistency Protocols

Maintaining consistency techniques have been studied for decades in distributed systems, such as distributed file systems [18], Web proxy caching or CDNs [56, 50, 39, 54] and mobile computing environments [15]. There are a range of techniques which were proposed to solve the problem, from simple approaches like *TTL* to complex approaches like cache profile language to specify the users' demands [24]. These techniques can be classified into three types as listed in Table 2.2 according to the consistency models. Likewise, they also can be classified into three types according to update propagation as listed in Table 2.3. We survey the popular consistency protocols as follows.

Time-To-Live: Time-To-Live (TTL) is a simple way to achieve some limited degree of data consistency. It has been widely used in Web pages. Explicit TTLs must be specified by Web developers as part of object creation, such as *expires* and *cache-control:max-age* headers. It cannot guarantee high degree of consistency. However, it is the least cost method. Most current caching systems use an adaptive heuristic TTL, which is based on the assumption that the longer a file has been unchanged, the longer it tends to remain unchanged in the future [32].

Cache Validation: Cache validation, also known as *client polling*, refers to the approach where clients verify the validity of their cached objects with the origin server. Netscape Navigator 1.1 implements the validation mechanism where the server sends down a chunk of data, including a directive (in the HTTP response or the document header) that says "reload this data in 5 seconds", or "go load this other URL in 10 seconds". After the specified amount of time has elapsed, the client does what it was told - either reloading the current data or getting new data [1].

A key issue for cache validation is when to send validation messages. The trade-off is among the degree of consistency, message consumption and latency overhead. The more frequent the validation messages are, the lower the probability of delivering stale content from the cache is, but the higher the message and latency overhead for validating unchanged objects are. A problem of validation is the message and latency overload. The extreme options are to validate every access, which provides strong consistency at the expense of a large number of unnecessary validation messages, or never to validate, which has zero messages overhead but a high probability of stale delivery. Therefore, validation usually provides a weak consistency because objects are typically validated only periodically.

Cache Invalidation: Cache invalidation protocols [77] are required when weak consistency is not sufficient. Many distributed systems rely on invalidation protocols to ensure that cached copies never become stale. With invalidation, the origin server notifies clients which of their cached objects have been modified. The clients mark those objects as invalid and assume that any objects they cache are always valid unless they are marked otherwise. HTTP1.1 allows an origin server to invalidate an object cached by proxy by submitting to the proxy a PUT, POST or DELETE request for the object. There has been no accepted standard for a protocol that would allow invalidation of browser caches. Open protocols for Web cache invalidation in the Internet are being actively discussed in the IETF [11].

Cache invalidation protocols are often expensive, in which two interdependent issues must be addressed: the client list problem and the delayed updated dilemma. The client list problem is twofold. First, it requires the server to record prior interactions with all clients. Second, it is unclear if the server can ever trim the lists because expecting clients to notify servers when they drop objects from their caches is generally unreasonable. The delayed updated dilemma is how the server should deal with an unreachable client that needs to be invalidated. That client will not receive the invalidation message and will continue using its cached content regardless of any updates. In summary, invalidation protocols can provide strong consistency in the absence of Internet disconnections. However, they introduce scalability problems or necessitating hierarchical caching.

Volume Lease Protocols: Lease protocols are proposed to address the limitation of invalidation protocols. With leases, the server must keep a client in the object client list only until the client lease expires [32]. Further, an update can be delayed by an unreachable client by at most the duration of its lease. Whenever a cache stores a data object, it requires a lease from the server. Whenever the object changes, the server notifies all caches who hold a valid lease of it; the invalidation contract applies only while the leases is valid.

Instead of maintaining separately for individual objects, volume lease protocols are used [50, 81, 82]. Several objects are combined into a volume and maintained consistency at the granularity of entire volumes. Thus, volume lease approach combines features of the validation (after the lease expires) and invalidation approaches (during lease period).

Server Push Protocols: Server push protocols are proposed to reduce the workload of origin servers. Netscape has recently added push capability to its Navigator browser specifically for dynamic documents [1]. Server sends down a chunk of data; the browser displays the data but leaves the connection open. Whenever the server desires, it continues to send more data and the browser displays it, leaving the connection open. In server push, a HTTP connection is held open for an indefinite period of time (until the server knows it is done after sending data to the client and a terminator, or until the client interrupts the connection). Server push is accomplished by using a variant of the MIME message format "multipart/x-mixed-replace". The "replace" indicates that each new data block will cause the previous data block to be replaced – that is, new data will be displayed instead of (not in addition to) old data.

The key to the use of this technique is that the server does not push the whole "multipart/x-mixed-replace" message down all at once but rather sends down each successive data block whenever it sees fit. The HTTP connection stays open all the time, and the server pushes down new data blocks as rapidly or as infrequently as it wants.

Continuous Multicast Push: For popular Web documents that rarely change, a caching hierarchy seems the best solution. Hit rates close to 50% [12] can be achieved, and the bandwidth usage and latency to the receivers are reduced. However, there are certain dynamic Web documents that change frequently. The root cache is heavily loaded because it deals with new document requests, updates, and missed requests that are not fulfilled by the lower level caches.

Continuous Multicast Push (CMP) is a mechanism for reducing the bandwidth usage and latency to the receivers on the Internet for very popular documents that change very frequently [65, 66]. CMP takes place at the transport layer with reliability and congestion control ensured by the end systems (server and clients).

Server housing a popular and frequently-changing object continuously multicasts the latest version of the object on a multicast address. Clients tune into the multicast group for the time required to reliably receive the document and then leave the multicast group. Due to varying nature of the different Web documents, there is room for both caching and continuous multicast distribution.

CMP does not suffer problems of overloaded servers or caches. It scales very well with the number of receivers. Receivers obtain at any moment the last available update without incurring on the overhead of checking for the updated document on all the cache levels. The multicast distribution uses bandwidth efficiently by sharing all common paths between the source and the receivers. However, some additional mechanisms should be well studied to make CMP a viable service. Servers should map the document's name into a multicast address. It should provide the multicast capable routers that maintain state information for each active multicast group. The overhead is high due to join and prune messages needed for the multicast tree to grow and shrink.

Hybrid and Heuristic Approaches: Hybrid and heuristic approaches are proposed to combine the advantages of existing methods and overcome their limitations. It is necessary to provide a heuristic decision model to adaptively select the optimal method. Due to these approaches' adaptable capacities, they are self-configurable to different scenarios without administrator configuration, and guarantee a relatively low response delay and minimize the network traffic in comparison to previous methods. For example, SPREAD [67] was designed for distributing and maintaining up-to-date Web content that simultaneously employs three different mechanisms: client validation, server invalidation, and replication. Proxies within SPREAD self-configure themselves to form scalable distribution hierarchies that connect the origin servers of content providers to clients. Each proxy autonomously decides on the best mechanism based on the object's popularity and modification rates. Requests and subscriptions propagate from edge proxies to the origin server through a chain of intermediate proxies. The core heuristic model of SPREAD is for proxies to estimate update rate and determine which mechanism to use based on local observations. Observing that lease duration is the critical parameter that determines the efficiency of the lease protocols, Duvvuri *et al.* [32] propose adaptive leases to balance the tradeoffs between large state space and control message overhead. The heuristic mechanism uses constraints on the state space overhead and the control message overhead to compute an appropriate lease duration adaptively. Deolasee *et al.* [30] combine push and pull techniques to achieve the best features of both approaches. *PoP* and *PaP* algorithms are introduced to tune according to the client requirements and conditions at the server/proxy.

Data Recharging: Data recharging [17, 52, 24] is similar to power recharging. Data recharging techniques make use of a centralized data server to disseminate the data updates to different users, meanwhile, a set of rich profile expressions are provided to describe the needs of the receivers' data consistency demands. The mechanism is totally driven by the requirement of users. Application-level knowledge is expressed as profiles [23] to manage the contents and freshness of caches. Although making delivery decision requires complex computation of profiles and scheduling, data recharging can allocate network bandwidth economically and save numerous useless data delivery. The data updates propagation is also delivered according to the priority of the user's requirements.

In summary, all data consistency proposals attempt to achieve some degree of consistency. The approach taken to achieve consistency depends greatly on certain scenario. Researchers adapt and extend some traditional techniques to meet certain new requirements. For example, cache invalidation report is used as an extension of cache invalidation protocol in mobile computing scenario [15]. Considering a P2P environment, where each peer caches certain data objects, in which frequently-changing data objects are suffered from consistency. Because of the scale of the network, unreliable nature of peers and lack of global topology information, maintaining cached data consistency in each peer node is more challenging than the work in conventional client-server model system. Many techniques previously developed for distributed systems will be inefficient or no longer be applicable. Cache validation is not efficient in millions of peers scenario, which will cost huge network bandwidth. The key limitation of client polling is that it is hard to predict the update rate of the cached objects. Cache invalidation suffers from unreachable client problem. Once disconnected from the network[2], the invalidation protocol does not work any more. CMP method relies on reliable network connection and nodes stability, and also results in significant relative penalty delay when systems scale. However, nodes are natively transient in P2P systems, which degrades the performance of data pushing techniques. CMP requires that the network is multicast capable. Only a few network providers can offer it as a service. Data recharging techniques provide more user-interactive procedure for data consistency to reduce the unnecessary network cost. It needs a centralized computation to maintain the requirements of users. New techniques are required to meet these challenges. Since data consistency is a general topic in data management, our design is built upon prior research and we adapt and extend them into P2P environment.

[2]Removing a small portion of peer nodes is possible to fragment the entire network into many isolated pieces [49].

2.4 Existing Consistency Work in P2P

Most consistency research work has been done on Web proxy caching and content distribution network scenarios [30, 32, 39, 50, 54, 67]. To our best knowledge, recent research which is related to our work is as follows:

Shal *et al.* propose hierarchical repositories architecture and the corresponding dynamic data dissemination techniques [73]. In their setting, each repository registers into the network with specific consistency requirement. Repositories in an upper level have more stringent consistency than those in a lower level. Thus, repositories in an upper level can feed the lower ones by pushing updates of data items. Client users can connect to different repositories according to their data of interest and consistency requirements. In this way, origin server workload is proportioned by other repositories in the overlay.

Shal *et al.* [72] present more techniques for creating a resilient and efficient content distribution network for dynamically changing streaming data. Their dissemination tree construction is better than that in [73]. To achieve fault tolerance, each node maintains two parents: one primary and one backup, where the backup serves the child with less than the request coherency.

Shal *et al.*'s work provides fine-grained data consistency and an intelligent filtering and dissemination techniques based on each repositories coherency requirement. However, their solution is not adequate in P2P environment. First, peers are autonomous. They come and go unexpectedly. The architecture cannot tackle the transient nature of the peers. Furthermore, peer users change their consistency requirements and data of interest freely. Their dissemination overlay provides no adaptive disseminating mechanisms. Second, their work does not consider the resource usage and network locality, which are key issues in large-scale P2P network.

Nodes in network proximity are more prone to cooperating with each other and bringing more benefits. Third, peer nodes in a real system have heterogenous capacity from mobile PDAs to powerful workstations. Shal *et al.*'s work never makes use of client peer powerful capacity.

Chen *et al.* [22] propose a dynamic replica placement for scalable content delivery to address the problem on how to achieve the maximum benefits by placing the minimum replicas in Tapstry [84] while satisfying all the client peer's query latency. Druschel *et al.* [31] state that an adaptive cache coherence system is required. They assume that replicas are stable content delivery servers, which are placed in the Tapstry. Clients are normal peers of the Tapstry. The replicas are formed as an application-level multicast tree piggyback on the structured routing techniques, and data consistency of replica are maintained using heuristic method proposed in SPREAD [67].

Their work is layered on Tapstry, a structured P2P infrastructure. Their goal is to place the minimal number of replicas to satisfy the maximum client peers querying latency. Therefore, their work is just to maintain all the data of replicas consistency, normal client peers should query those replica servers to get the newly updated data. In other words, their solution is not really for peer level, but in content distributed network.

Lan *et al.* [53] focus on the problem of consistency maintenance among multiple replicas in the presence of updates. They propose Gtk-Gnutella protocol, which is built over Gnutella-like P2P system. Gtk-Gnutella presents three different approaches: *push*, *adaptive pull* and *push combined with adaptive pull*. They assume that only server peers have the authority to modify the file objects, which may make all the other replicas inconsistent. To maintain cached data consistency, push-based mechanism lets server peers send invalidation messages to inform the client peers using flooding when updating the source data. The main advantage of this push-based approach is its simplicity and stateless nature. Since invalidation messages are propagated via flooding, the server peer does not need to maintain a list of client peers which hold a replica of the file. But, push is limited by the TTL value reachable scope and network disconnection. Therefore, the push mechanism is inadequate in P2P scenario. Adaptive pull-based approach puts the burden of consistency maintenance on individual peers. It is implemented like a client/server system such as the Web. Pull is more resilient to dynamic peer join and departure. However, it only guarantees weak consistency. The adaptive time-to-refresh computation also cannot guarantee good prediction of the updates frequency. So, Gtk-Gnutella protocol has provided a heuristic mechanism called hybrid push and adaptive pull technique. It combines the advantages of these two approaches. The hybrid method provides satisfactory cached data fidelity.

To our best knowledge, Lan *et al.*'s work is the first one to address consistency of the data cached by peer nodes. The work has some limitations. First, their approach is heavily relying on the traditional consistency techniques. Second, their solution never considers the network proximity. It results in numerous network traffic waste to disseminate invalidation messages regardless of file object popularity. Third, their three proposed approaches only provide strong consistency. Unfortunately, it is not necessary and practical to guarantee strong consistency in large-scale P2P network. Due to the centralized design, the origin server becomes the bottleneck to limit system scalability. Their work is the earliest work directly dealing with the local cached data in P2P environment. Although well-designed, there still remains some space to further improve.

P2P technology has provided numerous cooperative models in previous work. For example, CQ-Buddy framework has been designed for supporting continuous query processing based on P2P technology [57]. Working on the basis of peer heterogeneity, peers in CQ-Buddy network help one another by sharing query workload and providing data. The framework presents two strategies, *SELF-HELP*, and *BUDDY-HELP*, that allow for the grouping and sharing of multiple continuous queries amongst peers. Weaker peers (e.g., PDAs, mobile devices) are helped by stronger peers for complex queries processing. CQ-Buddy is distributed and highly scalable as there is no single-point failure and single-source bottleneck. CQ-Buddy has indicated that cooperation is essential to achieve scalability and extensibility.

