INDEXING IN PEER-TO-PEER SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Prakash Linga

May 2007

INDEXING IN PEER-TO-PEER SYSTEMS

Prakash Linga, Ph.D.

Cornell University 2007

Peer-to-Peer systems are large scale distributed systems whose component nodes participate in similar roles and hence are “peers”. Peer-to-peer systems have generated a lot of interest because of their scalability, fault-tolerance and robustness properties. The peer-to-peer paradigm was first popularized by file sharing systems like Napster, Kazaa and BitTorrent. It is increasingly being used in an enterprise setting to enable highly scalable applications using low cost commodity clusters. Amazon S3 is one such example that uses peer-to-peer technology to provide a simple scalable storage service.

With large number of peers and large amounts of data, one of the questions of fundamental interest in a peer-to-peer system is: how to find relevant data quickly? In this thesis, I present efficient peer-to-peer indices that support lookup of relevant data quickly. My thesis contains (1) Kelips, an efficient Distributed Hash Table (DHT), (2) Kache, a cooperative caching application, and (3) r-Kelips, an efficient peer-to-peer range index. In addition to complex range query support, demanding applications like transaction processing and military applications require strong correctness/availability guarantees. The final part of my thesis contains techniques that provably guarantee correctness and availability of peer-to-peer range indices.

BIOGRAPHICAL SKETCH

Prakash Linga, LP, Linga, Linga Prakash, L. Prakash and Prakash are the usual names that I respond to. If the above list is too small, some of my more esoteric names are: Plinga, Tilda-linga, PL222 and Lingam! I was born in a small town called Suryapet in India. The only claim to fame of this town is that it is equidistant from two big cities (135km from Hyderabad and Vijayawada)! Fortunately, my parents decided to live in Hyderabad where I did all my schooling. Like all Hyderabadis, I love Hyd. The unique dialect (probably a unique language that is a mix of Telugu, Urdu and Hindi), creative blend of northern and southern cultures, the hustle and bustle, and the list continues... After an adventurous life at Hyderabad, I moved to Chennai for four years of rather sedentary life at IIT Madras. I had my share of fun in eating contests and brain games. After undergrad, against all odds, I decide to go to grad school leaving two lucrative job offers. Guess where I end up? Ithaca, NY - a small city that is equally far away from many big cities! Even though I missed city life, I had a blast at Ithaca!!! I made a lot of friends that I could trouble, nurtured my enthusiasm for outdoor activities (could not avoid being badly injured, twice!), developed my professional interests, and identified my personal goals - all in the six years at Ithaca. My wife and I are very eagerly awaiting our stay in sunny California, far away from harsh Ithacan winters!

Amma, Daddy, Prathima and my dear wife, Binu

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help and suggestions of my advisers - Prof. Jayavel Shanmugasundaram, Prof. Johannes Gehrke and Prof. Ken Birman. My heartfelt thanks to my thesis adviser, Prof. Jayavel Shanmugasundaram, for his advise, guidance and support. I am especially grateful to Prof. Johannes Gehrke and Prof. Ken Birman, for being closely involved with my research and supporting me throughout my PhD journey. I would like to thank my other committee members, Prof. Alan Demers and Prof. Zygmunt Haas, and proxy committee members, Prof. Robbert Van Renesse and Prof. Radu Rugina, for their help.

Many thanks are due to my colleagues - Indranil Gupta, Adina Crainiceanu, Ashwin Machanavajjhala, Biswanath Panda and Vikram Krishnaprasad - for their insightful comments, long discussions and the long-nighters!

Cornell Computer Science department is a friendly and cozy place. I would like to acknowledge the help of Becky Stewart, Bonnie Maine, Cay Wilson, Stephanie Meik and Stacey Shirk in making my stay at this department comfortable and pleasant.

My sojourn in Ithaca was a memorable experience largely due to my friends - 1d, Anima, K, Kristin, Lavanya, Muthu, Panda, Pappu, Sattu, Smython, Vivek and Vugz. My heartfelt thanks to my friends, especially 1d, K, Smython and Vugz. My PhD journey would have been impossible without their love and support.

I want to express my deepest gratitude to my family for their love and support. My heartfelt thanks to my mom, dad and sister for always being there for me. I wish to take this opportunity to thank my dad, Prof. Venkateswar Rao Linga, for being an inspiration to strive for the best. Special thanks to my dear wife, Binu, for putting up with me during my good and bad days of grad life. I would like to thank my extended family in NJ, especially my little cousins - Ammalu, Aravind, Niki, Pranith and Pratika, for their love and support. My weekends in NJ are some of the best times of my life so far!

In addition to all the above individuals, I am indebted to several colleagues and friends who have assisted me in one way or the other. Thank you to one and all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Peer-to-peer Systems

1.1 Introduction

Peer-to-Peer systems have emerged as a promising paradigm to structure large scale distributed systems. There has been a lot of interest in peer-to-peer systems because of their scalability, fault-tolerance and robustness properties. Our goal as part of the PEPPER project at Cornell is to build a database infrastructure for peer-to-peer systems. We envision a future where users will publish semantically rich, semi-structured data. Users can also contribute storage and computational power to the distributed data management system. They should be able to query the data in this "P2P data warehouse" as if the data was stored in one centralized database system.

Traditional client-server architecture imply a sharp distinction between clients and servers. Clients request and consume services, and servers provide services. Popular servers can become hotspots. Client-server systems are therefore simple and easy to build, but suffer from scalability and fault-tolerance problems. On the other hand, key advantages of peer-to-peer systems are (1) scalability, due to resource-sharing among cooperating peers, (2) fault-tolerance, due to symmetrical nature of peers, and (3) robustness, due to self-organization after failures.

Napster introduced the peer-to-peer paradigm of collaboration between peers to share and download files quickly. Many commercial applications like Skype, NetMeeting and Amazon S3 have successfully used this paradigm. There has also been a flurry of academic research in peer-to-peer systems following the tremendous success of Napster. The main factor distinguishing traditional distributed systems and peer-to-peer systems is the involvement and empowerment of peers.

With large number of peers and large amounts of data, one of the questions of fundamental interest in a peer-to-peer system is: how to find relevant data quickly? In this dissertation, I address the issue of building efficient indices to allow for users to access relevant data quickly. The main challenges in designing such indices are scale and dynamism of the system. Many p2p applications require the system to scale to thousands of peers and millions of data items. Further, peers can join, leave or fail at any time (peer churn), and items can similarly be inserted or deleted at any time (item churn). Finally, providing correctness and availability guarantees in such a dynamic system presents new challenges.

1.2 Accomplishments in this Dissertation

The following is an overview of our results:

1. Kelips: An efficient, scalable, fault-tolerant and distributed index that supports equality queries in a peer-to-peer environment.

2. Kache: A cooperative caching scheme that provides low lookup latency and balances the load among peers.

3. r-Kelips: An efficient p2p range index that guarantees load balancing among peers even in the presence of skewed data and query workloads, while still guaranteeing the correctness and availability of queries.

4. Correctness and Availability in P2P range indices: Defining and guaranteeing correctness and availability of more complex peer-to-peer indices, such as p2p range indices.

In summary, this work addresses multiple issues surrounding building indices for a large scale distributed system. Efficiency, load balance, fault-tolerance, robustness, correctness, and availability are some of the design issues considered. This work is a first step towards building a peer-to-peer database. One promising application that can be scaled to large amounts of data using p2p databases is data warehousing. This application does not need the strong guarantees provided by a traditional database and hence can use highly available p2p databases to scale to peta-bytes of data. Many companies are now interested in building highly-available, highly-scalable shared-nothing databases to enable new data analytics applications.

Chapter 2

Related Work

Peer-to-peer systems are desirable because of scalability, fault-tolerance and robustness properties. For scalability reasons, we want the index to be fully distributed. We first survey work done on developing scalable distributed datastructures as part of the distributed databases community (Section 2.1). These techniques are not fully distributed and hence are not suitable for a p2p environment. In the rest of this chapter, we look at fully distributed proposals for indices in p2p systems.