Our work further extends the previous work. In particular, our work differs from previous research work in three aspects. First, each peer manages its local cache data, which is used to raise query performance. The data consistency is on the peer node granularity. Peers maintain the cached data consistency with cooperation among each other. Peers possessing data item with high stringent consistency can push data updates to the peers with lower requirements based on their demands. Thus, idle bandwidth is fully utilized. Second, we provide an adaptive dissemination overlay comprising of numerous dissemination trees. Source peers and client peers cooperate with each other to choose the optimal parent peer for new coming client peers, while taking peer workload and network locality into account. Moreover, the overlay can adjust itself according to consistency requirement variation of peer users. The initially-setup dissemination tree can adapt to demands without administration. Third, we introduce redundancy techniques to backup potential parent peers for each client peer when it joins the consistency overlay without any manual administration. Peer departure or failure can be repaired in time with robust recovery techniques. Differently, backup parents can also contribute to the self-adaptive procedure.

2.5 Summary

In this chapter, we have provided a literature review on popular P2P architecture development. We classify P2P architectures into three categories, and introduce the corresponding techniques. In addition, we state current P2P research open problems and existing solutions.

In particular, we have outlined the data consistency strategies which have been studied in distributed systems. We have analyzed the advantages and drawbacks of each strategy. Meanwhile, we present the design challenges of data consistency in a P2P environment. In the end, we examine the recent consistency related work in P2P research.

CHAPTER 3

PeerCast Building Blocks

PeerCast is the framework built on BestPeer [2], therefore we briefly outline some features about the BestPeer platform. We just mention some key related components here. Readers can refer to [58] for more details. In addition, PeerCast borrows and extends the idea of application-level data multicast techniques. We discuss and analyze these techniques as PeerCast building blocks.

3.1 BestPeer Platform

BestPeer is a generic P2P system designed to serve as a platform on which P2P application can be developed easily and efficiently. Figure 3.1 illustrates the BestPeer architecture. The network consists of two types of entities: a large number of computers (*nodes*), and a relatively fewer number of location-independent global name lookup (*LIGLO*) servers. The node registers with a LIGLO server when entering the BestPeer system. The LIGLO server will issue the node with a global

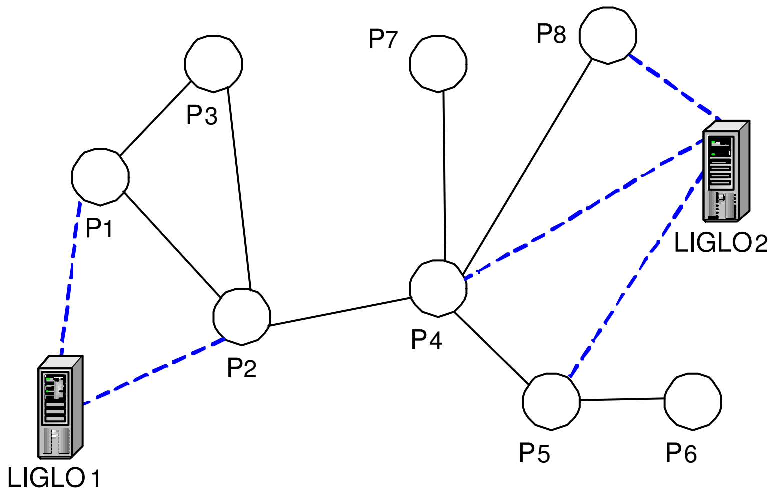


Figure 3.1: BestPeer Network Architecture

and unique identifier, which we shall refer to as BestPeerID (*BPID*). $BPID$ serves to uniquely recognize this 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 identifier for the node assigned by the LIGLO server. BestPeer properties circumvent the dynamic IP problem. We use *BPID* to identify nodes in our system implementation and simulation.

Each participating node runs the BestPeer (Java-based) software and is able to communicate or share resources with any other node in the BestPeer network. In order to support PeerCast efficiently, we implement the *push* capability on each peer node by the pull way, which means that a parent peer initiates to invoke the peer to download the data from the parent peer by sending a message to the child peer.

We further divide the data management into static data set and dynamic data set to integrate the dynamic data applications with BestPeer. Date items in dynamic data set will be maintained with consistency using peer cooperation. We also incorporate management mechanisms into BestPeer infrastructure with an original function.

3.2 Application-Level Data Multicast

PeerCast maintains dynamic data consistency using application-level multicast layered on top of the BestPeer. Application-level multicast data delivering techniques, a.k.a., end-system multicasting, are widely investigated in the computer network communication area [26, 14, 85, 20, 45] since IP multicast does not pertain to scalability. Peer nodes participating in BestPeer implement their own multicast trees. Multicast trees are built to efficiently deliver data end-to-end.

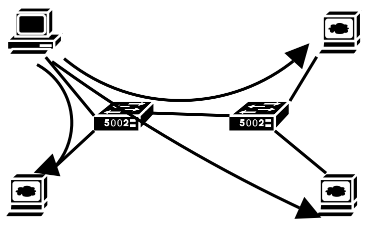
Multimedia conferencing, video-on-demand applications, etc. are based on application-level multicast, which can outperform unicast delivery. As shown in Figure 3.2, (b) reduces network traffic cost as compared with (a), and physical link stress is re-allocated to load balanced in (b). The link between two routers in (a) experiences higher stress, which will incur a larger end-to-end delay. The deployment of application-level multicasting can reduce network traffic cost and be easily implemented on systems since its deployment does not need to consider the lower level physical network topology. In addition, application-level multicasting techniques optimize the efficiency of the overlay by adapting to network dynamics and by considering application-level performance.

Classical cases for end-system multicast such as Narada [26] and Scattercast [19], etc., build application-level meshes formed by connections among a subset of node pairs. Unfortunately, these system protocols are clearly not designed for a large-scale network. Node arrival and departure information is disseminated to all members of the mesh to guarantee the quality of the mesh. Conventional application multicast tree design is only considered in a small scale overlay network.

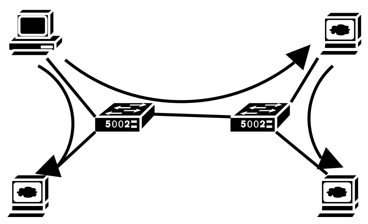
Multicast delivery suffers much because most of them assume a stable network. Some of the conventional application multicast tree designs even lack of scalability.

In the last few years, the P2P paradigm has attracted the attention of numerous researchers. Two main categories of research can be identified: research on protocols and algorithms (such as searching and replication), and research on building P2P systems. Significant research effort has addressed the problem of efficiently streaming multimedia, both live and on demand, over the best-effort Internet. Many systems rely on application level multicast to overcome the limited deployment of network level multicast. Each system has its own protocols for building and maintaining the multicast tree. For example, NICE [14] uses a multi-layer hierarchical distribution trees to scale to a large number of peers. However, NICE is not optimized for a high rate of node churn. The disruptions in dissemination tree due to node failure can take up 30 seconds to heal.

The design of PeerCast framework borrows the idea of application-level multicast techniques. In order to achieve scalability, PeerCast uses logical links and soft-state mechanism. "Heartbeat" messages are periodically sent to detect nodes failure instead of physical link meshes. Data delivery adopts demand-driven strategy based on the peer user's interests to minimize the link workload. As for frequent delivery links, they can upgrade into physical links by network reconfigure techniques. When constructing the disseminating tree, the overlay can return more than one backup parent peer node to a newcome client peer counteracting the node churn and link failures.



(a) Unicast Data Delivery



(b) Multicast Data Delivery

Figure 3.2: Multicast Vs. Unicast

3.3 Maintaining Consistency in Distributed Cooperative Manner

Our framework aims to distribute the server workload and high scalability while retaining efficient and balanced resource consumption of the underlying infrastructure.

Some previous work has proposed dynamic consistency. Out-of-date cached data are permitted. Bound cache and stale cache are proposed to query tolerance by Huang *et al.* [42]. TRAPP [60] supports users to supply a quantitative precision constraint along with each query. For example, those short-term investment speculative dealers need every minute stock price update while long-term investors or casual observers do not need so stringent consistency requirement. Different client users may have same data of interest but different consistency requirements. The requirements can be specified in units of time (e.g., the item should never be out-of-sync by more than 5 minutes), value (e.g., the stock price should never be out-of-sync by more than a dollar) or version (e.g., update times). Thus, we could use a cooperative manner on the basis of peer heterogeneity in consistency requirements for different data objects. However, the key issue is how and when the data updates are disseminated between peers in a distributed cooperative manner.

As illustrated in Figure 3.3, we show cooperation among peers: each peer pushes

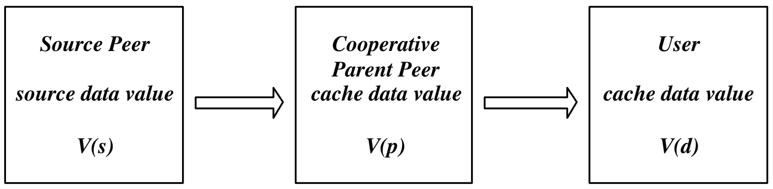


Figure 3.3: Distributed Cooperative Consistency Maintenance

updates of data items to other peers, which helps reduce system-wide communication and computation overheads for cache consistency maintenance. In a distributed cooperative approach, dependant clients may not get the knowledge of the source peer updating dynamic data items. The set of updates received by dependant peers is a subset of that received at its helper peer (i.e., peer in an upper level) which in turn is a subset of unique data values at the source peer.

To maintain consistency for each dynamic data within individual peers, the equation

$$|v(s) - v(l)| \leqslant c^l \tag{3.1}$$

should be held, where $v(s)$ represents the value of dynamic data at origin server, $v(l)$ represents the value of cached data in each client peer, c^l is the consistency requirement for individual peers. As in Figure 3.3, source peer S connects with intermediate peer P, and client peer D connects with P only. Let c^p and c^d denote the consistency requirements of data item d at peers P and D, respectively. If P serves D,

$$c^p \leq c^d \tag{3.2}$$

Thus, to effectively disseminate updates, we require that the consistency requirement at a repository should be at least as stringent as those of its dependents.

Let v_i^s, v_{i+1}^s, v_{i+2}^s ... denote a sequence of updates to v at the source peer S. Let v_j^p, v_{j+1}^p, v_{j+2}^p ... denote the updates received by intermediate peer P and v_k^d, v_{k+1}^d, v_{k+2}^d ... denote the updates received by the dependent peer D. Since $c^p \leq c^d$, the set of updates received by D is a subset of that received at P, which in turn is a subset of unique data values at the source. Specifically, an update u_j^p received by P is forwarded to D if

$$|u_j^p - v_k^d| \geqslant c^d \tag{3.3}$$

where u_k^q denotes the previous update received by D. Intuitively, Equation (3.3) indicates that any updates that violate the consistency requirements of D are forwarded to D. Note that this is a necessary but not sufficient condition for maintaining consistency at D. For instance, P takes 0.3 as c^p to certain dynamic data o, and D takes 0.5 as c^d. At some point of time, the value of o at source S is 1.4, and P has cached o, whose value is 1.4, and D keeps the older version of o, whose value is 1. A subsequent update to o makes an increase in value to 1.5 at S. Consequently, for P, the update does not result in a violation, Equation (3.1) holds. Its cached data still meet the requirement. For D, Equation (3.1) does not hold. However, because D does not have any knowledge of the source update, it has no knowledge of the source update, D has no knowledge that its cached data has been out of the bound of c^d. D will continue using the stale data. Therefore, the missing update problem appears.

There are several approaches to address this issue. In our setting, we adopt the consistency reassignment proposed in [73]. In order to prevent the missing update problem, each parent peer should forward the update to his children, if $|v_j^p - v_k^d| \geqslant c^d - c^p$, which is equivalent to raising the consistency requirement of D. Although $c^d - c^p$ is less than the original consistency requirement of a dependant client, the condition can provide 100% updates delivery. In the previous example, P will forward the 1.4 to D. When the source value increases to 1.5, the consistency requirement is still met in D.

CHAPTER 4

PeerCast Framework Design

This chapter gives an overview of the architecture of PeerCast framework. We present different policies for data dissemination tree construction, self-adaptive procedure, the fault-tolerance mechanisms and the strategies to address peer leave and recovery problems. In next chapter, we discuss the PeerCast enhancement issues and performance improvement. We report the results by the simulation experiments to show the efficiency of our approach in Chapter 6.

4.1 Motivation Revisit

We consider the following objectives during the design and implementation of PeerCast: scalable, self-adaptive, fault-tolerant and efficient, i.e., low latency and small network traffic to achieve high fidelity. In order to avoid the single-point failure problem, we reduce the workload of origin servers as much as possible with peer cooperation on the basis of peer heterogeneity in data consistency requirements.

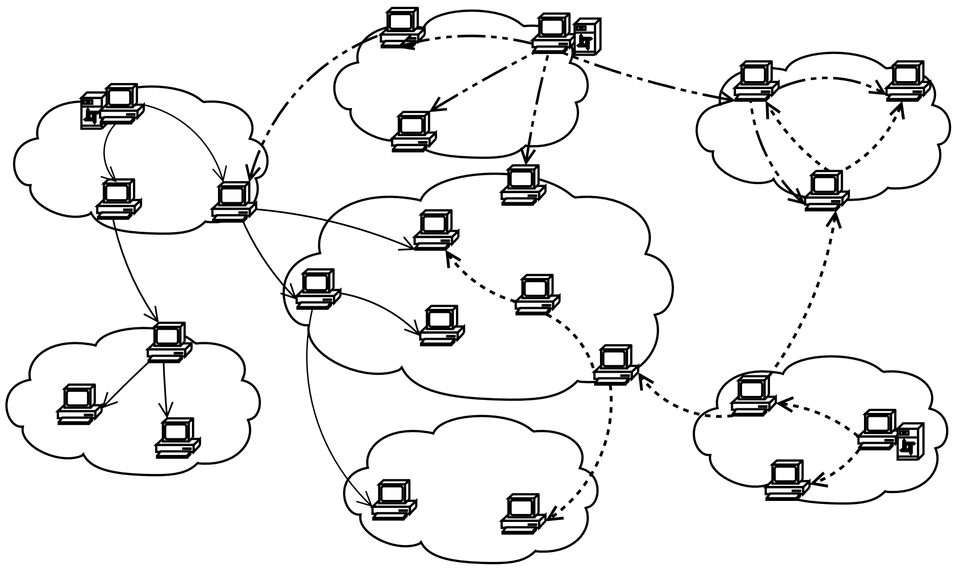


Figure 4.1: PeerCast Overview

4.2 PeerCast Framework Overview

We suppose that only *source peers* have the authority to update data items and initiate disseminating the freshest version of data to other peers. We also suppose that the source peers are long-running nodes. We address source peer failure and unsubscription problems in a later section. The peers caching the dynamic data items for querying are called *client peers*. Source peers provide the service just like the origin servers in Web applications. The difference is that peers in P2P systems can play source peer or client peer simultaneously. Figure 4.1 illustrates the overview of the PeerCast framework as an overlay formed with numerous dissemination trees. The overlay is maintained by the participating peers among the PeerCast automatically independent from the lower P2P infrastructures.

PeerCast provides the filtering and pushing service by organizing the peer members into a self-organized, source-specific, and logical spanning tree that is maintained as nodes join and leave. PeerCast framework uses the push approach to disseminate updates. Via the logical multicast trees, source peers push data up-

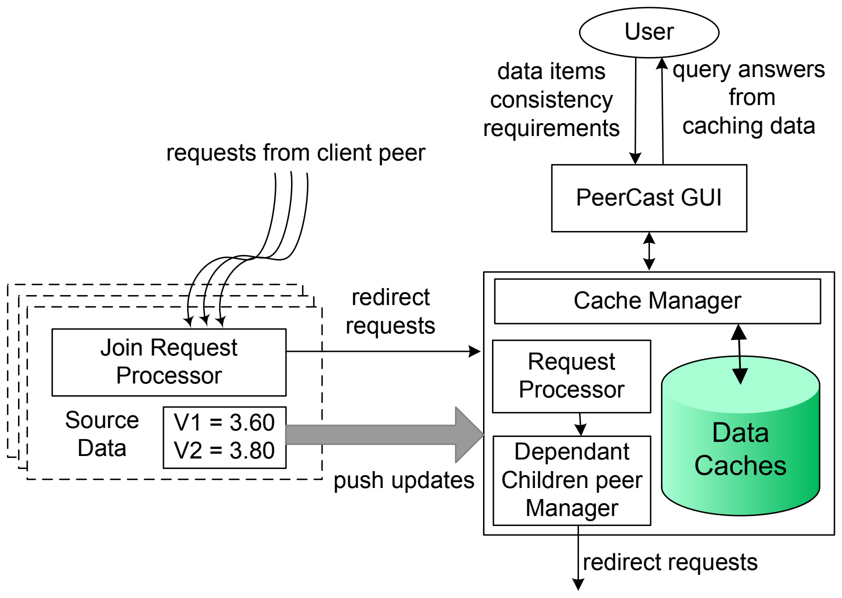


Figure 4.2: PeerCast System Architecture

dates to their dependent peers, which in turn push these changes to their dependent children peers. Each client peer participating in the dynamic data consistency overlay has a set of interested dynamic data items, say, their *ID*s (e_1, e_2, e_3, ...), with the corresponding consistency requirements (cr_1, cr_2, cr_3, ...). Client peers maintain metadata about the dynamic data, which include enough related source peer information. Not every update needs to be pushed to a dependent - only those updates necessary to maintain the consistency requirements at a dependent peer need to be pushed.

Figure 4.2 illustrates the internal structure of a peer node in PeerCast. There are essentially four components that are loosely integrated. The first component is a *dependant children peer manager* which facilitates immediate dependant peers management, manipulates their associated consistency requirements, maintains the data values pushed to the client peers last time, and checks the push condition satisfaction upon receiving data updates from upper level peers. For each child peer which is taken in, the associated metadata (BPID, transfer statistics, etc) are stored in a connection manager. The connection manager also monitors the statistics and manages the network reconfiguration policies with heuristic optimization mechanisms.

The second component is a *redirection process manager*. When a peer receives a join request from new coming client peer, it can take in the client peer as its child peer or redirect the join request to its existing immediate children peer to further process. The redirection mechanism influences the topology of dissemination tree greatly.

The third component is a *cache manager*. Cache manager takes charge of all the cached data. Peer users can query local cached data to achieve small response latency. Cached data can take any form, files or relational database tuples. Furthermore, potential parent peers are backed-up in cache manager when client peers join the dissemination tree.

The last component is a *graphic user interface*. It provides a user-friendly environment for users to specify their data items of interest and to set the associated consistency requirements. Upon receiving data updates, it presents them to the users. Peer users maintain their cached data items, and insert/delete data items with corresponding function models.

4.3 PeerCast Maintenance Policies

In this section, we discuss the design issues of PeerCast framework and provide detailed algorithms for dissemination tree construction and maintenance. Before presenting the maintenance policies, we define some metrics which are used in procedures. In addition, we describe the overhead on each participating peer to

Table 4.1: Metadata Structure

Metadate Attributes	Function
Master owner	data object owner
Fresh status	whether object is fresh
Membership	whether peer is one of multicast consistency tree active node
Level in tree	level no. of multicast consistency tree

maintain the functions in PeerCast.

In order to manage dynamic data efficiently in P2P systems, we provide more semantic metadata to describe the dynamic data items. In addition to the metadata to describe conventional static data used in static file object sharing P2P systems, we provide more metadata. Please see Table 4.1 for more details.

Peer nodes capacity are heterogeneous in real world P2P systems. There are high capacity peers in a typical P2P overlay network. They may have probably high rate CPU, large disk, broad bandwidth, and high quality of network connection. They may provide longer access availability than usual transient peers do. Peer capacity factor should also be taken into consideration of parent peer selection.

We define following metrics which will be used in building PeerCast:

1. In a heterogeneous operating environment, peers may be devices of different computing capacities. It is necessary to tell the computation capacity differences among the peers. The higher capacity of a peer, the more benefits it can bring to the P2P systems. *Advertisements* are used to represent peers' resources [6]. Advertisements are typically represented as a text document (e.g., XML file). Resources such as CPU speed, space, and upstream bandwidth are advertised.

2. *consistency requirements* : *cr* is staleness degree that users can tolerate. It is specified by peer users to each data item. Consistency requirement must

be set to a valid value when a peer joins the consistency overlay.

3. $preference\,factor$: $preference\,factor = delay(P,Q) \times numDependents(P)/ numDataItemsPsQ$, in which P stands for a peer in the consistency overlay; Q stands for a new coming client peer; $delay(P,Q)$ is measured using *hops* or round-trip time. $numDependents(P)$ is the current number of P's child peers; and $numDataItemsPsQ$ is the number of data items that P can serve Q. $preference\ factor$ is for client peer Q to choose parent peer in the potential parent peers set. The smaller this factor is, the more preferred a client peer is to be a parent of Q.

4. Each participating peer node has a *maximum_connection_number* parameter for serving children peers, and a *maximum_physical_connection_number* for immediate neighbor peers, which are set in BestPeer and decided by peer node's capacity and bandwidth.

Different from the conventional application-level multicast, PeerCast constructs dissemination tree by an incremental way, i.e., nodes arrive and depart one by one. Each internal node in the tree not only keeps the status information of its children nodes, but also keeps the status information about parent peer and several backup parent peers.

Obviously, the dissemination of data updates with peer cooperation take certain computational overheads and space overheads demand in each participating peer:

Computational Overheads: When the source peer or parent peer has to push data updates to its immediate child peers, for each change that occurs, the peer has to check if the *cr* of any of its immediate child peers has been violated. This computation is directly proportional to the rate of arrival of new data values, the number of child peers registering temporal consistency requirements associated with certain data value, and the total number of the cached data items. It is a time-varying quantity in the sense that the rate of arrival of data values as well as number of connections change with time. Parent peer responds to individual children peers one by one, which may incur queueing related overheads.

Space Overheads: Parent peers must maintain the cr value for each child peer, the latest pushed value, and the identifier of each child peer ($BPID$) along with the state associated with an open/logical connection. Since these states are maintained throughout the duration of children peers connection, the number of children peers which certain parent peer can handle is limited by the capacity (measured in *Advertisement* metric). As we will show that there also is an optimal number of dependant children peers cooperation for one data item. When the state space overhead becomes large, it will result in scalability problems. Therefore, we provide not only the pre-computational cooperation degree but also the self-adaptive procedure to adjust the workload of each participating peer. On the other hand, each client peer also maintains the status of its direct parent peer and potential parent peer status. Without maintaining the data and cr needs for individual children peers separately, a simple way to reduce the space needed is that the parent peer combines all the registers for a particular data item e and needs a particular cr_e (choose a minimum value from different crs). As soon as the change to e is greater than or equal to cr_e, all the children peers associated with e are notified. It reduces the space overhead. However, it may increase the network traffic, and decrease the benefit of cached data in children peers. We estimate the space overhead as follows. Suppose a client peer maintains n PeerCast connections. Each connection is specified by a (e, cr, $identifier$, $value$) tuple, and k backup parent peers. The state space needed is:

$n \times$ (bytes needed for a (e, cr, $identifier$, $value$) tuple) + $n \times$ (bytes needed for a connection state) + (k + 1) × (bytes needed for parent peer state)

Since peer node capacity has been raised exponentially recently and users also can adjust the cooperating capacity, the cost of the space is less than the cost of the heavy network traffic without using cooperation.

4.3.1 Dissemination Tree Construction Policies

Any peer n interested in maintaining consistency of dynamic data item e_i can submit a join request to the source peer of e_i since the identifier of source peer is always available (a unique URL has the information). Peer n receives the data item updates via the dissemination tree after participation. The source peer serves the coming client peers by registering their entries and establishing a logical connection if there is space in the capacity, or redirect the request to one or more suitable peers among its direct children peers if it has suffered a heavy dissemination burden. The procedure is repeated until a potential parent peer is discovered. Any potential parent peer should meet the consistency requirements of the new coming client peers at least. If coming client peer's requirements are more stringent than all the existing client peers, it will replace one of the existing children peer and let that peer become its dependant child. In this way, the client peers with stringent consistency requirement to certain data item are guaranteed to be placed much closer to the source peer of that data item than other peers with lower requirements.

We provide three different dissemination tree construction policies as follows and present the algorithms using pseudo-code.

Since a peer only knows its local topology, peer n can only forward the join request to one of n's immediate children, or its parent. Some of the options in choosing such a target are the following:

1. **Randomized Construction**

Upon having no available capacity to serve new client peers, the node n chooses one of its immediate child peers which can serve the new coming client peer at random as the target t, and redirects the request to t. Such a policy requires minimal state and computation cost at n. On an average, the form of tree is expected to be balanced. The submission entry of the client peer includes the dynamic data item identity e_i and the corresponding client peer consistency requirement cr_i. Since the computation cost is minimized, the first response delay to a new coming client peer is also expected to be small. It can return a small set of k potential parent peers to the new coming client peer. These k potential parent peers are considered to be the systematic parameters.

2. **Round-Robin Construction**

The node n maintains a list of its immediate children, and forwards the join request of new client peer to the child t at the head of the list. The child t is then moved to the end of the list. Such a policy requires some state maintenance, but is expected to keep the tree well-balanced. Since round-robin and random policies do the redirection computation without any global topology information, they are by no means optimal.

3. **Locality-Biased Construction**

Randomized and round-robin construction policies do not consider the network locality property. However, locality-biased construction policy helps in constructing dissemination tree by taking the network proximity into account. The node n redirects the request based on the peer locality in such a policy. To make use of locality property, one trivial way is that any client peer which wants to join the consistency overlay not only should submit the join request to source peer, but also should ping the potential parent peers to achieve the optimal choice. In locality-biased construction policy, node n chooses the immediate child peer with the least access latency to the coming client peer. This simple way is unpractical because it will cost huge round-trip messages. In order to save ping messages, PeerCast uses the Group-based Distance Measurement Service (*GDMS*) [55] to improve the performance. The inter-group and intra-group estimation can be figured out by the *GDMS* service. Node n chooses the redirect targets by the information of distance estimation. The locality property of tree construction naturally leads to the locality of PeerCast, i.e., parent peer and his immediate children peer tend to be close to each other. This provides PeerCast near-optimal data updates delivery delay and saves the bandwidth consumption.

When submitting the join request, a new coming client peer waits for the position response from the overlay. The client peer sets the parameter *max_waiting_time* after receiving the first answer. In the *max_waiting_time* period, client peer computes an optimal parent peer from the collected answers using the metric $preference$ $factor$. Meanwhile, client peer backups some peer nodes once they satisfy the consistency requirement and have not overloaded.

The difference between P2P scenarios and web proxy scenarios is that dissemination overlay should be established in an incremental way. The previous policies deployed in content distribution networks may not be practical in P2P environment.

We illustrate the dissemination tree construction procedure with our pseudocode using the following notations: c refers to the client peer, and s refers to the source peer, which is a tree root. o is the dynamic data item for consistency maintenance.

ls is capacity load of peer. lc_s is current load of peer. rc_s = *ls* - lc_s is the remaining capacity of peer.

We present detailed algorithms of randomized construction and locality-biased construction. We also present consistency overlay side procedure and new coming client peer side procedure.

Algorithm 1 *Randomized Construction Procedure*

```
Require: data item identifier, consistency requirement $\geq 0$, current value

switch message type
case "ACK":
if $c$ is not $s$'s direct neighbor then
    add $c$ into $s$'s virtual neighbor list;
end if
add $c$ into $s$'s children list;
break;
case "JOIN":
if consistency requirement < local consistency requirement then
    create "ROTATE" message response to $c$;
else
    if $rc_s > 0$ then
        create "ACK" message response to $c$;
    else
        choose a set of children peers $Q$ at random;
        redirect $c$'s request to $Q$;
    end if
end if
break;
```

Algorithm 2 *Locality-Biased Construction*

```
Require: data item identifier, consistency requirement $\geq 0$, current value

  switch message type
  case "ACK":
  if $c$ is not $s$'s direct neighbor then
    add $c$ into $s$'s virtual neighbor list;
  end if
  add $c$ into $s$'s children list;
  break;
  case "JOIN":
  if consistency requirement < local consistency requirement then
    create "ROTATE" message response to $c$;
  else
    if $rc_s > 0$ then
      create "ACK" message response to $c$;
    else
      estimate distance measurement between each children peer and $c$ using
      GDMS;
      choose a set of children peers $Q$ based on distance estimation;
      redirect $c$'s request to $Q$;
    end if
  end if
  break;
```

Algorithm 3 *Choosing Optimal Parent Peer*

```
  input data item identifier, consistency requirement, current value;
  create Message with those variants;
  /* since c has the metadata about s from o; */
  $c$ sends a "join" request to $s$ with $o$ through overlay network;
  receive first response message;
  set max wait time parameter;
  repeat
    if receive response message then
      add into potential parent peer list;
    end if
  until passed time > max wait time
  calculate all the potential parent peers, choose the optimal one measure in
  preference factor;
  create "ACK" message response to the parent peer;
  backup potential parent peer list;
```

4.3.2 Peer Leave/Recover Policies

Peers may come and go unexpectedly and behave autonomously. In an ideal situation, peers leave the systems gracefully and rotate all the intermediate dissemination responsibility to other cooperative peers before their departure. However, there is a problem of ungraceful leaves where a node departs because of a network disconnection, host crash, or another reason that gives it no opportunity to notify its children peers. Backup mechanism is expensive and it cannot guarantee robust recovery, e.g., backup peers go off. To accommodate such ungraceful leaves and repair the disconnection among the tree nodes immediately, PeerCast even provides soft maintenance messages to detect ungraceful leave and recovery from failure in time.