Peer-to-peer indices have the following design parameters:

- Query Expressiveness: Distributed Hash Tables (DHTs) are structured p2p systems that support equality queries efficiently. In Section 2.2.1, we summarize work on DHTs. DHTs cannot be used to answer range queries efficiently because of the use of a hash function. We then survey work on structured p2p indices that support range queries efficiently (Section 2.2.2). In Section 2.2.3, we summarize work on more complex queries like keyword searches.

- Query Correctness: Demanding applications like transaction processing and military applications require both complex range predicates (to search for objects within a region) and strong correctness/availability guarantees. In Section 2.3, we present work on correctness of query results in a p2p environment.

- Load Balancing: The assumptions under which load balancing is desired are varied. In Section 2.4, we survey work on load balancing on p2p indices.

Finally, in Section 2.5, we survey work on p2p applications like File Sharing, Name Services and Cooperative Caching. These applications are built using p2p indices and hence are scalable and fault-tolerant.

2.1 Semi-Distributed Indices

Distributed datastructures like LH* [LNS93] were developed in the database community for efficient file access on a network of interconnected computers (multicomputers). However, most of these techniques maintain consistency among the distributed replicas by using a primary copy, which creates both scalability and availability problems when dealing with thousands of peers. Some index structures, however, do maintain replicas lazily [Lom96]. However, these schemes are not designed to work in the presence of peer failures, dynamic item replication and reorganization, which makes them inadequate in a P2P setting. In contrast, our algorithms are designed to handle peer failures while still providing correctness and availability guarantees.

Scalable Distributed Data Structures (SDDS) is one class of data structures that is defined for multicomputers. Data is stored on computers called *servers* and is accessed from computers called *clients*. A computer could be both a server and a client. Clients insert, delete or search for items in a file (given primary keys or OIDs), and the servers store the file and process clients' queries. Clients do not store the actual records but only store some file access computation parameters (like the parameters of the actual dynamic function in LH*) that will be useful for accessing the file. Every SDDS must respect three design requirements.

1. No central directory: A central directory is not used for data addressing.

2. Client autonomy is preserved: Client image is updated only through messages called Image Adjustment Messages (IAM). These are only sent when a client makes an addressing error.

3. Correct routing: A client with an outdated image can send a key to an incorrect server, in which case the structure should deliver it to the right server, and trigger an IAM.

LH* is an example of a hash-based SDDS. RP* is an example of an ordered SDDS. We will now discuss some of these distributed datastructures in more detail.

LH* [LNS93]: LH* generalizes linear hashing to a distributed setting. This scheme scales well with number of objects and number of servers. An LH* file can be retrieved much faster than a regular single site disk file and/or can hold a much larger number of objects. File is organized into buckets (pages) on the disk or on RAM. Just like in linear hashing, buckets are addressable through a directoryless pair of hash functions, h_i and h_{i+1} ($i = 0, 1, 2, \cdots$). Hash function h_i maps OID (key) to 2^i addresses. A special value n is used to determine which of h_i or h_{i+1} should be applied to a given object. Under insertions, file expands gracefully by splitting one bucket at a time into two buckets. This is essentially replacing h_i with h_{i+1}, and is done one bucket at a time. When a bucket splits, it recruits a new server to store the newly created half. It is assumed that such a server can always be found. The split is coordinated by one special site, called the *split coordinator* to ensure serialization of splits. The authors mention that the split coordinator could be eliminated and several splits could be executed in parallel. Bucket addresses are mapped to statically or dynamically allocated server addresses. The split coordinator is necessary for dynamic mapping of bucket addresses to server addresses.

Clients maintain a possibly outdated view of the file: (i, n). It uses this view to compute the address of the server storing a given item. On receiving a query, server first checks if it is indeed the destination. If yes, the query is processed there. Otherwise, the server performs a new address computation and forwards the query to the server with the new address. The authors show that the number of messages per lookup is two in general and four in the worst-case. Client view of a file is updated through Image Adjustment Messages(IAM), as mentioned before.

The first LH* structure proposed does not support failure of any server. More recent papers present new variants of LH* (LH*g, LH*RS) that can support failures of a limited number of servers while maintaining the availability of data [LS00]. The main idea here is to store parity objects and reconstruct lost objects using coding theory techniques. However, servers still cannot join or leave the system at will. Our goal is to design index structures that can handle peer churn gracefully.

RP* [LNS94]: RP* is an example of an ordered SDDS. The authors present three structures: RP_N^*, RP_C^*, RP_S^*, designed for answering range queries in a distributed setting. In these datastructures, records are partitioned into buckets based on the key value. The records are sorted and each bucket stores the records that fall into its range. Every bucket is stored at a different server. In RP_N^*, the basic idea is to avoid indices through the use of multicast. The authors assume a local network model where the diameter is small, enabling use of multicast and/or broadcast. RP_C^* attempts to enhance throughput for faster networks by adding indices on clients and RP_S^* does the same by having indices on both clients and servers. As mentioned before, the authors assume a very restricted model where multicast and broadcast operations are practical. Also, the servers are assumed to remain available for the entire duration of the system. This model is therefore very different than our model, where thousands of potentially faraway peers can be in the system at any time and peer churn cannot be ignored.

DRT [KW94]: DRT is a distributed binary search tree for storing and querying data in a distributed environment. An important property of this structure is that a server that was once responsible for a range always knows how to route the requests for that range using its subtree. This allows for lazy propagation of structure updates. Moreover, since interior nodes of the tree never change, insertions and deletions have only localized effect and the structure supports concurrent insertions, deletions and searches. However, the global tree constructed is not balanced and in the worst case, the height of the tree is linear in the number of data items stored in the system. In the worst case, the number of messages needed to answer a request is linear in the number of servers in the system. Unless the clients construct local images of the global tree, the initial server having the root of the tree will be overloaded since all requests start with the root. If the clients/servers construct local images of the tree (nodes are replicated), the space needed for the index increases. Finally, the assumption that servers never crashes or leave makes the structure unsuitable for our purposes.

db-tree [JK93]: There has been a lot of proposals on distributed B+-tree like structures for a distributed environment [JC92a], [JK93], [Lom96]. We will discuss one representative example here. db-tree [JK93] is an example of a distributed B-tree. The idea is to replicate internal nodes to reduce message passing and increase parallelism. The leaves are distributed among the processors (peers), but are not replicated. Any processor that stores a leaf node also stores the nodes on the path from the root to the leaf node. Like in a B-link tree, internal nodes have pointers to the children as well as to the left and right siblings. db-tree algorithms build on the concurrent B-link algorithms. A primary copy is used to ensure that all replicas converge to the same state. All update actions are decomposed in two actions: initial action (executed at the first of a set of copies) and relayed action (propagated to the other copies). The authors do not consider processor failures or system crashes (peer failures). Moreover, they assume the existence of a primary copy that needs to be available all the time and needs to know the location of all replicas. This creates scalability and availability problems in a highly dynamic p2p environment with thousands of peers.

dPi-tree [Lom96]: Compared with some of the earlier approaches, the dPi-tree scheme presented in this paper is very appealing. In this scheme, convergence of the replicated index nodes is not required and lazy update of replicas is possible. The approach taken in this paper is to construct a distributed version of Pi-tree. Pi-tree is a multi-dimensional, high concurrency index tree that also supports recovery. Pi-tree can be thought of as a generalization of B-link trees to higher dimensions. Both these index structures have sibling pointers that help separate node splitting from the posting of the index term of the new node, thereby increasing concurrency. As before, only internal nodes of a Pi-tree are replicated. Replicated Pi-tree indices need not be coherent. The main insight is: information about changes to the index is propagated lazily (only on messages in which exchange of data is required in any event). Each replica can maintain itself based only on this information and hence index replicas need not exchange messages to ensure structure convergence. The authors argue that such convergence is not required because of sibling pointers. The main drawback of this approach is the fact that it does not allow for peer (site) leaves/failures, which makes it inadequate for a p2p setting.

2.2 Fully Distributed Indices

The indices present in this section are fully distributed i.e they do not assume any kind of centralized control. These systems do not suffer from the drawbacks of SDDS like existence of a primary copy and work fine in the presence of peer failures.

2.2.1 P2P Equality Indices : DHTs

DHTs are *structured* indices that support equality queries only. Chord [SMK$^+$01], Pastry [RD01b], Tapestry [ZKJ01], CAN [RFH$^+$01], Kelips [GBL$^+$03], Viceroy [MNR02], Kademlia [KK03a] and Koorde [KK03b] are some examples of DHTs proposed in the literature. DHTs address some of the problems associated with unstructured p2p indices. In particular, search and communication costs are reasonable. DHTs can be extended to provide correctness and availability guarantees for equality queries. DHTs however cannot be used to support range queries efficiently.

Chord [SMK$^+$01]: Chord is a simple DHT that support insert, delete and lookup operations. Chord guarantees a worst case search cost that is logarithmic in the number of peers, using space that is also logarithmic in the number of peers. Peer addresses and keys are hashed to the same identifier space (say, 0 to $2^m - 1$). The peers are arranged in a virtual ring in the identifier space. The identifier length m is chosen such that the probability of two different keys or peers mapping to the same identifier is small. A key is stored with the first peer having a peer id equal to or greater than the key id. Each peer has pointers to its successor and its predecessor and also to peers that are 2^i ($i = 1$ to $m-1$) hops away on the identifier space. Routing a message involves forwarding it to the farthest peer in the finger table that is before the target peer. Search time is logarithmic in the worst case, assuming all the finger tables are accurate. Note that search will succeed when the finger tables are not accurate, but the worst case cost is no longer logarithmic. When a peer wants to join the system, it contacts some peer that is already a part of the system. This peer will forward the join message to the peer with peerId numerically closest to the joining peer. Each peer runs the stabilization protocol periodically to fix the finger table entries and complete the join of peers into the ring. When a peer fails, it's neighbors in peerId space detect its departure and fix the finger table at the lowest level. The stabilization process then fixes the other entries in the finger table appropriately.

Pastry and Tapestry [RD01b, ZKJ01]: Pastry [RD01b] and Tapestry [ZKJ01] use similar algorithms for routing. We will therefore discuss only Pastry here. Like Chord, Pastry guarantees a worst case logarithmic search performance in a stable system. Each peer in a Pastry network has a unique 128-bit peer id. This is obtained by applying a hash function to the peer address (say, IP address). Pastry provides a routing mechanism to efficiently route a message to peer with peerId numerically closest to the key. The peerId is viewed as a number in base 2^b. $(i, j)^{th}$ entry of the routing table is the target peer which agrees with the peerId in first i bits and whose $i + 1^{th}$ bit is j. This can be used to route any message in at most logarithmic number of hops. In addition to the routing table, 2^b immediate left and right neighbors (according to the peerId) are stored in the leaf set. Also, there is a neighborhood set that stores $2{*}2^b$ peers which are closest to this peer according to the proximity metric (say, rtt).

When a peer X wants to join the system it will first find and contact peer A that is close to X and is also a member of the system. Peer A will issue of a join request for peer X. This request reaches peer Z, the peer in the system with peerId closest to X. Peer X gets its leaf set from Z and neighborhood set from A. It gets its routing table candidates from the peers in the path from A to Z. X sends a copy of its state to all the peers it knows of. The recipient peers intern updates their state based on the fact that peer X is now in the system. Peer departures are detected by neighbors in peerId space. When a peer is detected to be failed, the farthest peer in the same half of the leaf set is contacted and a new candidate is chosen. Peers with the failed peer as a routing table entry will detect the failure when they are trying to route to that peer. On detecting a failed peer, the peer finds an alternative for this entry by contacting some other peer with an entry in the same row. Locality can be considered when constructing the routing table and the authors argue that route chosen for a message is likely to be good with respect to the proximity metric.

CAN [RFH+01]: CAN is a scalable, fault-tolerant and self-organizing distributed hash table where each node stores a part of the hash table. CAN is a scalable P2P indexing system with applications in other areas like large scale storage management systems.

Basic idea is to consider a virtual d-dimensional Cartesian co-ordinate space on a d-torus with each node owning a region of this space. This space is used to store <key, value> pairs. Suppose node n is inserting a $< key, value >$ pair into the system. The $< key, value >$ pair is mapped onto a point P in the co-ordinate space using a hash function. If n owns the region of space containing P, then the <key, value> pair is stored at n. Otherwise, the request is routed along the straight line path from source (this node) to destination (the node which owns the space containing point P). Lookup for a <key, value> pair works similarly given that you know the key. Coming to node insertions (node picks a random point P which it decides to cover): new node first finds a node already in CAN; then using CAN routing mechanism finds a node whose zone will be split; the neighbors of the split zone are notified so that routing can include the new node. Node deletions: when a node is deleted the space it used to own is merged with that of one of its neighbors (one which detects this node is dead or one with least space) and the rest of the neighbors are informed about it. Simulation results show that system is scalable and fault-tolerant. The average number of messages sent for a search is $O(N^{1/d})$. This structure is more flexible than the Chord structure because it allows a tradeoff between search speed and maintenance cost, by changing d.

2.2.2 P2P Range Indices

In this section, we will look at proposals for p2p indices supporting range queries. All these indices [AS03, GBGM04, BRS04] give logarithmic search guarantees.

Approximate Ranges [GAE03b]: This paper is a first attempt to answer range queries in a p2p system. Peer nodes are hashed into the identifier space using a hash function (SHA). The range specifying the data partition is also hashed into the same identifier space using locality sensitive hashing. Locality sensitive hashing ensures that similar ranges are hashed to the same identifier with high probability. Like in Chord, partition with identifier i is mapped to the peer with least identifier not less than i in the circular identifier space. Chord finger table pointers are used to locate a peer responsible for a certain identifier. Given a query range Q, l identifiers are computed for Q and peers holding those identifiers are contacted. There can be at most l different peers holding the identifiers. Parameter l is chosen such that at least one of the l peers will have data relevant to the query range. Each contacted peer checks the list of partitions that it has associated with the identifier and finds the best match for the query partition in the list and sends the best match to the requesting peer. The requesting peer can now choose the best match from the l replies it gets, and contact the peer with that partition for the data of the partition. The main drawback of this approach is that it provides approximate answers to range queries in a p2p system.

P-trees [CLGS04]: In this paper, we propose P-trees: a new distributed, fault-tolerant index structure for P2P systems. P-trees can be used to answer equality, range, prefix match and one-dimensional nearest neighbor queries in a P2P system. P-trees are essentially distributed B+-trees with enough redundancy for fault-tolerance. Each peer stores the nodes in its path from root to leaf. Search is like in B+-trees but uses pointers over the network. P-trees are implemented both in a simulator and a real distributed implementation using C#. Experimental results show that P-trees perform well in a dynamic environment, with the average cost per operation (equality search, insertion, deletion) being logarithmic in the number of data items (and peers) in the system. P-trees work under the assumption that there is a single data item stored at each peer, so the number of data items in the system equals the number of peers. In the multiple data items per peer case, Ptrees can be used by creating one *virtual* peer for each item and replacing the peers with *virtual* peers in the indexing structure. The main drawback of this approach is that search is no longer logarithmic in the number of peers but is logarithmic in the number of data items which is potentially much larger than the number of peers.

SkipGraphs [AS03]: Skip graphs is a randomized, distributed tree like index based on skip lists to answer equality and range queries efficiently in a p2p system. Unlike skip lists or other tree like data structures, skip graphs are highly resilient and tolerate a large fraction of failed nodes with loosing connectivity. Skip list is a randomized balanced tree datastructure organized as a tower of increasingly spare linked lists. There is one linked list per level. Level 0 is a linked list of all nodes in increasing order by key. For each $i > 0$, each node in level $i - 1$ appears in level i independently with some fixed probability p. Skip graph is a generalization of skip lists with enough redundancy to handle failures. Skip graph is a trie of skip lists that share their lower levels. Search algorithm for skip graph is same as that for skip list, except for minor modifications to adapt to the distributed setting. SkipGraphs suffers from the same shortcoming as Ptrees, which is the limitation of one item per peer.