When peer n wants to unsubscribe any dynamic data item or unsubscribe from the overlay, it needs to forward a valid target t to its descendant peers. Node n is definitely aware of two nodes in PeerCast overlay: its parent peer, and the source peer. There are at least two candidate values for t. Therefore, we have the following four alternative policies:

- **All-via-Source (AVS)**: The node n chooses the source peer as the target. Peer n sends a redirect notification message to its immediate children peers, and it is recursively forwarded to all the descendants of n specifying source peer as a target. All the descendants of peer n submit requests to a source peer just like the join procedure. The advantage of this policy is that the dissemination tree is expected to remain balanced because of a redistribution of the affected nodes. However, it will cost more time to recover and reconstruct the system.

- **All-via-Grandfather (AVG)**: The node n chooses its parent peer p as the target. Peer p is the grandfather of n's immediate children peers. All the descendants of n are recursively redirected to p. Peer p takes in these children peers or redirects the request to its descendant peers. The advantage of such a policy is that the effect of the unsubscription is limited to the subtree rooted at p. Moreover, the source peer is protected from such requests in the event of multiple simultaneous failures. The dissemination tree is expected to remain balanced as the subtree is reconstructed from the same nodes as before.

- **Partial-via-Source (PVS)**: The node n still chooses the source peer as target t. However, only the immediate children peers of node n attempt to recover by contacting t. The rest of the descendants still retain their setting without any change. They rely on those nodes to recover their connections with the dissemination tree. The advantage of such a policy is that only the peers of n's immediate children peers could be accommodated near the source and others need no change. An explosion of requests to source peer is avoided. The shape of the dissemination of the original topology is not kept as the previous one.

- **Partial-via-Grandfather (PVG)**: The node n chooses its parent p as target t. Only the immediate children peers of n attempt to recover by contacting t. The advantage of such a policy is that the effects of failures are localized. However, the level of dissemination tree will be enlarged because of the failures, and it will increase the hops to deliver data updates to client peers in low level.

In the case of client peer's graceful leave, four alternative recovery methods provided by PeerCast can be used. They trade off the locality against the request explosion or tree balanced-topology. Despite of their drawbacks, they can be deployed adaptively to achieve a better performance according to the system situation.

However, in the case of an ungraceful leave, the departing node is unable to notify its children. The children peers of the departing node send heartbeat message periodically to detect the parent peer's failure. Once the time interval during which they do not get response messages from parent node goes beyond a time threshold, it is confirmed that parent peer has ungracefully left or failed. Children peers start up local backup parent peers. Recall that every new coming peer may get k, ($k \leqslant 3$ usually) backup parent peers when client peer joins the dissemination tree. Therefore, there are at least (k+1) target peers to recover. Thus, children peers choose an optimal online peer as the primary parent from k peers according to the metric *preference factor*. If all the k potential peers reject the requests because of overload or departure, a child peer re-joins the overlay by sending requests to source peer via AVS or PVS policies.

4.3.3 Self-Adaptive Policies

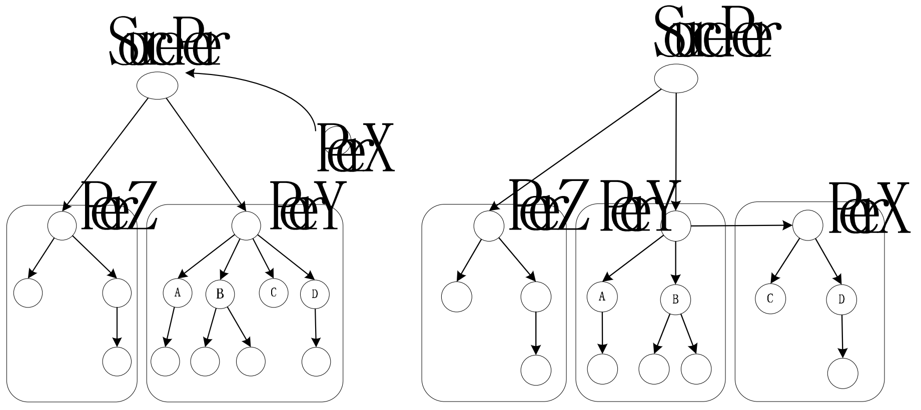
The previous construction policies of the dissemination trees do not consider the dynamic network attributes and users' changing of data accessing. It is important that the topology is rearranged in keeping with dynamic measurements of those factors. Self-adaptive policies are proposed to improve the overlay efficiency, reduce the workload of intermediate peer's heavy burden, and adapt to network dynamics.

It is possible that peer users switch to other new interesting data items such as the new stock price monitoring and adjust consistency requirements at will. When a client peer's dynamic data or its data coherency requirement needs change, the self-adaptive policies re-organize the node position to satisfy the peer's new demands.

When a client peer caches any new dynamic data item, it re-applies the join procedure to maintain the new data items consistency. When client peer n removes the dynamic data item from his monitoring set, which will affect its dependant peers, n searches a suitable target peer t to replace its role and the immediate children peers choose a primary parent peer to serve that dynamic data updates among t and the backup parent peers. When client peers change data item consistency requirements, it will impact the descendant peers or the immediate parent peer. In the case of n increasing the consistency requirements, the parent peer of n cannot serve it any longer. In the case of client peer n relaxing the consistency requirements, it may not be able to serve its descendant peers. n can submit "update" request to its parent peer to claim for switching to a more stringent consistency requirement peer to serve. n searches for target peer to serve its immediate children peer when relaxing the consistency requirements.

Some of the peer nodes can turn out to be unstable, because they join and leave frequently. This is called churning. The churning problem results in an overlay suffering from data delivery inefficiency. It is necessary for the need of adaptation. Even if we include redundancy and failure discovery mechanisms, instability must be taken into account when creating the topology. A useful optimization is to have long time available peers play as intermediate nodes while unstable ones are moved to the leaves of the topology. Of course the stability of a node is unknown when it first joins the overlay. A default value is first assigned to the node stability variable and the latter is regularly updated as time goes by. We postpone the leaf-sinking design [64] in Chapter 5.

The basic tree construction algorithm is greedy in nature. The order of peer nodes joining the dissemination tree can affect the topology of tree and its quality. Self-adaptive policies evolve the multicast tree by load balancing. As shown in Figure 4.3, when new coming client peer x enters the overlay, peer y can share its heavy workload with x only if x has available resource and enough stringent consistency to serve peer c and d.



(a) Before Peer X Participation (b) After Peer X Participation

Figure 4.3: Self-Adaptive Policies Load Balancing

Algorithm 4 *Load Balancing Procedure*

```
Require: Peer p1, p2

  if p1 receives p2's join request and p1.isOverloaded() then
    for all peer child r in p1 do
      if $r.cr > p2.cr$ then
        insert into list;
      end if
    end for
    divide list into $set_1$, $set_2$ at random;
    for all peer c in subset do
      add c into p2's children list;
    end for
  end if
```

4.3.4 Source Peer Recovery

In Web proxy caching, content distribution networks, or mobile computing environment research areas, it is taken for granted that servers are stable and seldom failed. The server failure or departure is hardly considered in these situations. However, in a P2P environment, source peers are not guaranteed to be long-running.

Hardware maintenance, software update, or just reboot can take place anytime. To recover from source peers' unexpected failure, we regulate source peers by publishing dynamic data item to register the pair (*PeerID*, *ObjectID*) and source peer's immediate children peers, which are located in top level of the dissemination tree, to *LIGLO* server in BestPeer. Because the pair and related status information take just a few bytes and cost a little overhead, it does not burden the *LIGLO* server. When the source peers are down, the corresponding consistency multicast trees lose roots. If the logical connections are not maintained, it will cost much traffic cost in re-building the overlay when the source peer resurrects. All the client peers in the original dissemination tree keep the status of the logical parent-child relations for a given period threshold. After the threshold is exceeded, the client peers thought the tree would no longer revive and throw away the storage if no information about source recovery is provided.

When source peer comes up next time and publishes the same dynamic data items, it can retrieve immediate dependant peers from BestPeer *LIGLO* servers, fix and reactivate the date dissemination tree.

4.4 Summary

In this chapter, we present the design issues of PeerCast framework. We define the metrics used in the system and discuss the essential policies for maintaining the PeerCast topology and cooperation relations. Efficiency and performance of PeerCast will be examined in Chapter 6. In Chapter 5, we supplement PeerCast design with system implementation enhancements.

CHAPTER 5

PeerCast Implementation Issues

The previous chapter presents the design of PeerCast. In this chapter, we discuss some techniques which can optimize PeerCast to enhance the performance. We provide improvements in two aspects. One is about resource usage optimization based on heuristic network re-organization, the other is to about the mechanism for pushing the unstable client peers to the edge of the topology in order to prevent the churning problem.

5.1 Heuristic Optimization for Resource Usages

In PeerCast, there are two kinds of connections, physical connection and logical connection. One is a long-running socket connection between two nodes, called a physical connection. The other is no long-running socket connection between two nodes, called a logical connection. In logical connection, the two nodes just maintains the corresponding IP address each other without real socket connection.

When data needs to be delivered, the two nodes establish the socket connection and inter-deliver. Otherwise, they release the connection to save the resources.

The initial neighbors to which peers connect are the starting points when they enter into the BestPeer infrastructure. Since PeerCast overlay construction procedure never considers the low level infrastructure physical topology, it is not optimal. Data delivery by logical connection needs to initialize the physical connection and release it after finishing disseminating updates by the links. When the frequent dissemination such as numerous data updates for inter-delivering happens during the connection between two nodes, it will cause huge overhead to the peers in the overlay because of the connection initialization and release. Consequently, the significant latency is incurred. Motivated by the above observation, PeerCast provides the heuristic policy to optimize the efficiency of the data delivery.

Each peer has a number of available network resources. It is supposed that a peer maintains a limited set of neighbors in BestPeer infrastructure and a number of logical connections in upper overlay PeerCast. The goal is to assign a set of neighbors to each peer n so that there is a high probability for n to obtain or deliver the updates from them in shortest latency. Since the number of the allowed network connections is expected to be small, each connection is assigned a benefit value dynamically and heuristically to manage the topology in our method. As expected, after periodically collecting the statistics, we can figure out the most beneficial connections and choose their ends to be the peers' immediate neighbors.

We formulate the problem as a case of finding the optimal combination in a greedy manner to achieve the maximum benefits. The procedure relies on the BestPeer infrastructure network reconfiguration primitives and connection management in PeerCast. Ideally, frequent data updates delivery is all disseminated via the physical connections and the infrequent delivery via logical connections. In

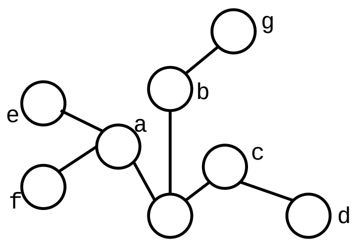


Figure 5.1: A Sample Network for Reconfiguration

this way, it not only saves the connection initialization and releases the consumption but also minimizes the updates transfer delay to improve the fidelity of cached data.

As illustrated in Figure 5.1, for instance, peer node n's maximum physical connections, $pcmax = 3$. Its direct neighbors are peer nodes a, b and c. Meanwhile, n maintains the logical connections with peer nodes d, e, f and g with maximum logical connections, $lcmax = 4$. However, in our extreme sample network overlay, n's direct neighbors provide no cooperation benefits to n. It is a waste of n's network resources. To avoid this situation, each peer manages the connections usage rates by a vector counter. Every element in the vector stores the updates delivery statistics of a corresponding connection. After the counter runs for a specified period (*system parameter*), the elements of frequently used connections in the vector counter should have a higher value. According to the *least recently used* (LRU) locality principle, it is reasonable to assume that these frequently used connections will be used more often than the others in the near future. Then, a different benefit value is assigned to each connection. The network reconfiguration procedure starts to reorganize the physical topology if some logical connections bring more benefit than the existing physical connections do.

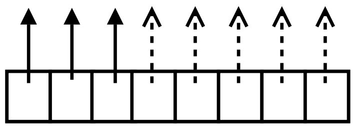


Figure 5.2: Heuristic Policy for Optimization

5.2 Preventing Churning Problem

Some of the nodes can turn out to be unstable (e.g., mobile computing devices or a bad wireless connection). Even if PeerCast provides the robust failure recovery mechanism, the instability still should be taken into account when creating the disseminating overlay. Otherwise, the whole system will suffer from their churning. Child peers connected to unstable nodes could not get the data updates timely. In this section, we provide the heuristic mechanism to prevent such churning problem.

The idea is to push the transient peers to the edge of the topology, and let them undertake less child nodes than stable nodes. Nevertheless, the stability of a node is unknown when it first joins the overlay. In order to make adaption possible, we assign a *node_stability* variable to each peer node, which is given a default value at first time and updates as time goes by. In addition, *node_stability* is encrypted to achieve the security. According to the system context, we set two corresponding threshold values: $leaf\_only$ value and *good_intermediate* value. The values between these two are considered as *intermediate_possible*. We assume that the number of unstable/churning nodes is still minor in comparison to the normal peer nodes.

Source peers take bloom filter[1] to record history of peer entrance. Each client peer has a default value for the node stability. When it joins the overlay, source peers can check how many times it has entered the overlay. For those peers joining and leaving repeatedly, *node_stability* of them is decreased. On the other hand,

[1]A bloom filter is a method for representing a set of n elements to support membership queries[61]. We use the similar bloom filter function as [46].

node_stability will be increased periodically if peer persists.

Based on the extra information about the node stability, new coming client peer could consider the stability of the internal nodes when choosing the parent peer. The peer whose *node_stability* is below the threshold *leaf_only* is never considered as a parent peer. It plays as the leaf node only. Thus, the dissemination tree construction can guarantee more robust topology and provide higher quality data update delivery service.

CHAPTER 6

Experimental Evaluation

In this chapter, we demonstrate the efficiency of PeerCast framework through an experimental evaluation. We present our experiments by two parts. In the first part, we mainly examine the impact factors to the performance of PeerCast framework; in the second part, we compare our approach with Gtk-Gnutella protocol. In each part, we describe our experimental methodology, procedure and metrics used for evaluating the performance of PeerCast. Then we report the results from a series of simulations and present detailed analysis.

6.1 Experiment Methodology

6.1.1 Environment Setup

We employed two implementations to evaluate our methods. The first one is a JAVA prototype built on BestPeer platform which runs on Pentium III PCs with 1GB RAM and Microsoft Windows XP. It was used to derive the basic parameters of the

Table 6.1: Parameters Derived from the Prototype

Parameter	Value	Comments
TR_R	3.6889 KB/sec	Average transfer rate between remote peers (WAN)
TR_L	594.935 KB/sec	Average transfer rate between local peers (LAN)
MSG_{join}	1.0996 KB	Join request message size
$MSG_{redirect}$	1.1777 KB	Redirect request message size
$MSG_{response}$	0.9766 KB	Response to join request message size
MSG_{insert}	1.1738 KB	Response to join request with insertion message size

system, see Table 6.1. The parameters were used in the second implementation, which was a simulator based on SIM: a C++ library for discrete event simulation [10]. We employed the simulator since it would be impractical to set up a large network. Furthermore, the benefits of our approach could become significant when there are many participating nodes. We also implemented Gtk-Gnutella cache consistency protocol [53] on the simulator for comparison.