PRing [CLM+04a]: In this paper, we propose a novel index structure called P-Ring that supports both equality and range queries, is fault-tolerant, provides guaranteed search performance, and efficiently supports large sets of data items per peer. The P-Ring data store achieves load balance within a factor of $2 + \epsilon$ with constant amortized insertion and deletions cost. The cost metric is the number of tuples moved by the algorithm per insert or delete. Additionally, we also propose a new content router, the Hierarchical Ring, which provides efficient access to data even when the data distribution is arbitrarily skewed. A real distributed C++ implementation of P-Ring has been deployed on PlanetLab on 100 peers distributed all over the world.

Online Load-Balancing [GBGM04]: Ganesan et al. consider the problem of horizontally partitioning a dynamic relation across thousands of peers by the use of range partitioning. They propose efficient and asymptotically optimal algorithms that ensure storage balance across peers, even against adversarial insertion and deletion of items. The cost metric for a load balancing algorithm used in this paper is the number of tuples moved by the algorithm per insert or delete. Two operators are used for load balancing - (1) NBRADJUST: If a node is responsible for too much data, it can attempt to balance out the load with its neighbor by moving a part of its data. (2) REORDER: A node n that is responsible for too much data can recruit a lightly loaded node m to help out. The lightly loaded node m will first transfer its data to its neighbor. Then the original node n will transfer some of its data to this newly recruited node m and m will now be the n's new neighbor. These two operations are universal in that they can together be used to efficiently implement any load-balancing algorithm. The following threshold algorithm for load balancing is proposed in this paper: a node attempts to shed its load whenever its load increases by a factor δ, and attempts to gain load when it drops by the same factor.

Mercury [BRS04]: Bharambe et al. proposed Mercury, a randomized index structure that supports scalable multi-attribute range queries and performs explicit load balancing. Main contributions of the paper are: (1) A low-overhead random sampling algorithm that allows each node to create an estimate of system-wide metrics such as load distribution. (2) A message routing algorithm that supports range-based lookups within each routing hub in $O(log^2 n/k)$ hops when each node maintains k links to other nodes. (3) A load-balancing algorithm (which exploits the random sampling algorithm) that ensures that routing load is uniformly distributed across all participating nodes. (4) An algorithm for reducing query flooding by estimating how selective each of the predicates in a query is, based on past database insertions. A sampling mechanism is used to estimate the distribution of peers in the value space and then long links are chosen based on the harmonic distribution. Like in PRing, nodes in a hub H_a corresponding to attribute a are arranged in a circular overlay with each node responsible for a contiguous range r_a of attribute values. Queries are passed to exactly one of the hubs corresponding to the attributes that are queried. Within the chosen hub, the range query is delivered and processed at all nodes that could potentially have matching values. Range query processing is similar to that in PRing - route to first value in the range and then continue along the ring.

They leverage on the histograms of node properties (like load) to help implement load balancing. First, each node can use histograms to determine the average load existing in the system, and hence, can determine if it is relatively heavily or lightly loaded. Second, the histograms contain information about which parts of the overlay are lightly loaded. Using this information, heavily loaded nodes can send probes to lightly loaded parts of the network. Once the probe encounters a lightly loaded node, it requests this lightly loaded node to gracefully leave its location in the routing ring and re-join at the location of the heavily loaded node. This leave and re-join effectively increases the load on the neighboring (also likely to be lightly-loaded) nodes and partitions the previous heavy load across two nodes. Unlike P-Ring, Mercury only provides probabilistic guarantees even when the index is fully consistent.

Caching Range Queries [SGAA04]: This paper presents a scheme based on CAN to answer range queries by caching answers to previously issued range queries. The virtual hash space is divided into rectangular zones and each zone is maintained by one (active) node. The zones are maintained dynamically using passive nodes. As in CAN each node maintains a neighbor list. Search request is routed towards the target zone by routing it to the neighbor closest to the target zone. When the search request reaches the target zone, stored results in this zone are used to compute the answer to the range query. Search request could be forwarded from the target zone to other zones that could have the satisfying tuples. It is possible that a large number of zones need to be contacted to answer a range query with small number of results. The worst case complexity of answering a range query with small number of results is also very high.

2.2.3 Other P2P Complex Queries

Unstructured P2P Systems: The main functionality provided in unstructured p2p systems is support for equality and keyword queries in a decentralized fashion. These systems do not suffer from the drawbacks of SDDS like existence of a primary copy and work fine in the presence of peer failures. However, there are many drawbacks of using unstructured p2p systems to answer queries in a p2p setting. First, search is expensive. Second, an item satisfying the query is not guaranteed to be found, even if it exists in the system. Finally, communication costs are high. In our work, we provide richer query semantics and stronger search guarantees.

Gnutella [webc]: Gnutella is an example of an *unstructured* p2p system and relies on flooding. Every node/peer exports the data it wants to share with other peers in the system. Each request is associated with a TTL and the request is forwarded to all neighbors of the initiator node. Every node receiving a request decrements the TTL and forwards the request to its neighbors. Forwarding of the request continues as long as TTL is positive. Lookup time can be controlled using TTL but the bandwidth requirements are tremendous.

Freenet [ICH00]: This is a peer-to-peer application that supports insert, delete and lookup operations for files, while protecting the anonymity of both the authors and the readers. Files are identified by file keys. File keys are obtained by hashing file names. Keyword-signed key (KSK), signed-subspace key (SSK) and content-hash key (CHK) are three types of file keys discussed in the paper. In KSK, a descriptor text is used as input to generate a public/private key pair; the public half is hashed to get the file key. Each node makes a local decision about where to forward a request to. Each node has knowledge of only immediate upstream and downstream neighbors in the proxy chain to maintain privacy. Nodes closer together store files with closer file keys. When a node gets a request for a file key it checks if it knows where this file key is stored. If not, it will forward the request to a neighbor which holds file keys closest to this file key. This forwarding continues until the file is found or HTL (hops-to-live) becomes one (forwarding continues with some probability to prevent attackers to get info from HTL). This mechanism can be used to insert/search a file assuming you can obtain/calculate the file key. Nodes enroute can also cache the information about where the file is stored. LRU cache policy is used. Requests have an associated depth field for the replying node to set the HTL of the reply accordingly. Depth field is initialized to some small random value to obscure location of the requester. Performance numbers show that the system is reasonably scalable and fault-tolerant.

Other unstructured p2p systems: KaZaa [webd], Routing Indices [CGM02] are some other examples of *unstructured* indices. In Routing Indices, a node forwards a query to a subset of its neighbors, based on its local RI, rather than by selecting neighbors at random or by flooding the network by forwarding the query to all neighbors.

Odissea [SMW+03]: Odissea considers the problem building a P2P-based search engine. In particular, it addresses the problem of finding a ranked list of documents with the given keywords in a p2p setting. Odissea uses the Fagin's Algorithm(FA) and Threshold Algorithm(TA) for pruning the search space when intersecting the document lists containing the relevant keywords. Both of the algorithms compute the top-k results. For m terms and n documents FA requires $O(n^{\frac{m-1}{m}} k^{\frac{1}{m}})$ documents to be looked at. In [SMW+03], first the documents for each keyword are sorted according to their ranks and then a distributed version of the FA algorithm is applied. The ranking may be according to pageranks or tf-idf. For a two keyword query and the servers A and B responsible for them respectively, the algorithm to determine top-k documents is as follows. Server A sends x top documents from its list. Server B computes the intersection. If r_k is the minimum ranking of the intersection and r_{min} is the minimum ranking sent by server A then, B sends documents that have a ranking greater than $r_k - r_{min}$ back to A. Then A determines the overall top-k documents. The authors claim that x may be determined by experiments. This algorithm can be extended to more than two keyword queries.

PIER [HHB+03]: PIER is a distributed query engine based on DHTs. PIER provides support for some types of complex queries like equi-joins. However, they do not support efficient processing of range queries.

2.3 Correctness in P2P Systems

The Price of Validity in Dynamic Networks [BGGM04]: This paper specifies a correctness condition (single-site validity) for best-effort algorithms used to efficiently process aggregate queries. The authors present three correctness conditions: (1) Snapshot Validity: Result is aggregate query computed over hosts at a certain time t. (2) Interval Validity: Result is aggregate query computed over host in H where $H_I \leq H \leq H_U$. H_I is the set of hosts that are live all through the execution of the query and H_U is the set of hosts that are live at some point in the query. (3) Single-site Validity: Result is aggregate query computed over host in H where $H_C \leq H \leq H_U$. H_C contains all h such that there is at least one stable path in G from h_q (originator node) to h and H_U is the set of hosts which are live at some point in the query. Snapshot Validity and Interval Validity are hard to satisfy. There is an algorithm (in relaxed asynchronous model with reliable ordered communication) that achieves single-site validity. Techniques proposed in this paper cannot be used for defining and/or guaranteeing correctness in p2p range indices. Availability issues are not considered in this paper.