6.1.2 Testing Data Setup

The performance characteristics of our solution are investigated using real world stock price streams as dynamic data. We used the same trace data as in [30, 73, 72]. The presented results are based on historical stock price traces obtained from the *http://finance.yahoo.com*. We collected 50 traces, which were the most active stocks in Nasdaq. The details of the traces are listed in Table 6.2 to suggest the characteristics of the traces used.

6.1.3 Network Setup

We simulated the typical P2P network topology. The nodes were connected either through a slow WAN or a fast LAN line to the network. We employed the power-law topology [33]. The physical network model was randomly generated. We set up M corresponding source peers. These source peers were assigned same number

Table 6.2: Characteristics of the Traces Used for the Experiments

Stock symbol	*Time Interval*	*Min*	*Max*
Microsoft	2-Jan-03:31-Dec-03	22.81	57.0
Intel	3-Jan-03:31-Dec-03	13.0	29.01
Oracle	2-Jan-03:31-Dec-03	10.65	13.92
IBM	2-Jan-03:31-Dec-03	75.25	93.9
Cisco	2-Jan-03:31-Dec-03	12.87	24.83
SINA	2-Jan-03:31-Dec-03	5.6	45.6
SUN	3-Jan-03:31-Dec-03	30.0	52.5
YAHOO	2-Jan-03:31-Dec-03	17.5	46.44
SAP	2-Jan-03:31-Dec-03	18.85	44.75
AMD	4-Jan-03:31-Dec-03	4.95	18.23

of specific stocks data. As we stated in Chapter 4, source peers are in charge of updating data and initiating data dissemination for numerous client peers. In our experiments, we varied the size of the network N from 100 nodes to 1000 nodes. Meanwhile, we set up $\sqrt{N}$ peer groups in the locality-biased tree construction policy. Each group has inter-group and intra-group distance estimations which would affect the average transfer rate between remote peers or local peers. We used the proportional rates to simulate the inter-group and intra-group network locality factors. The computational delay incurred at the peer to disseminate an update to a dependant child peer is totally taken to be 12.5 ms, which is estimated based on the [73]. It included the time to perform any checks to examine whether an update needs to be propagated to a dependent and the time to prepare an update for transmission to a dependent.

We simulated static and dynamic P2P networks. Static P2P network was used to compare the cost among dissemination tree construction policies. We evaluate PeerCast performance mainly under dynamic P2P network. We modeled node departure by assigning each node an up-time picked uniformly from [0, *max_life_time*] at random, where *max_life_time* was set to be the same value as the total simulation time. To maintain the size of network, departure nodes act as new coming peers to rejoin the overlay after they leave the overlay. The procedure can be regarded as (1) peers disconnect from their neighbors, (2) shutdown, and (3) peers immediately rejoin the system by connecting initially to a random number of neighbors. We assumed that rejoining nodes took the original data of interest, but with different consistency requirements.

6.1.4 Simulation Metrics

We considered the following metrics for performance analysis. The key metrics for our experiments were the *fidelity of the cached data* and *query false ratio*. Fidelity of the cached data is used to measure how a peer user's consistency requirements are met. It is time for which the difference between local cached data value and source peer data value are kept within user's consistency requirements. To illustrate the figures and explain clearly, our results were plotted using *loss of fidelity*. It is an alternative metric simply $100\% - fidelity$. Likewise, query false ratio is referred to the ratio of querying the stale cached data. To some extent, query false ratio is more important, since peer users care more about the correct query answers. The smaller query false ratio, the better performance of the consistency mechanism. We also measured the number of messages and bandwidth consumption. They are the major metrics to examine the network traffic overhead.

6.1.5 Simulation Procedure

The whole procedure has two parts:

In the first part, we constructed the dissemination overlay using three different policies in order to compare their network traffic consumption. We ran the trace procedure. Randomized, round-robin and locality-biased construction policies are compared taking centralized approach as the baseline. We changed the maximum degree of cooperation to examine its impact on the system performance. In this part, we also showed the performance of PeerCast framework using heterogenous peer capacity with real world distribution. We evaluated the PeerCast recovery policy performance with varying the number of backup peers.

In the second part, we compare PeerCast with Gtk-Gnutella protocol [53]. The reason we choose Gtk-Gnutella protocol to compare is that it is so far the cache consistency protocol designed for P2P caching systems. Moreover, it is built on Gnutella. Likewise, PeerCast is built over BestPeer, a Gnutella-like P2P infrastructure. Gtk-Gnutella provides three alternative approaches to adapt the dynamics of P2P network and update rate of the dynamic data. We compared the efficiency and overhead between them. Lastly, we evaluated the scalability of PeerCast. We also present results by our heuristic policies to show their enhancement to the performance of PeerCast.

6.2 Experimental Results and Analysis

Each client peer cached certain number of dynamic data items, between 10 and 50. These items are randomly picked from the stock list. The consistency requirements for each specific stock were different in different peers. We set the T% of the cached data in client peers with high stringent consistency requirements. High stringent consistency requirements were uniform randomly picked from [0.01, 0.099]. The rest 100 - T% data are less stringent requirement, picked randomly from [0.1, 0.99]. We set the T% value equals 50% initially, which means each peer user was interested in half of all the provided stock with high attention.

Time Cost to Redirect Request: This experiment is to test the delay to redirect a request. We set up six PCs placed in local area network (LAN), all

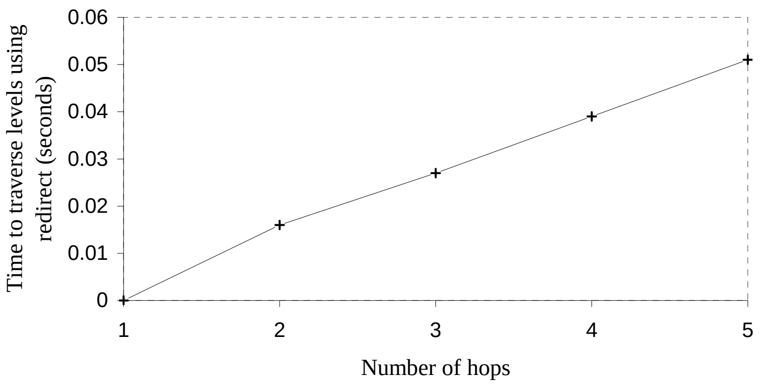
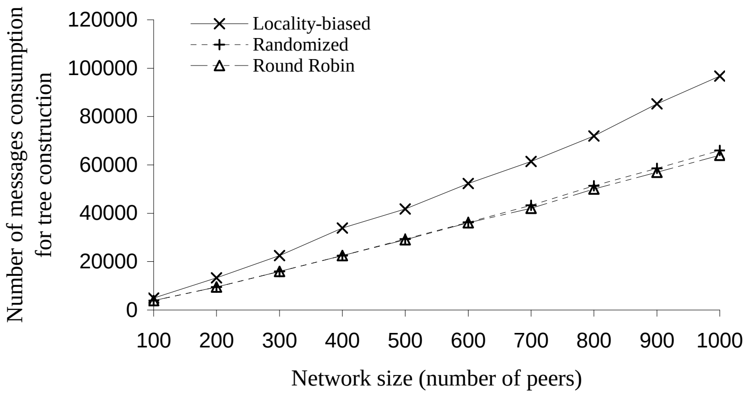


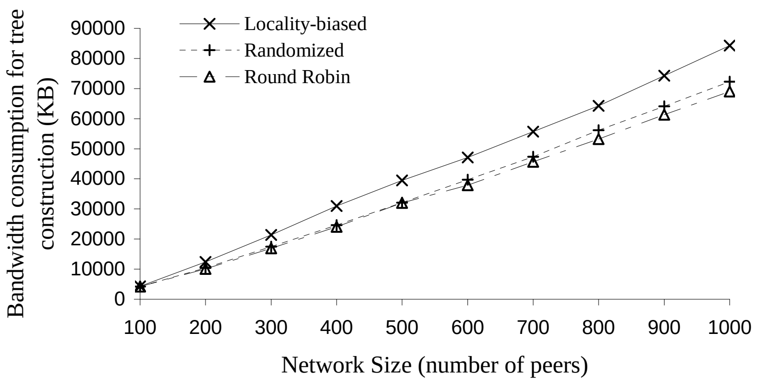
Figure 6.1: Redirect Message Latency

running PeerCast framework. Peer topology was formed as a chain rooted at one PC as the source peer. Measurements were then made to estimate the redirect join request cost time. The Chain topology showed us the effects of increasing number of levels of a peer from the source. Note that this experiment results will be biased toward peers having a high bandwidth capacity. Hence, the results of this is illustrated as the performance trends. As shown in Figure 6.1, the times to traverse levels in Chain were basically formed a linear in the number of levels. Note that the redirect cost time is far smaller than the update interval, so the redirect mechanism is efficient.

Dissemination Tree Construction Cost: We examined the cost comparison of the different dissemination tree construction policies in PeerCast. The topology of the dissemination tree has a significant impact on fidelity of data. The larger delay between node to node, the greater the loss in fidelity of cached data. As illustrated in Figure 6.2, the locality-biased construction takes more message consumption and bandwidth cost than the randomized and round-robin constructions. It is because that locality-biased construction policy should use numerous multicast ping messages to generate peer groups to estimate the distance between nodes [55]. Randomized and round-robin policies cost nearly the same bandwidth. Although locality-biased tree construction policy consumes more network resources, we see from the latter experiments, it brings more benefits than the other two policies.



(a) Number of Messages



(b) Bandwidth usage

Figure 6.2: Tree Construction Cost

Average Time to Join in the Overlay: *Time to join in the overlay* is an important metric as it records the response time of the PeerCast system. A new coming client peer submits the join request, and waits in the transient state until it receives the first response from the overlay. Figure 6.3 shows the average join time collected by every client peer subscribing to the dissemination overlay. The X-axis plots the number of participating client peers, while the Y-axis plots the waiting time interval of the new coming client peers joining in on average. We can observe that the response time of locality-biased constructed overlay is less than

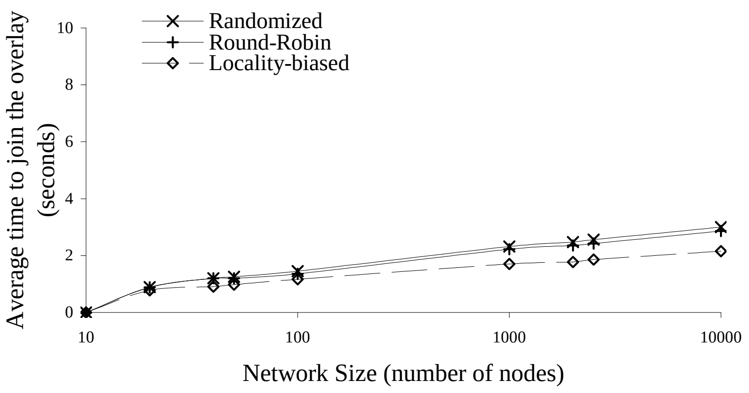


Figure 6.3: Average time to join in the overlay

the other two methods. Group-based Distance Measurement Service has collected the network proximity information so that it can reduce the message delivery time cost.

Performance Comparison with Centralized Approach: This experiment was to compare the performance of overlay constructed by randomized, round-robin and locality-biased policies. We took the centralized approach as the baseline. The results are illustrated in Figure 6.4. Locality-biased construction performs best as we had expected. Due to taking network proximity into account, it takes less delay to disseminate the data updates. Randomized and Round-Robin have the similar performance. Centralized approach performs worst. We can see that the centralized curve takes a sudden jump at the point of 300 peers. The reason is possibly that updates are queued in prior to consistency requirement. So, peers with stringent consistency requirement are placed at the front of the queue. The low ones are placed at the end of queue. With the same queue delay, high stringent peers suffered more than lower ones. Therefore, after the jump, the centralized curve increases slowly. Although centralized approach could take minimum communication latency, it suffers from large computational delay. Centralized approach fails to scale with client peer nodes.

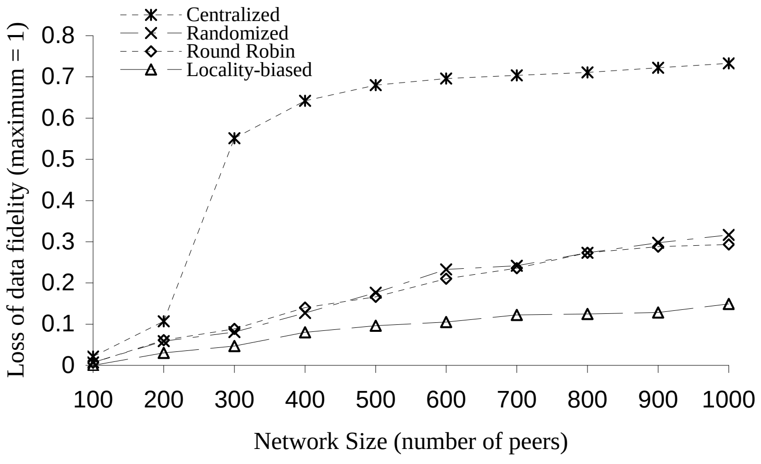


Figure 6.4: Performance comparison

Impact of Client Peer Bandwidth: We did this experiment to show the impact of bandwidth of client peer capacity, i.e., the effects of cooperative degree of client peers on the performance of the PeerCast. Since each peer filters and forwards the data updates to its child peers, the performance of the PeerCast framework is sensitive to the available bandwidth at the nodes participating in the system. We characterize the bandwidth which nodes can contribute to PeerCast by the number of maximum children, $number_{max}$. We can see from Figure 6.5 that centralized approach presents as a horizontal line. Because centralized approach does not take the advantage of client peer cooperation, it is immutable to the variety of the client peer capacity. PeerCast has great performance fluctuation with the variety of client peer capacity. We can see that the performance fluctuation of them is both like a V-curve with the increment of the max allowed children.

Due to the computation delay and network dissemination delay, when the allowed children is increased, the computation delay will enlarge. It will take more time to finish assembling the updates to deliver in the waiting queue. When $number_{max}$ equals 1, tree is formed as a chain. The computation delay is small. However, the network delivery delay is increased so that the performance is even

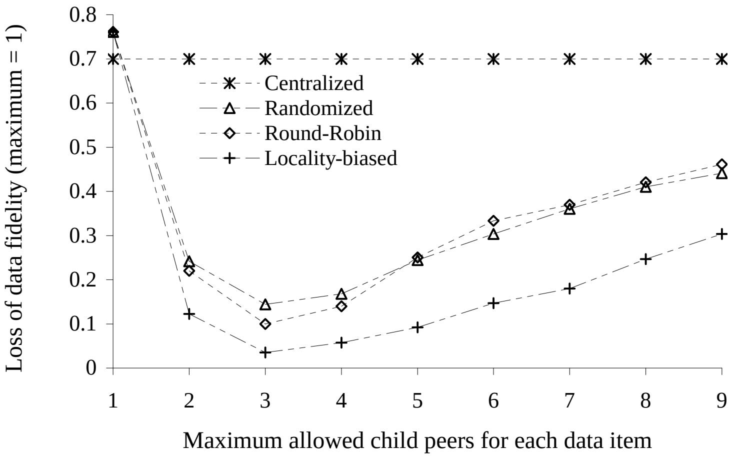


Figure 6.5: Impact of client peer bandwidth capacity

worse than centralized approach. Figure 6.5 shows that it brings no more benefits after increasing the cooperative degree beyond a threshold. The threshold was 3 in our experiment.