On the Correctness of Query Results in XML P2P Databases [Sar04]: This paper provides a definition of correctness of Query Results in XML P2P Databases. A query result is considered correct if it is a subset of the result obtained on executing the query on an equivalent central database. Query is sent to query plan generation layer and query plans are sent back to peers for execution. Author make the following assumptions: (1) Tree local completeness (tree is fully contained within the same location) (2) Query plan generation layer generates no false positives. It is possible that locations with no data relevant to the query are returned. The following is a simple example motivating the problem: Query Q returns all the authors who have published less than 5 papers in the last year. If $\{l_a, l_b, l_c, l_d\}$ is the set of peers with the relevant data. If l_a, l_b contain 3, 4 papers of John Doe respectively and if the returned set of peers is $\{l_b, l_c, l_d\}$ then the result will incorrectly include John Doe. Main results of the paper are: (Result

1) Correctness follows if query Q is monotone. (Result 2) Correct if query Q does not contain incorrect nested queries and it does not contain set predicates or aggregation functions applied to variables bound to incomplete sets. (Result 3 - Syntactically checkable queries): Correct if query Q does not contain incorrect nested queries and it does not contain set predicates or aggregation functions. According to the correctness definition provided in this paper, an empty result for a range query is always considered correct.

Scope: Scalable consistency maintenance in structured p2p systems [CRWZ05]: In this paper, the authors propose an effective way to maintain consistency between frequently updated files and their replicas. The naive approaches to replication suffer from hot-spot and node-failure problems. Hot spot problem may arise because the number of replicas per object varies significantly due to different object popularities, making popular nodes heavily loaded. If the node storing the replica information fails then update notifications have to propagated by broadcasting. The authors propose using a hierarchical structure called RPT (replica-partition-tree) to decentralize the replication location information and handle node churn efficiently. The basic idea is to split the whole identifier space into partitions and select one representative node in each partition to record the replica locations within that partition. Each partition may be further divided into smaller partitions in which child nodes are selected as representatives to take charge of the smaller partitions. Techniques proposed in this paper do not deal with correctness of query results and do not guarantee availability of items.

2.4 Load Balancing

2.4.1 Homogeneous

In this section, we will look at peer-to-peer load balancing schemes which work under the uniform node capacity and uniform query workload assumptions.

Consistent hashing [KLL+97]: This paper has two contributions: (1) random cache trees to decrease or eliminate hotspots, and (2) consistent hashing. Consistent hashing enables development of caching protocols which do not require users to have a current or even consistent view of the network. In loose terms, consistent hash function has the additional property that it changes minimally when the range of the function changes. Given buckets B and items I, a consistent hash function specifies an assignment of items to buckets for every possible view V (that is a subset of the buckets). The hash function should have some good properties like balance (roughly same number of items should be mapped to each bucket in a given view), low load (for a given bucket, the number of items that at least one person thinks belongs to the bucket is small). The basic idea is to choose two random functions rb and ri. rb maps buckets randomly to the unit interval and ri maps items randomly to the unit interval. Item i is then mapped to the closest bucket to i. To ensure load balancing, each bucket is replicated Theta(log C) times and rb maps each replica randomly, where C is the number of caches.

In the peer-to-peer parlance, when peer capacities and items popularities are uniform, the consistent hashing scheme proposes using log(N) virtual servers to ensure load balance. This algorithm works well for the case when peer capacities are uniform and popularities of items is uniform. However, increasing number of virtual servers per peer has the following shortcomings: Firstly, it increases churn.

This is because when one node departs it takes log(N) virtual servers with it. Secondly, each node must hold log(N) times more routing state. Storage is not an issue but maintenance cost now increases log(N) times. Finally, more virtual servers in the system causes number of hops per lookup (and hence latency) to increase.

Improvements to the consistent hashing scheme:

Simple efficient load balancing algorithms for peer-to-peer systems [KR04]: In this paper, the authors propose an extension of the consistent hashing scheme wherein load balance is ensured without the need for O(log N) virtual servers. This eliminates the problems with using virtual servers, the most important being increased network bandwidth. Each peer has log(N) ids, but only one of them is active at any given time. Nodes activate and deactivate virtual servers to balance the distance between themselves and their successor. As each node is allocated a fixed number of ids, this scheme has good security properties.

Balanced binary trees for ID management and load balance in distributed hash tables [Man04]: The algorithm proposed in this paper provides good load balancing with low arrival and departure costs. The load ratio of the most loaded peer to the least loaded peer is at most 4 whp.

Novel architectures for p2p apps: the continuous-discrete approach [NW03] and *A stochastic process on the hypercube with applications to peer-to-peer networks* [AHKV03]: Both these papers include low cost algorithms for load balancing, when peer capacities and items popularities are uniform. These algorithms depend on the history of node IDs that each node has used and their analysis are given only for insertions.

Simple load balancing for DHTs [BCM03]: In this paper, the authors propose using k hash functions (instead of one) for achieving load balance. The idea is to balance load by placing object at one or some of the least loaded k target peers. Other target peers point to a target peer with the object. To find an object, one can use any of the k functions to locate the target peer with the object. The scheme proposed in this paper is not a dynamic scheme - there is a fixed k for all objects, irrespective of their popularities.

2.4.2 Heterogeneous

In this section, we will present load balancing schemes that consider relaxing the uniform node capacity assumption. Some of these schemes do consider skewed workloads, item churn and peer churn.

Wide-area cooperative storage with CFS [DKK+01]: This paper considers the load balancing problem when peer capacities are heterogeneous. When a node joins the system, it is initialized with number of virtual servers proportional to its capacity. In addition, history of workload can also be taken into account to decide the initial number of virtual servers. Each node periodically checks its load and adds or sheds load without any communication with other nodes. If the node is overloaded and is using more than one virtual server, it just deletes the least loaded virtual server that will bring its load within bounds. If the node is underloaded, it just adds a new virtual server.

The above load balancing scheme has its drawbacks. First, a node with only a few VSs may not be able to form a good estimate of what the cost of creating a new one will be. Second, a meager machine still might be overloaded even if it is only running one VS. If a new physical server enters and has significantly less capacity than the current low-end servers, the system may take a long time to adjust to this new lowest common denominator. Third, if an overloaded node deletes one of its VSs, this may overload its neighbor, resulting in cascades of deletes. Finally, when the system is underloaded, above scheme can cause all nodes to create their maximum number of VSs, greatly increasing state, routing hops, and churn.

Load balancing in structured p2p systems [RLSK03]: The main idea of the paper is to move virtual peers for load balancing. Load on a virtual server is bounded by a predefined threshold and each node is responsible for enforcing this threshold by splitting the virtual servers as needed. Three simple load-balancing techniques are proposed based on amount of information used to decide how to rearrange load. Simplest is a one-to-one scheme where two nodes are picked at random and virtual server transfer is initiated if one node is heavy and the other node is light. Next scheme is a one-to-many scheme where a heavy node considers more than one light node at a time. The last scheme is a many-to-many scheme where many heavy nodes are matched to many light nodes. This scheme takes into account heterogeneity of peers, but assumes uniform object arrival pattern. It also doesn't consider peer churn.

Load balancing in dynamic p2p systems [GLS$^+$04a]: This paper is an extension of the previous paper. This scheme works fine even when objects consecutive on the id space are inserted/deleted (skewed object arrival pattern) and in the presence of churn. However, this approach does not work when there are some objects which are highly popular. For example, there is one object which is accessed heavily.

Compact, adaptive placement schemes for non-uniform requirements [BSS02]: In this paper, the authors consider the problem of designing compact, adaptive strategies for distribution of objects among a heterogeneous set of servers according to the server capabilities. Such a strategy should allow low space and time complexity computation of the position of an object. It also needs to adapt to servers joining and leaving the system and changing server capacities, so that objects are always distributed among servers according to their capabilities. The authors argue that standard hashing techniques can be used to for such an object distribution but they do not usually adapt well to change in server capabilities. In this paper the authors propose two strategies based on hashing that achieves all the above mentioned goals. One strategy is SHARE: The basic idea is to reduce the non-uniform placement problem to the uniform placement problem and use previously proposed strategies (ex: consistent hashing) which work well for the uniform case. Note that in the uniform case, capacities can only change when new servers enter the system or existing servers leave the system. The main shortcoming of this work is that it does not handle non-uniform query workload.

Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn [LS05]: In this paper, the authors argue that deployed systems have skewed workload distributions (heavy-tailed query distribution), high churn (high rate of node joins and leaves), heterogeneous capacities (wide variation in node bandwidth and storage capacities) and require reasonable security properties. None of the previous proposals for load balancing work under the above conditions. The authors propose k-choices, a load balancing algorithm for structured p2p system that supports skewed workloads, heterogeneous node capacities, and high churn, while retaining the security and application advantages afforded by verifiable ids. At a high level, the algorithm works as follows: (a) each node generates a set of verifiable IDs based on a single unit of certified information, (b) at join time, a node greedily reduces discrepancies between capacity and load both for itself and for nodes that will be affected by its join, and (c) optionally, each node experiencing overload or underload may periodically probe the network and reposition itself to another element from its set of verifiable IDs. Minimizing discrepancies between load and capacity achieves load balance, and limiting IDs to a well-defined set keeps the algorithm secure and resistant against Sybil attacks.

2.4.3 Miscellaneous

In this section, we preset some of the papers that deal with aspects similar to the load balancing question we are interested in.

Beehive [RS04b]: This paper addresses the question: How to place objects on peers and how to route to a copy of the object to achieve constant lookup performance for zipf-like query distributions? The authors propose to replicate objects based on their popularities such that an object can be retrieved in $O(1)$ hops on average. However, such a placement of objects does not necessarily lead to load balancing. In particular, a peer could be serving multiple popular objects and hence could be heavily loaded.

Uncoordinated load balancing and congestion games in p2p systems [STZ04]: We are given peers and the set of objects stored at these peers. We are also given users who are interested in some subset of objects. The problem is to map users to peers providing the objects they are interested in such that the total latency is minimized. Note that latency of a request served by a peer depends on the load on that peer. Main problem here is that the question of where to place the objects such that load is balanced is not addressed.

Adaptive peer selection [BFLZ03]: In file sharing systems, the objective is to find a peer from which you can quickly download a given file f. This paper tries to pick the best peer (using machine learning strategies) from a set of possible peers that serve the file f based on the past behavior of peers (download speeds, average uptime etc). Like in the previous paper, the question of where to place the objects such that load is balanced is not addressed.

2.5 Applications

2.5.1 File-Sharing and Content Distribution

The killer-app Napster [webe] started it all. Napster was the first to implement a peer-to-peer file sharing system for sharing music files. Napster had 50 million users in about 15 months from its inception, making it the web-application with the fastest growth recorded so far. Lookups in Napster were resolved at a central node and hence Napster was not fully decentralized.

Many other peer-to-peer file sharing systems like Gnutella [webc], Freenet [ICH00], and KaZaa [webd] followed suite. Currently, BitTorrent [weba] and eDonkey/Overnet [webf] are among the most popular file sharing systems used today. BitTorrent uses p2p file download with different pieces of the file downloaded in parallel from different users. Until recently, a central tracker web site (like Suprnova [webg]) was used to resolve lookups in BitTorrent. Legal issues caused the shut down of these websites. Now, p2p tracking mechanisms (like Exceem [webb]) are available for BitTorrent. Overnet/eDonkey is the only popular structured p2p file-sharing system in use today.

2.5.2 P2p Name Services

Overlook [TJ02]: This paper presents a scalable fault-tolerant name service that utilizes a P2P overlay network (like Pastry). Here, we have a set of Overlook servers connected by Pastry overlay network. The Overlook servers register themselves with the DNS. Clients find an Overlook server using DNS and then send the requests to the selected server. Servers act as proxies for client requests, forwarding the requests on the Pastry network and relaying the replies back to the client. Experimental results show that Overlook scales well.

Serving DNS using a P2P Lookup Service [CMM02]: DNSSEC (DNS with security extensions) allows for verification of records obtained by alternate means. This enables exploring alternative storage mechanisms for DNS records. Once such mechanism using DHash (a distributed hash-table built on top of Chord) is explored in this paper. An important shortcoming of DNS is the fact that it requires significant expertise to administer. Serving DNS over Chord using DDNS solves this by separating service from authority. This also provides better load balancing and robustness against DOS attacks.

Cooperative Domain Name System [RS04a]: This paper presents a high-performance, failure-resilient, and scalable name service for the Internet. It could be used as an alternative or as a safety-net for existing DNS. CoDons is built on top of Beehive [RS04b], a replication framework that provides $O(1)$ lookup latency on average. CoDoNS has the following good properties: (1) DoS-resilient: Servers that are targeted by denial-of-service attacks automatically shed their load to other nodes in the system, (2) High performance: CoDoNS works on Beehive and hence provides low latency query responses, and (3) Fast updates: Unlike in legacy DNS, updates in CoDoNs can be performed at any time and take relatively short time to propagate.

2.5.3 Peer-to-peer Caching

Peer-to-Peer caching or cooperative caching proposes elimination of proxy servers completely and instead stores meta-information at the individual clients (or peers). Padmanabhan et al [PS02] examine a server redirection scheme that uses IP prefixes, network bandwidth estimates, and landmarks to redirect a client request at the web server to a nearby client. Peer-to-peer web caching schemes such as COOPnet, BuddyWeb, Backslash and Squirrel organize network clients in an overlay within which object requests are routed. Stading et al [DLN02] propose institutional level special DNS and HTTP servers, called "Backslash" nodes. Backslash nodes are organized within the Content Addressable Network (CAN) overlay, and an external web cache request is routed from a client to the nearest Backslash node, and then into the CAN overlay itself. BuddyWeb [WNO$^+$02] uses a custom p2p overlay among the clients themselves to route object requests. Squirrel [IRD02] builds a cooperative web cache on top of the Pastry p2p routing substrate.

2.5.4 Resource Discovery

Resource discovery is an important application in a grid or cluster setting. P2P indices can be used to efficiently answer questions like: find a node running Win XP with memory $>= 100MB$ and CPU $>= 2GHz$. Such a query can be used by VMWare to find a suitable machine to host a given virtual machine.

2.5.5 Storage Services

Amazon S3 uses DHT based storage system to provide a highly scalable, highly availability simple storage service. Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web

2.5.6 Parallel Computing

Peer-to-peer networks have also begun to attract attention from scientists in other disciplines, especially those that deal with large datasets such as bioinformatics. P2P networks can be used to run large programs designed to carry out tests to identify drug candidates. The first such program was begun in 2001 the Centre for Computational Drug Discovery at Oxford University in cooperation with the National Foundation for Cancer Research. There are now several similar programs running under the auspices of the United Devices Cancer Research Project.

2.5.7 Routing Infrastructures

There are several routing infrastructures mentioned in the literature that use DHTs or are inspired by DHTs. For example, IPv6 routing can be implemented as a self-organizing overlay network on top of the current IPv4 infrastructure (see [ZvRM02]), Virtual Ring Routing (VRR) is a point-to-point network routing protocol inspired by DHTs that can be implemented on any link layer (see [CCN+06]).

2.5.8 Miscellaneous

P2P-based digital libraries, Anonymity using p2p networks, using DHTs to build tools such as a *Distributed lock manager* are some other very interesting applications of p2p systems.

Chapter 3

Kelips:An Efficient and Stable P2P DHT

3.1 Introduction

A peer-to-peer (p2p) distributed hash table (DHT) implements operations allowing hosts or processes (nodes) to join the system, and fail silently (or leave the system), as well as to insert and retrieve files with known names. Several DHTs are in deployment, e.g. Gnutella and Kazaa, while many others are a focus of academic research, e.g., Chord [SMK+01], Pastry [RD01b], Tapestry [ZKJ01], etc. [IPT02].