When $number_{max}$ increases, the depth of the corresponding dissemination tree decreases. Although the data updates delivery delay is reduced, due to each parent peer takes more child peer to serve, the overlay suffers from the large computation delay. So, in PeerCast deployment, it should use the optimal cooperative degree to achieve better performance.

Impact of the Peer Departure (A): PeerCast provides four alternative peer departure/failure recovery methods, we measured the four recovery methods in this experiment. The cost of different policies depends on the shapes of dissemination trees that result from changes in the overlay. In the absence of peers departure or failures, the dissemination tree would be almost-complete as it is initially established. Figure 6.6 shows the distribution of the depth of nodes in the dissemination tree for *AVS*, *AVG*, *PVS*, and *PVG* leave policies with the join policy set to round-robin policy, and $number_{max}$ set to 3. The X-axis plots the level number in the tree. The Y-axis plots the average percentage of nodes subscribed to the tree at a

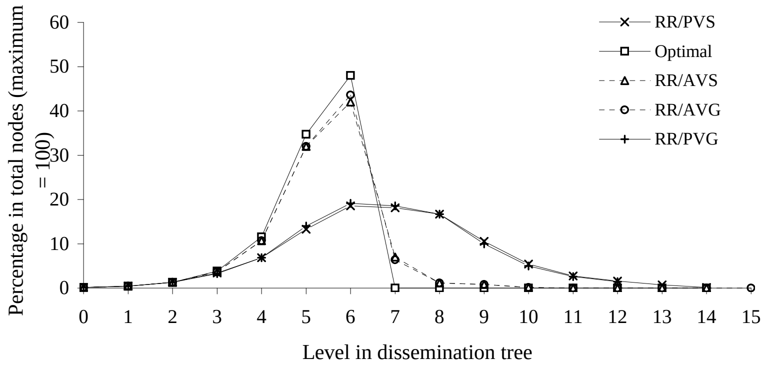


Figure 6.6: Impact of Peer Departure to Topology of PeerCast

certain level. We can see that *AVS* results in a smaller mean depth tree than *PVS*. The *AVG* and *AVS* curves peak and fall to 0 in a small number of levels, indicating a desirable compact tree. The *PVG* and *PVS* curves rise with *AVG* and *AVS*, but have a smaller percent of nodes at their peak in the middle levels. Instead, the remaining percentage of nodes fall off gradually along higher levels. The reason is that in *PVG* and *PVS* policy, the failure of a peer node n results in each of its child peer rooted sub-tree moving together, causing an increase in height. In *AVS*, all the descendants of n, independently contacts source peer and get distributed across the tree, causing the height to remain balanced. A high depth dissemination tree is undesirable because end-system delays increase linearly with levels. However, as we experimentally observed in redirect request time consumption, the increment of tree level will still let *PVS* and *PVG* acceptable. Furthermore, *PVS* and *PVG* will reduce the crash re-joins to the source peer.

Impact of the Peer Departure (B): Peer's frequent leaving does negative effect to the performance of PeerCast because of the disconnection of the dissemination tree. Disconnected child peers cannot get the data updates pushed from his parent peers. If peers are just logically connected, the failure can only be detected using soft heartbeat message to check the status of the parent peer. So, if the

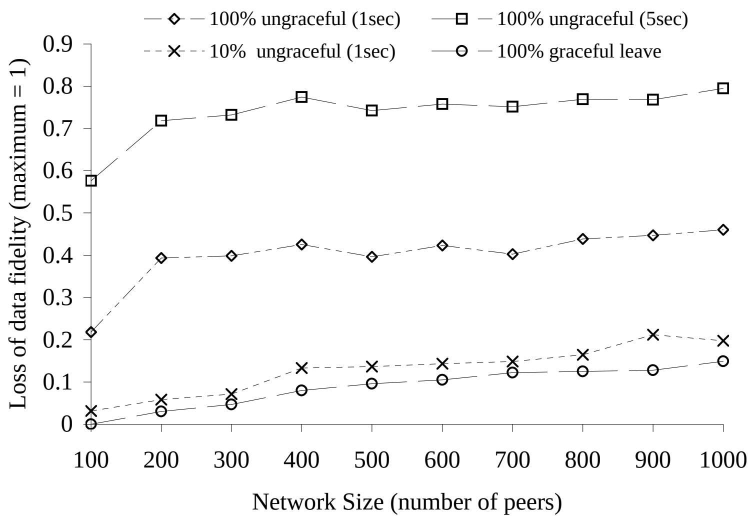


Figure 6.7: Impact of the peer departure

children peer can not get the update in a systematic parameter T, it would submit a heartbeat message to get the status of the parent, and adopt recovery method if it detects the failure of the parent. Those total time affects the performance of PeerCast. Moreover, the proportional of the client peer's ungraceful quitting from overlay is also a factor of the efficiency of the PeerCast. We got the results from Figure 6.7. If the 10% leave are graceful leave, the performance is still acceptable. The degree of ungraceful leave and repair interval can have a significant impact on the PeerCast performance. As illustrated in Figure 6.7, we set repair time 5 seconds instead of 1 second, the fidelity of cached data declines greatly.

Impact of Different Consistency Requirements: Initially, we set all the client peers take 50% dynamic data items with stringent consistency requirements. In this experiment, we increased the proportion of high consistency requirements, as illustrated in Figure 6.8, we can find that with more stringent consistency requirements to dynamic data, loss of data fidelity increases. Augmenting the number of stringent consistency items increase the fidelity of data, meanwhile, reduce the benefit of cached data in peers.

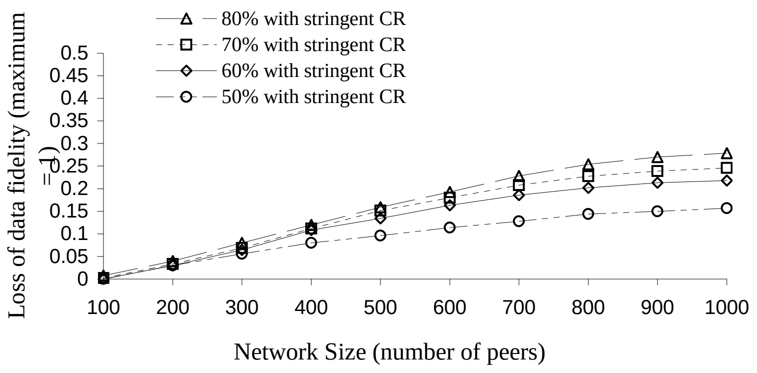


Figure 6.8: Impact of different consistency requirements

Real World Capacity Distribution: Peer capacity is heterogeneous in P2P environment. We simplified the simulation model in previous experiments. This experiment, we took this factor into consideration. We used the peer capacity distribution in [21]. 20% peers of the whole system can be regarded as connected in with cable modem so that can only serve one children peer at most. 45% peers of the whole system could serve five dependant peers. The rest 35% peers were strong peers, their capacity allowed them to serve ten children peer at most. We got the performance results from Figure 6.9, The combination of locality-biased construction and AVG recovery policy performs better than others.

Impact of the Number of Backups: Lastly, we evaluated the impact of the number of backups in PeerCast. The number of backups is decided in the procedure of the overlay construction. In ideal case, all the peers leave gracefully. The increment of the number of backups can not bring any benefit. However, in worse case, peers churn in the overlay or ungracefully leave without notifying children peer. Children peers have to re-join the overlay if having no backup parent peers. It will take more time to recover from disconnection. Since backup peers may have departed the system, the increment of backups can raise the system performance and minimize the recovery latency. As illustrated in Figure 6.10, we

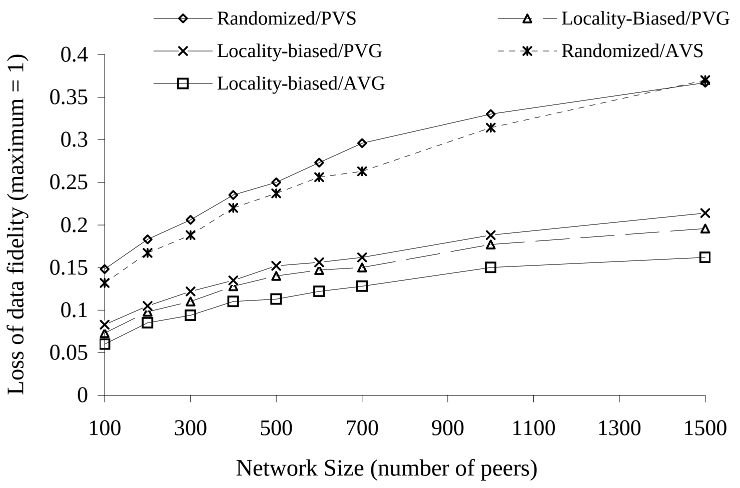


Figure 6.9: Impact of heterogenous peer capacity

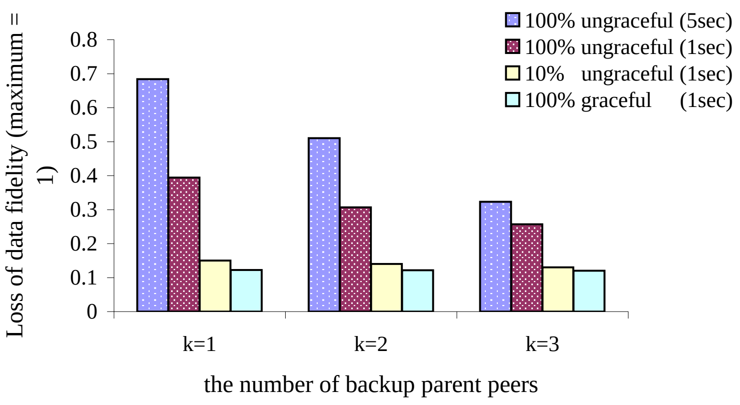


Figure 6.10: Impact of number of backups

set 100% peer ungraceful leave and repair time interval 5 seconds, more backups can reduce the loss of fidelity in evidence.

6.3 PeerCast VS. Gtk-Gnutella Protocol

Gtk-Gnutella is an enhancement protocol designed for Gnutella-like P2P caching systems [53]. It is incorporated with three cache consistency techniques: *push*, *adaptive pull* and hybrid approach called *push with adaptive pull*. To our best knowledge, this protocol is the only existing work for designing maintaining cached data consistency in unstructured P2P systems so far. Therefore, in order to prove that PeerCast framework can outperform the previous work, we also did a set of experiments to show that PeerCast can achieve more efficiency and performance benefits than Gtk-Gnutella protocol.

Metrics for Performance Analysis: In this part, we mainly use the metric: *query false ratio* (FVR) instead of the loss of fidelity in data which was used in our previous experiments. The query false ratio is the fraction of query responses that deploy the cached data which is out of the bound of consistency requirement. Peer users care the correctness of query answers more than the actual cached data staleness. We think it is the major factor about the cached data function.

Simulation Environment: We implemented Gtk-Gnutella protocol on our event-based simulator. A randomized uniform overlay network is setup as the topology of the Gnutella scenario. Each peer has 3 or 4 direct neighbors. As illustrated in Table 6.3, the major parameters are set before we run the experiments. Each peer can not only initiate query to the cached data but also be responsible for propagating invalidation messages to neighbors. Inter-arrival times of queries, I_{query}, are exponentially distributed. We still use the same dataset as in the previous experiments. There are 10 source peers taking charge of 50 fluctuating stock data updates. Each source peer is assigned an update rate I_{update}. We varied the network size, i.e., total participating peers. PeerCast and Gtk-Gnutella protocol were compared.

Performance Comparison: We ran the three strategies of Gtk-Gnutella and PeerCast framework, then collected the results. Here, for PeerCast, we used locality-biased construction and *all-via-grandfather* (AVG) for peer departure recovery policy. We firstly fixed the average I_{update} to 1 second, and the average I_{query}

Table 6.3: Parameters for Experiments

Parameter	Description	Default Value
L_{sim}	Length of simulation	10 hours
I_{update}	whether object is fresh	1 second
I_{query}	whether peer is one	5 seconds
Poll rate	frequency of checking consistency	adjust with TTR
TTR	time to refresh	associated by data item
TTL	total traverse hops of message	7

to 1 second. For adaptive pull policy, polling frequency is total decided by a time-to-refresh (TTR) value. TTR value is initially set by the value associated with each data item and adaptively adjust from the history records statistics. The TTL value for invalidation message was set to 10 hops. We vary the network size from 100 nodes upto 1500 nodes. Figure 6.11 ploted the performance comparison between PeerCast and Gtk-Gnutella protocol. The performance of push-based invalidation approach is poor when the network size is over 700 nodes. It is because the scope of the invalidation message is limited by the TTL value. For large-scale network, client peers out of the reach scope of the message will not be notified the updates. Likewise, adaptive pull policy is suffered from the scalability. Since data updates happen unpredictably, it is hard to determine when and how frequently to poll the source peer to check for the consistency. Adaptive pull policy can only provide weak consistency. On the other hand, push with adaptive pull and PeerCast can provide satisfactory query fidelity, very close to PeerCast. Note that we have not counted the overload problem in Gtk-Gnutella, so, in fact, Gtk-Gnutella protocol is overrated. Push with adaptive pull approach combines the advantages of the push and adaptive pull techniques. The client peers out of message reachable scope also can poll the consistency adaptively.