All p2p systems make tradeoffs between the amount of storage overhead at each node, the communication costs incurred while running, and the costs of file retrieval. With the exception of Gnutella, the work just cited has focused on a design point in which storage costs are logarithmic in system size and hence small, and lookup costs are also logarithmic (unless cache hits shortcut the search). But there are other potentially interesting design points.

One could vary the soft state memory usage and background network communication overhead at a node in order to realize $O(1)$ lookup costs. For example, complete replication of soft state achieves this, but this approach has prohibitive memory and bandwidth requirements.

The Kelips[1] system uses $O(\sqrt{n})$ space per node, where n is the number of nodes in the system. This soft state suffices to resolve lookups with $O(1)$ time and message complexity. The constant cost continuous background overhead is

[1]System name derived from *kelip-kelip*, Malay name for the self-synchronizing fireflies that accumulate after dusk on branches of mangrove trees in Selangor, Malaysia [URLc]. Our system similarly organizes into affinity groups, and nodes in a group "synchronize" to store information for the same set of file indices.

used to maintain the index structure with high quality, as well as guarantee quick convergence after membership changes. In contrast, many classical p2p designs suffer because large numbers of nodes are found to be inaccessible when an access is attempted. The $\sqrt{n}$ design point is of interest because, within Kelips, both the storage overhead associated with the membership data structure and that associated with replication of file-index (henceforth called *filetuple*) data impose the same $O(\sqrt{n})$ asymptotic cost. Kelips uses query rerouting to ensure lookup success in spite of failures. The mechanism also allows us to use an idea from the widely cited "small worlds" algorithms when selecting peers for each node.

Memory usage is small for systems with moderate sizes - if 10 million files are inserted into a 100,000-node system, Kelips uses only 1.93 MB of memory at each node. The system exhibits stability in the face of node failures and packet losses, and hence would be expected to ride out "churn" arising in wide-area settings as well as rapid arrival and failure of nodes. This resilience arises from the use of a lightweight Epidemic multicast protocol for replication of system membership data and file indexing data [Bai75, DGH+87]. We note that whereas many DHT systems treat file replication as well as lookup, our work focuses only on the lookup problem, leaving replication to the application. For reasons of brevity, this chapter also omits any discussion of privacy and security considerations.

3.2 Core Design

Kelips consists of k virtual *affinity groups*, numbered 0 through $(k-1)$. Each node lies in an affinity group determined by using a consistent hashing function to map the node's identifier (IP address and port number) into the integer interval $[0, k-1]$. Let n be the number of nodes currently in the system. The use of a cryptographic hash function such as SHA-1 ensures that with high probability, each affinity group contains close to $\frac{n}{k}$ nodes.

Node soft state consists of the following entries:

• Affinity Group View: A (partial) set of other nodes lying in the same affinity group. Each entry carries additional fields such as round-trip time estimate, heartbeat count, etc. for the other node.

• Contacts: For each of the other affinity groups in the system, a small (constant-sized) set of nodes lying in the foreign affinity group. Entries contain the same additional fields as in the affinity group view.

• Filetuples: A (partial) set of tuples, each detailing a file name and host IP address of the node storing the file (called the file's *homenode*). A node stores a filetuple only if the file's homenode lies in this node's affinity group. Filetuples are also associated with heartbeat counts.

Figure 3.1 illustrates an example. Entries are stored in AVL trees to support efficient operations.

Memory Usage at a node The total storage requirements for a Kelips node are $S(k,n) = \frac{n}{k} + c \times (k-1) + \frac{F}{k}$ entries (c is the number of contacts per foreign affinity group and F the total number of files present in the system). For fixed n, $S(k,n)$ is minimized at $k = \sqrt{\frac{n+F}{c}}$. Assuming the total number of files is proportional to n, and that c is fixed, k then varies as $O(\sqrt{n})$. The minimum $S(k,n)$ varies as $O(\sqrt{n})$. This is larger than Chord or Pastry, but reasonable for most medium-sized p2p systems.

Consider a medium-sized system of $n = 100,000$ nodes over $k = \lceil \sqrt{n} \rceil = 317$ affinity groups. Our current implementation uses 60 B filetuple entries and 40 B membership entries, and maintains 2 contacts per foreign affinity group. Inserting

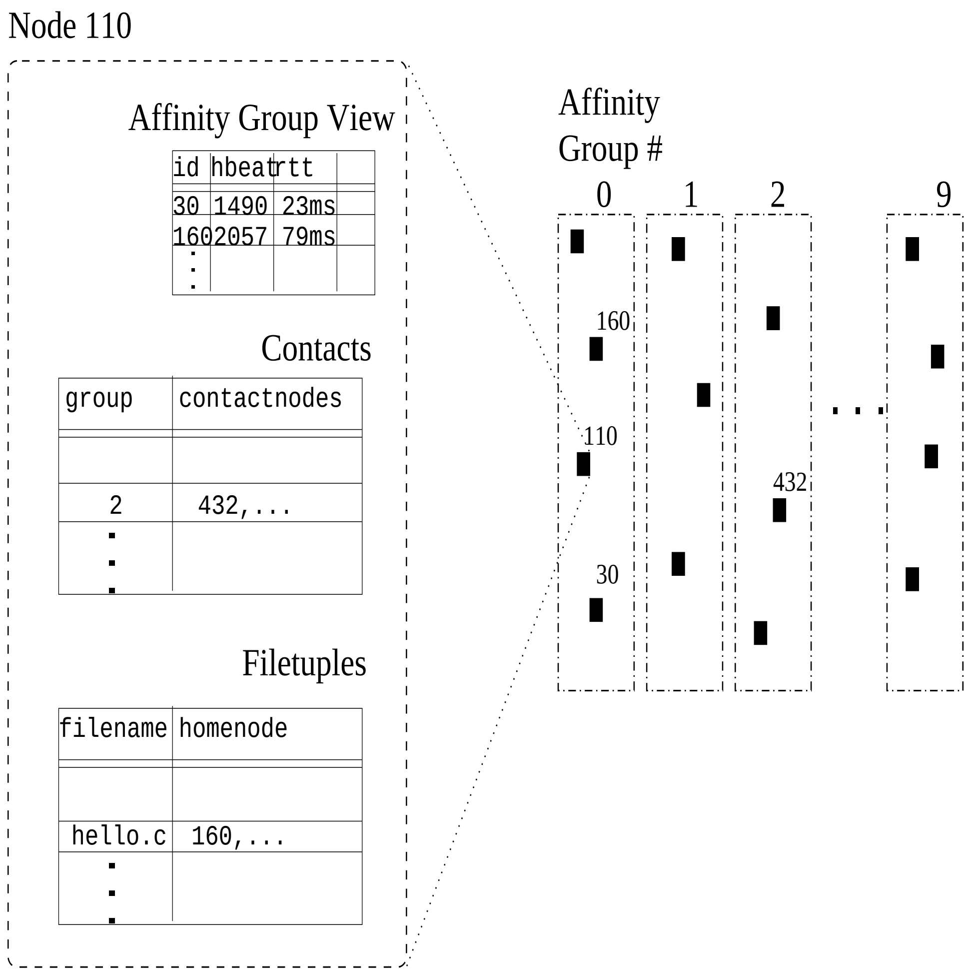


Figure 3.1: **Soft State at a Node:** *A Kelips system with nodes distributed across 10 affinity groups, and soft state at a hypothetical node.*

a total of 10 million files into the system thus entails 1.93 MB of node soft state. With such memory requirements, file lookup queries return the location of the file within $O(1)$ time and message complexity (i.e., these costs are invariant with system size n).

3.2.1 Background Overhead

Given a system of n nodes across k affinity groups, view, contact and filetuple entries are refreshed periodically within and across groups. This occurs through a heartbeating mechanism. Each view, contact or filetuple entry stored at a node is associated with an integer heartbeat count. If the heartbeat count for an entry is not updated over a pre-specified time-out period, the entry is deleted. Heartbeat updates originate at the responsible node (for filetuples, this is the homenode) and are disseminated through a peer-to-peer Epidemic protocol [vRMH98].