Network Traffic Consumption: We collected the bandwidth consumption from the above experiment. We can see from the Figure 6.12, PeerCast saves

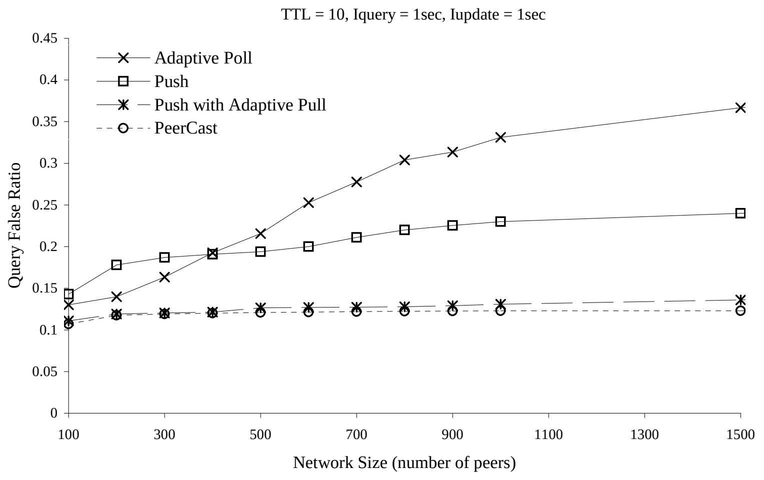
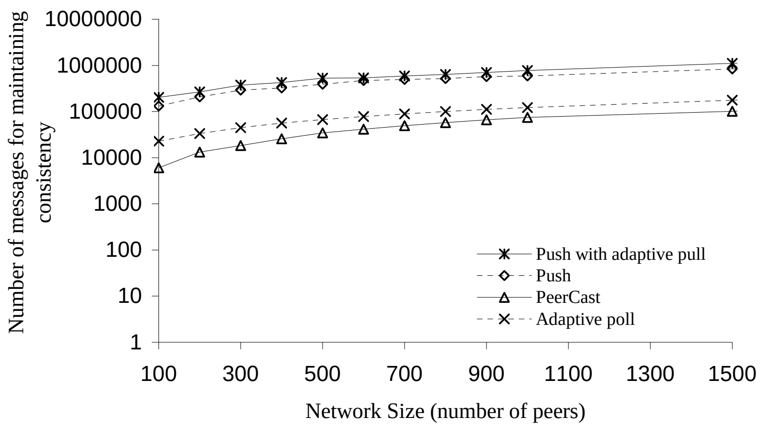


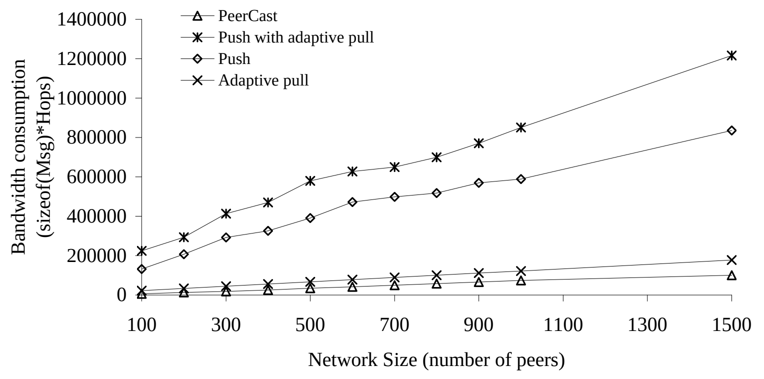
Figure 6.11: PeerCast Vs. GtK-Gnutella

huge bandwidth cost than the three strategies of Gtk-Gnutella. Push-based and hybrid approaches impose far more network overhead when compared to adaptive pull and PeerCast approaches. The reason is that push-based policy uses blind broadcasting invalidation message which leads to huge traffic waste. In order to prevent transient peers missing updates, hybrid policy costs additional periodically polling messages to push-only policy, taking more network traffic consumption. The cost of adaptive pull policy is correspondingly small, however, it cannot provide satisfactory consistency. The network traffic cost of PeerCast is small, furthermore, PeerCast can alleviate the network congestion, idle network links could be fully utilized.

Impact of the Update Rate: For this experiment, we fixed the query inter-arrival time I_{query} to 1 second, and varied the update intervals from 1 second to 10 seconds. We ploted the query false ratio for push, adaptive pull, hybrid push-pull approaches and PeerCast. Figure 6.13 shows that query false ratio decreases with the increase of the rate between update intervals and query intervals. The message overhead for these experiments is shown in Figure 6.14. The figure shows that



(a) Number of Messages



(b) Bandwidth Usage

Figure 6.12: Network Traffic Consumption

the number of message consumption also decreases with the increase of the rate between update intervals and query intervals. Hybrid approach costs the largest overhead. Push-based invalidations impose two orders of magnitude larger overhead when compared to adaptive pull and PeerCast. PeerCast takes the minimum overhead because its overhead is only decided by the source update rates. Due to the relatively small overheads and lower query false ratio, PeerCast is more efficient than Gtk-Gnutella.

Impact of the Message TTL Value: Since TTL values determine the reach of each invalidation broadcast, query false ratio will decrease for larger TTL values in Gtk-Gnutella approach. To quantify the effect on fidelity, we varied the TTL

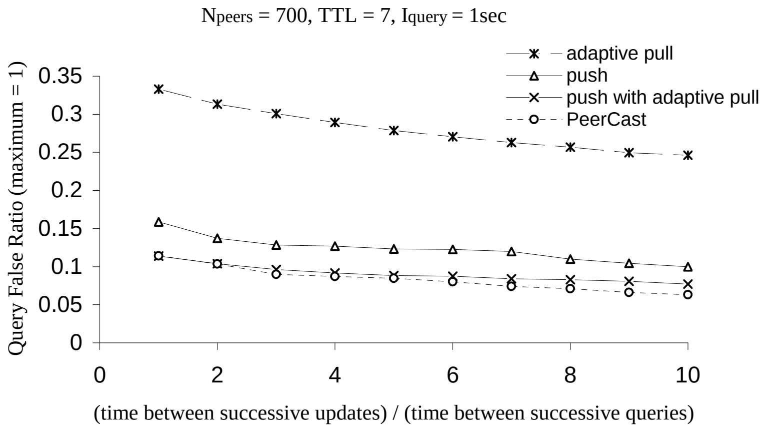


Figure 6.13: Impact of Ratio between update and query

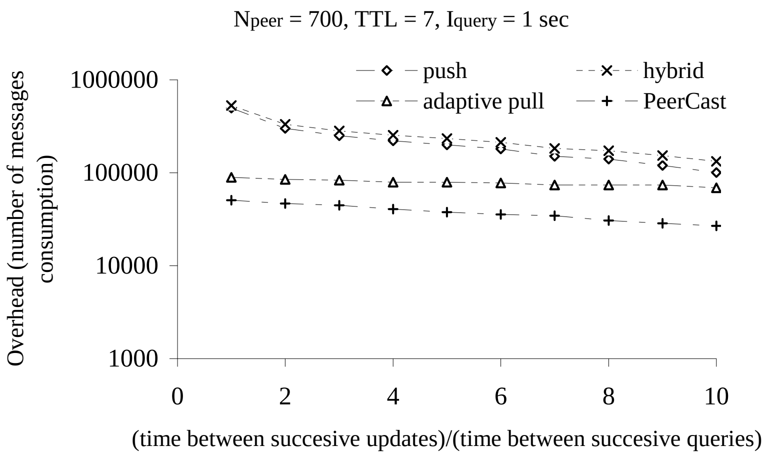


Figure 6.14: Impact of Update Rate on Message Overhead

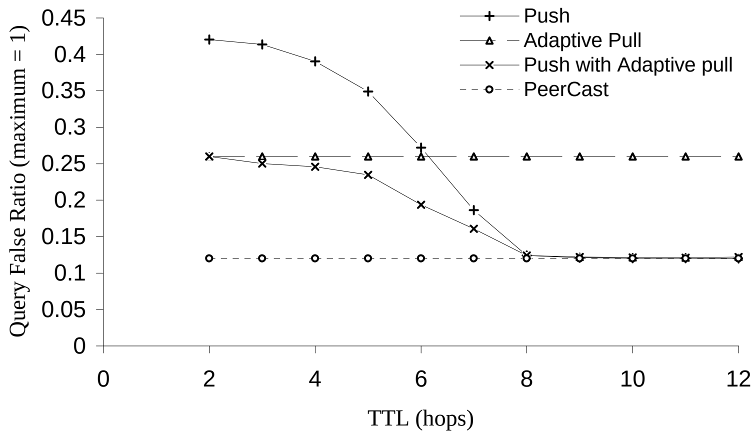


Figure 6.15: Impact of TTL values

from 2 to 12 hops and measure query false ratio for three strategies, the network size was set to 1000 peers. Meanwhile, we ploted the result from PeerCast. As illustrated in Figure 6.15, the change of TTL value has no impact to PeerCast, it is because that PeerCast maintains the date dissemination tree itself. Message flooding is avoided.

Performance on Scalability: In this experiment, we examined the scalability of PeerCast. As shown in Figure 6.16, in Gtk-Gnutella system, the workload of servers increases rapidly with the scaling of the system. On the contrast, the workload of source peers remains the same basically in PeerCast. The results show us that Gtk-Gnutella or other centralized systems suffer from the single-source overload problem. Centralized origin servers undertake heavy load to disseminate data updates to a large number of clients. Response queueing and update dissemination workload at the server greatly limit the scalability. However, PeerCast alleviates the workload of origin source servers by proportioning it to the participating peers. Source peers just disseminate data updates to their immediate child peers. Those dependent peers filter and disseminate the updates to others. Breaking the bottleneck of servers, PeerCast achieves more scalability.

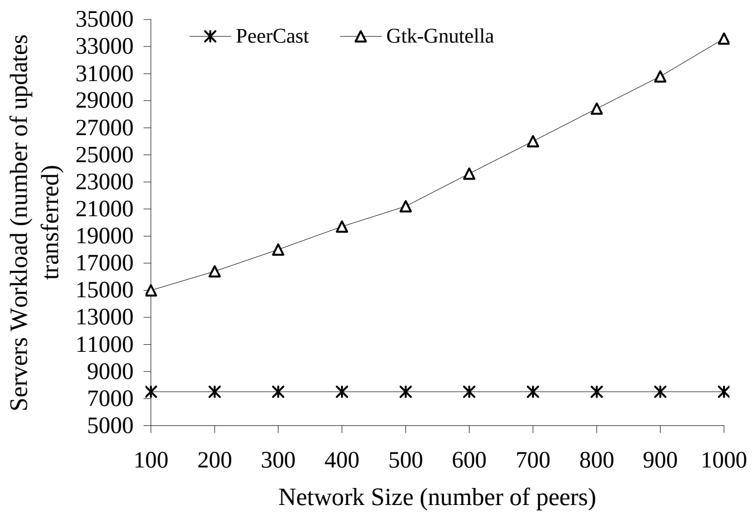


Figure 6.16: Workload on Servers

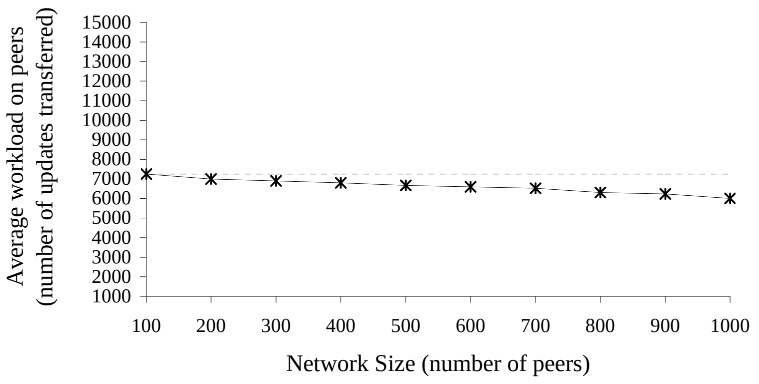


Figure 6.17: Average Workload on Peers

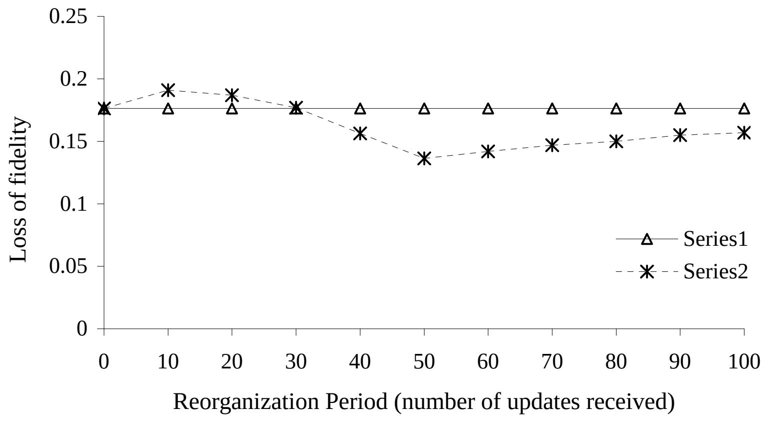
Furthermore, we also examined the average workload of peers. As shown in Figure 6.17, the average workload of peers reduce gradually with more client peers participating the overlay. The reason is that the self-adaption procedure of PeerCast could adjust the parent peer when new peer enters the disseminating tree and keep the intermediate peers well-balanced. The latter joining peers can undertake part of child peers from overloaded parent peers.

Effect of Network Reorganization: As we have discussed in Section 6.1, in the last set of experiments, we evaluated the heuristic methods to enhance the performance of PeerCast. We set the number of physical neighbors per peer to be three and five, and we vary the heuristic computation period T (measured in number of updates received) from zero to 100.

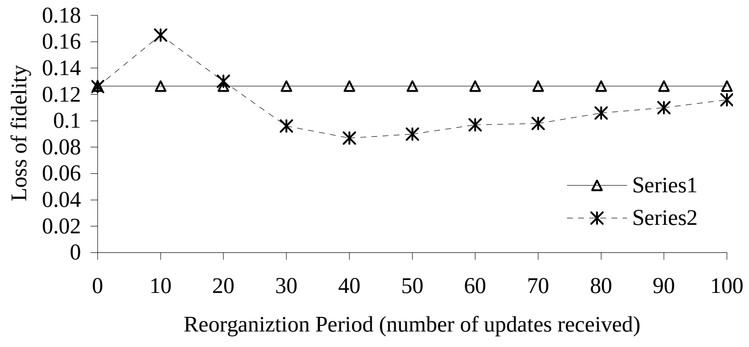
If each peer has enough physical neighbors connection to maintain, the establish and release connection cost is saved. However, each peer maintains just a number of direct connections to other neighbors. Therefore, we examine the effect of network reorganization. As shown in Figure 6.18 (a), we set the number of physical neighbors per peer to be three. Varying the heuristic computation period, we collected the results. When T = 0, heuristic method is never started. When T becomes 10, the reorganization has the negative effect to the system. This is due to the fact that there has not been enough time to gather accurate statistics and done a poor prediction. The initial network structure happened to be quite beneficial. With the incremental of T, the performance increases. Notice that when the T greater than 50, the loss of data fidelity increases slowly. The reason is that optimize procedure is performed so infrequently that it cannot predict the near future's updates correctly. In the extreme case, if T approaches infinity, heuristic method never makes a decision, it will have no effect to system.

As the number of neighbors increases, the performance of the static network. As the number of neighbors increases, the performance of the static network improves, because of the better knowledge about the contents of other peers.

Effect of Peer Node Adaption: We randomly chose small fraction, 5% and 10%, of the all peers as the churning nodes. These peers join and depart the overlay far more frequently than others. After receiving 20 updates, these peers depart the overlay in ungraceful manner, then reenter the disseminate trees. The procedure is repeated. Peer node adaption mechanism is to mark these transient peers and



(a) Three



(b) Five

Figure 6.18: Network Reorganization

push them to the edge of the topology heuristically in order to reduce their negative effect.

As shown in Figure 6.19, churning problem does negative effect to the performance of the PeerCast. It brings extra overhead to the network and leads to more latency in forwarding the updates because of the consumption to repairing the dissemination trees repeatedly. Peer adaption is a necessary mechanism to prevent churning problem.

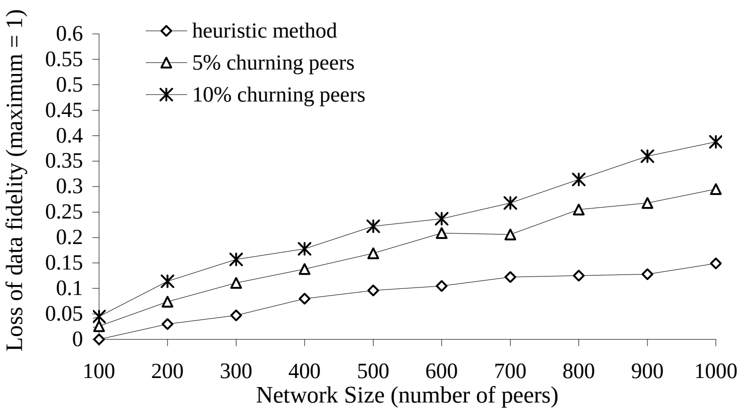


Figure 6.19: Effect of Peer Adaption

6.4 Conclusions

We have done a range of experiments to evaluate the performance of PeerCast framework. We have compared the different tree construction policies, and experimentally proved the performance of PeerCast is superior than the conventional centralized approach in large-scale network. Meanwhile, it is proved that locality-biased policy can achieve higher fidelity in data at the cost of network resource consumption. We examined the impact of client peer capacity to the system. The results showed that the capacity of client peers has great effect on PeerCast, furthermore, there is an optimal cooperation degree for each peer to contribute to system.

We also simulated the dynamic P2P network. In such scenario, we examined the impact of peer recovery mechanisms to the topology of the PeerCast, and the performance of peer recovery with leaving gracefully or ungracefully. In the end, we changed the number of backups to prove that it can achieve better fidelity in data, especially in transient situations.

In addition, we have implemented Gtk-Gnutella protocol as our objective of comparison. Gtk-Gnutella is existing cache consistency protocol designed for Gnutella system. Collected results show that although push with adaptive pull in Gtk-Gnutella can achieve the same data fidelity with PeerCast, it cost far more network traffic overhead than PeerCast. Meanwhile, TTL value impacts Gtk-Gnutella greatly. In contrast, PeerCast takes the computation and space overhead as the tradeoff to achieve the higher performance. Our experiments further indicate that the peer cooperation is essential to achieve high scalability. Our heuristic approaches are also necessary to guarantee the performance of PeerCast framework.

CHAPTER 7

Conclusion

The objective of this research is to investigate and propose optimal approach of maintaining dynamic data consistency in P2P caching systems. We present our framework PeerCast to address the major problems in achieving high fidelity of caching data in P2P systems with high-scalable, self-adaptive and fault-tolerant properties.

We have proposed dissemination overlay with different tree construction policies: randomized, round-robin and locality-biased constructions without relying low infrastructure knowledge. Our approach has been experimentally proved that it is more efficient than the conventional centralized approaches. We have extended the bounded cache techniques which have been proposed in previous centralized systems such as TRAPP [60] into P2P environment, without decentralized management or any centralized computing. Due to the demand driven delivery mechanism in PeerCast, the upper level peer can filter the data updates so as to disseminate them to dependant peers selectively. In this way, our approach can also outperform the recent approaches proposed to multicast media streams on P2P systems [25, 63] in aspects of scalability and relative delay penalty.

In PeerCast implementation, we provide two heuristic approaches to raise the performance and efficiency of the PeerCast. One is to optimize the resource usage, the other is to prevent the churning problem in overlay.

7.1 Future Work

We could extend PeerCast in several directions in our future work. First, due to the heterogeneous dynamic data popularity distributions, we could even combine some traditional consistency techniques mentioned in Chapter 2 with PeerCast, such as validation or invalidation, etc. Despite of their limitation, they cost less overhead when handling with small population. Second, we could incorporate the rate of source peers updating dynamic data factors into the data updates dissemination management policies. The prediction of updates can bring some benefits with the limited resource usage.

Second, in our current work, we implement the application layered on unstructured decentralized P2P systems. We could deploy the system on the structured P2P system, like Chord or CAN. These structured P2P systems have the routing ability, which makes data location more efficient. Thus, PeerCast could combine the structure P2P techniques into our applications. Furthermore, we consider incorporate hybrid P2P systems, super-peer architectures, like KaZaA, which combines the advantages of centralized servers, and the autonomous peers.

Last but still important point, in our current setting, PeerCast framework is lack of the fairness and cooperation incentive mechanism. We suppose that peer users are all in a friendly-cooperative manner. Free riding, or non-cooperation issues are hardly addressed. For instance, we can not punish the peers who are free riding. Peer users contribute nothing to the community despite of their high capacity. What they do is just to claim they are overloaded or have no available capacity to serve child peers, but actually, they have. Although, there have some recent work about the incentive work to study the P2P cooperation, the previous protocol can not immediately be deployed in our system, it is because our system is not designed for static data file sharing, but dynamic data item management. In order to make our system a real kill application, this is one aspect we must address in our future work.

BIBLIOGRAPHY

[1] An Exploration of Dynamic Documents, Netscape Inc. *http://home.netscape.com/assist/net_sites/pushpull.html*.

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

[3] BT. *http://www.bt.com/*.

[4] eDonkey. *http://www.edonkey2000.com/*.

[5] GNUTELLA.WEGO.COM. Gnutella: Distributed Information Sharing, 2000. *http://gnutella.wego.com/*.

[6] JXTA Advertisements. *http://people.jxta.org/stevew/jxta/advertisements.html*.

[7] Napster. *http://www.napster.com/*.

[8] SETI@home: the Search for Extraterrestrial Intelligence at home. *http://setiathome.ssl.berkeley.edu/*.

[9] SHARMAN NETWORKS LTD. KaZaA Media Desktop, 2001. `http://www.kazaa.com/`.

[10] SIM: A C++ library for Discrete Event Simulation. `http://www.cs.vu.nl/~eliens/sim/sim_html/sim.html`.

[11] The Internet Engineering Task Force. `http://www.ietf.org/`.

[12] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. World Wide Web Caching: The Application-Level View of the Internet. *IEEE Communications Magazine*, 1997.

[13] S. Bakiras, P. Kalnis, T. Loukopoulos, and W. Ng. A General Framework for Searching in Distributed Data Repositories. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2003.

[14] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, 2002.

[15] D. Barbara and T. Imielinksi. Sleeper and Workaholics: Caching Strategy in Mobile Environments. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1994.

[16] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the 5th WebDB*, 2002.

[17] D. Carney, S. Lee, and S. Zdonik. Scalable Application-Aware Data Freshening. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[18] V. Cates. Alex - A Global Filesystem. In *Proceedings of the USENIX File Systems Workshop*, 1992.

[19] Y. Chawathe. Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service. *PhD thesis, University of California, Berkeley, USA*, 2000.

[20] Y. Chawathe, S. McCanne, and E. Brewer. RMX: Reliable Multicast for Heterogeneous Networks. In *Proceedings of IEEE INFOCOM*, 2000.

[21] Y. Chawathe, S. Ratnasamy, L. BresLau, N. Lanham, and S. Shenker. Making Gnutella-like P2P System Scalable. In *Proceedings of ACM SIGCOMM*, 2003.

[22] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *International Workshop on Peer-to-Peer Systems*, 2002.

[23] M. Cherniack, M. J. Franklin, and S. Zdonik. Expressing User Profiles for Data Recharging. *IEEE Personal Communications: Special Issue on Pervasive Computing*, 2001.

[24] M. Cherniack, E. F. Galvez, M. J. Franklin, and S. Zdonik. Profile-Driven Cache Management. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[25] P. Chou, V. Padmanabhan, and H. Wang. Resilient Peer-to-Peer Streaming. *Technical Report MSR-TR-2003-11, Microsoft Research*, 2003.

[26] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of the ACM-SIGMETRICS International Conference*, June 2000.

[27] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science*, 2009, 2001.

[28] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *Proceedings of ACM SIGCOMM*, 2002.

[29] B. Cooper and H. Garcia-Molina. Studying Search Networks with SIL. In *2nd International Workshop on Peer-to-Peer Systems*, 2003.

[30] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive Push-Pull: Disseminating Dynamic Web Data. In *Proceedings of the 10th International Conference on WWW*, 2001.

[31] Peter Druschel, Frans Kaashoek, and Antony Rowstron. *Peer-to-Peer Systems*. Springer, 2002.

[32] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. *IEEE TRANSACTION ON KNOWLEDGE AND DATA ENGINEERING*, 15, AUGUST 2003.

[33] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *Proceedings of ACM SIGCOMM*, 1999.

[34] Bugra Gedik and Ling Liu. PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[35] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, 1989.

[36] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can P2P do for database, and vice versa? In *Proceedings of WebDB Workshop*, pages 171–182, June 2001.

[37] Y. Guo, K. Suh, J. Kurose, and D. Towsley. P2Cast: Peer-to-Peer Patching Scheme for VoD Service. In *Proceedings of the 12th International Conference on WWW*, 2003.

[38] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate Range Selection Queries In Peer-to-Peer Systems. In *Proceedings of the 1st CIDR*, 2003.

[39] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, 1996.

[40] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the Structure Chasm. In *Proceedings of the 1st CIDR*, 2003.

[41] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *Proceedings of the 19th ICDE*, 2003.

[42] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, 1994.

[43] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th Conference on Very Large Data Bases*, 2003.

[44] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A Decentralized Peer-to-Peer Web Cache. In *Proceedings of the 21st annual symposium on Principles of Distributed Computing*, 2002.

[45] J. Jannotti, D. K. Gifford, and K. L. Johnson. Overcast: Reliable Multicasting with an Overlay Network. In *USENIX Symposium on Operating System Design and Implementation*, 2000.

[46] Chenqing Jin, Weining Qian, Chaofeng Sha, Jeffrey Xu Yu, and Aoying Zhou. Dynamically Maintaining Frequent Items over A Data Stream. In *Proceedings of ACM CIKM*, 2003.

[47] P. Kalnis, W. Ng, B. Ooi, D. Papadias, and K. Tan. An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2002.

[48] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proceedings of the ACM SIGMOD*, 2003.

[49] P. Keyani, B. Larson, and M. Senthil. Peer Pressure: Distributed Recovery from Attacks in Peer-to-Peer Systems. In *Web Engineering and Peer-to-Peer Computing Workshops*, 2002.

[50] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Computer Networks and ISDN Systems*, volume 30, August 1998.

[51] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rheaand H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*, 2000.

[52] Wang Lam and Hector Garcia-Molina. Multicasting a Changing Repository. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[53] J. Lan, X. Liu, P. Shenoy, and K. Ramamritham. Consistency Maintenance in Peer-to-Peer File Sharing Networks. In *Proceedings of the 3rd IEEE Workshop on Internet Applications*, 2003.

[54] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of ICDCS*, 1997.

[55] J. Liu, X. Zhang, B. Li, Q. Zhang, and W. Zhu. Distributed Distance Measurement for Large-Scale Networks. *The International Journal of Computer and Telecommunications Networking*, 41:177 – 192, 2003.

[56] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V.Jacobson. Adaptive Web Caching: towards a new global caching architecture. In *3rd International WWW Caching Workshop*, 1998.

[57] W. Ng, B. Ooi, Y. Shu, K. Tan, and W. Tok. Efficient Distributed CQ Processing using Peers. In *Proceedings of the 12th International Conference on WWW*, 2003.

[58] W. Ng, B. Ooi, and K. Tan. Bestpeer: A Self-Configurable Peer-to-Peer System. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[59] W. Ng, B. Ooi, K. Tan, and A. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proceedings of the 19th ICDE*, 2003.

[60] Chris Olston and Jennifer Widom. Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 144–155, 2000.

[61] Li Pan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of ACM SIGCOMM*, 1998.

[62] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, 2001.

[63] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. *Application-level Multicast using Content-Addressable Networks.* Lecture Notes in Computer Science, 2001.

[64] V. Roca and A. El-Sayed. A Host-Based Multicast (HBM) Solution for Group Communications. In *1st IEEE International Conference on Networking*, 2001.

[65] P. Rodriguez and E. Biersack. Continuous multicast distribution of web documents over the internet. In *IEEE Network Magazine*, volume 12, 1998.

[66] P. Rodriguez, K. Ross, and E. Biersack. Improving the WWW: Caching or Multicast? In *Proceedings of Computer Networks and ISDN Systems*, 1998.

[67] Pablo Rodriguez and Sandeep Sibal. SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution. In *Proceedings of the 9th International Conference on WWW*, pages 33–49, 2000.

[68] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *IFIP/ACM Middleware*, September 2001.

[69] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM symposium on Operating systems principles*, 2001.

[70] O. Sahin, A. Gupta, D. Agrawal, and A. Abbadi. A Peer-to-Peer Framework for Caching Range Queries. In *Proceedings of International Conference on Data Engineering*, 2004.

[71] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22:299 – 320, Dec 1990.

[72] S. Shah, S. Dharmarajan, and K. Ramamritham. An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[73] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining Consistency of Dynamic Data in Cooperating Repositories. In *Proceedings of the 28th Conference on Very Large Data Bases*, 2002.

[74] Scott Shenker. The Data-Centric Revolution in Networking. *Keynote in VLDB 2003 Conference.*

[75] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM*, 2001.

[76] X. Wang, W. Ng, B. Ooi, K. Tan, and A. Zhou. BuddyWeb: A P2P-based Collaborative Web Caching System. In *Web Engineering and Peer-to-Peer Computing: Networking 2002 Workshops*, 2002.

[77] Kurt Worrell. *Invalidation in Large Scale Network Object Caches.* Master's thesis. University of Colorado, Boulder, 1994.

[78] B. Yang and H. Molina. Improving Search in Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002.

[79] B. Yang and H. Molina. Designing a Super-peer Network. In *Proceedings of the 19th International Conference on Data Engineering*, 2003.

[80] Cheng Yang. Peer-to-Peer Architecture for Content-Based Music Retrieval on Acoustic Data. In *Proceedings of the 12th International Conference on WWW*, 2003.

[81] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, 1998.

[82] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, 11:563–576, 1999.

[83] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[84] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. In *U. C. Berkeley Technical Report UCB//CSD-01-1141*, 2001.

[85] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *Proceedings of ACM NOSSDAV*, 2001.