We briefly describe epidemic-style dissemination within an affinity group. Then we generalize to multiple affinity groups.

An epidemic (or gossip-based) protocol disseminates a piece of information (e.g., a heartbeat update for a filetuple) in the following manner. Once a node receives the piece of information to be multicast (either from some other node or from the application), the node gossips about this information for a number of *rounds*, where a round is a fixed local time interval at the node. During each round, the node selects a small constant-sized set of target nodes from the group membership, and sends each of these nodes a copy of the information. With high probability, the protocol transmits the multicast to all nodes. The latency varies with the logarithm of affinity group size. Gossip messages are transmitted via a lightweight unreliable protocol such as UDP. Gossip target nodes are selected through a weighted scheme based on round-trip time estimates, preferring nodes that are topologically closer in the network. Kelips uses the spatially weighted gossip proposed in [KKD01] towards this. A node with round-trip time estimate rtt is selected as gossip target with probability proportional to $\frac{1}{rtt^r}$. As suggested in [KKD01], we use a value of $r = 2$, where the latency is polylogarithmic ($O(log^2(n))$.

Analysis and experimental studies have revealed that epidemic style dissemination protocols are robust to network packet losses, as well as to transient and permanent node failures. They maintain stable multicast throughput to the affinity group even in the presence of such failures. See references [Bai75, BHO+99, DGH+87].

Information such as heartbeats also need to propagate across affinity groups (e.g., to keep contact entries for this affinity group from expiring). This is achieved by selecting a few of the contacts as gossip targets in each gossip round. Such cross-group dissemination implies a two-level gossiping scheme [vRMH98]. With a uniform selection of cross-group gossip targets, latency is more than that of single group gossip by a multiplicative factor of $O(log(k))$ (same as $O(log(n))$).

Gossip messages in Kelips carry not just a single entry, but several filetuple and membership entries. This includes entries that are new, were recently deleted, or with an updated heartbeat. Since Kelips limits bandwidth use at each node, not all the soft state can be packed into a gossip message. Maximum rations are imposed on each of the number of view entries, contact entries and filetuple entries that a gossip message may contain. For each entry type, the ration subdivides equally for fresh entries (ones that have so far been included in fewer than a threshold number of gossip messages sent out from this node) and for older entries. Entries are chosen uniformly at random, and unused rations (e.g., from few fresh entries) are filled with older entries.

Ration sizes do not vary with n. With $k = \sqrt{n}$, this increases dissemination latencies a factor of $O(\sqrt{n})$ above that of the Epidemic protocol (since soft state is $O(\sqrt{n})$). Heartbeat timeouts thus need to vary as $O(\sqrt{n} \times log^2(n))$ for view and filetuple entries, and $O(\sqrt{n} \times log^3(n))$ for contact entries.

These numbers thus are the convergence times for the system after membership changes. Such low convergence times are achieved through only the gossip messages sent and received at a node (henceforth called the *gossip stream*). This imposes a constant per-node background overhead. The gossip stream keeps heartbeats flowing in spite of node and packet delivery failures, thus allowing lookups to succeed.

3.2.2 File Lookup and Insertion

Lookup: Consider a node (querying node) that desires to fetch a given file. The querying node maps the file name to the appropriate affinity group by using the same consistent hashing used to decide node affinity groups. It then sends a lookup request to the topologically closest contact among those known for that affinity group. A lookup request is resolved by searching among the filetuples maintained at the node, and returning to the querying node the address of the homenode storing the file. This scheme returns the homenode address to a querying node in $O(1)$ time and with $O(1)$ message complexity. The querying node fetches the file directly from the homenode.

Insertion: A node (*origin node*) that wants to insert a given file f, maps the file name to the appropriate affinity group, and sends an insert request to the topologically closest known contact for that affinity group. This contact picks a node h from its affinity group, uniformly at random, and forwards the insert request to it. The node h is now the *homenode* of the file. The file is transferred from the origin node to the homenode. A new filetuple is created listing the file f as being stored at the homenode h, and is inserted into the gossip stream. Thus, insertion also occurs in $O(1)$ time and with $O(1)$ message complexity. The origin node periodically refreshes the filetuple entry at homenode h in order to keep it from expiring.

Clearly, factors such as empty contact sets or incomplete filetuple replication might cause such one-hop lookup or insertion to fail. Biased partial membership information might cause uneven load balancing. This is addressed by the general multi-hop multi-try query routing scheme of Section 3.3.

3.3 Auxiliary Protocols and Algorithms

We outline Kelips' protocols for node arrival, membership and contact maintenance, topological considerations and multi-hop query routing.

Joining protocol: Like in several existing p2p systems, a node joins the Kelips system by contacting a well-known introducer node (or group), e.g., a well-known http URL could be used. The joiner view returned by the introducer is used by the new node to warm up its soft state and allow it to start gossiping and populating its view, contact and filetuple set. News about the new node spreads quickly through the system.

Spatial Considerations: Each node periodically pings a small set of other nodes it knows about. Response times are included in round-trip time estimates used in spatial gossip.

Contact maintenance: The maximum number of contacts is fixed, yet the gossip stream constantly supplies potential contacts. *Contact replacement* policy can affect lookup/insert performance and system partitionability, and could be either proactive or reactive. Currently, we use a proactive policy with the farthest

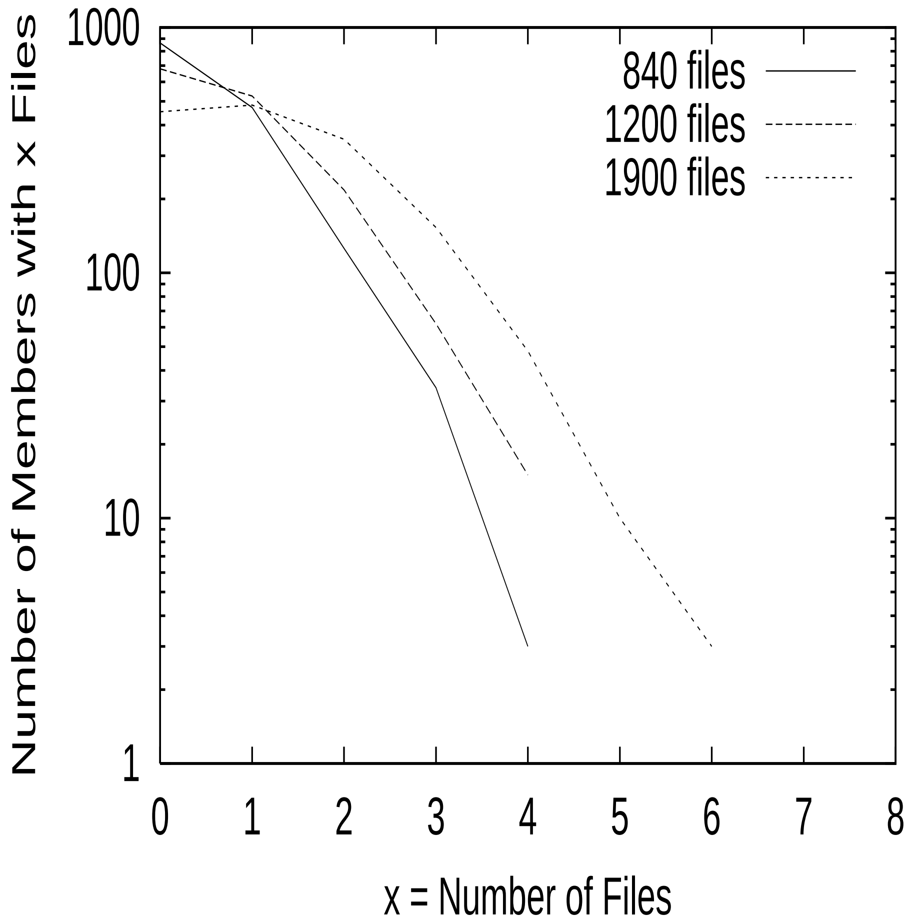


Figure 3.2: **Load Balancing I:** *Number of nodes (y-axis) storing given number of files (x-axis), in a Kelips system with 1500 nodes (38 affinity groups).*

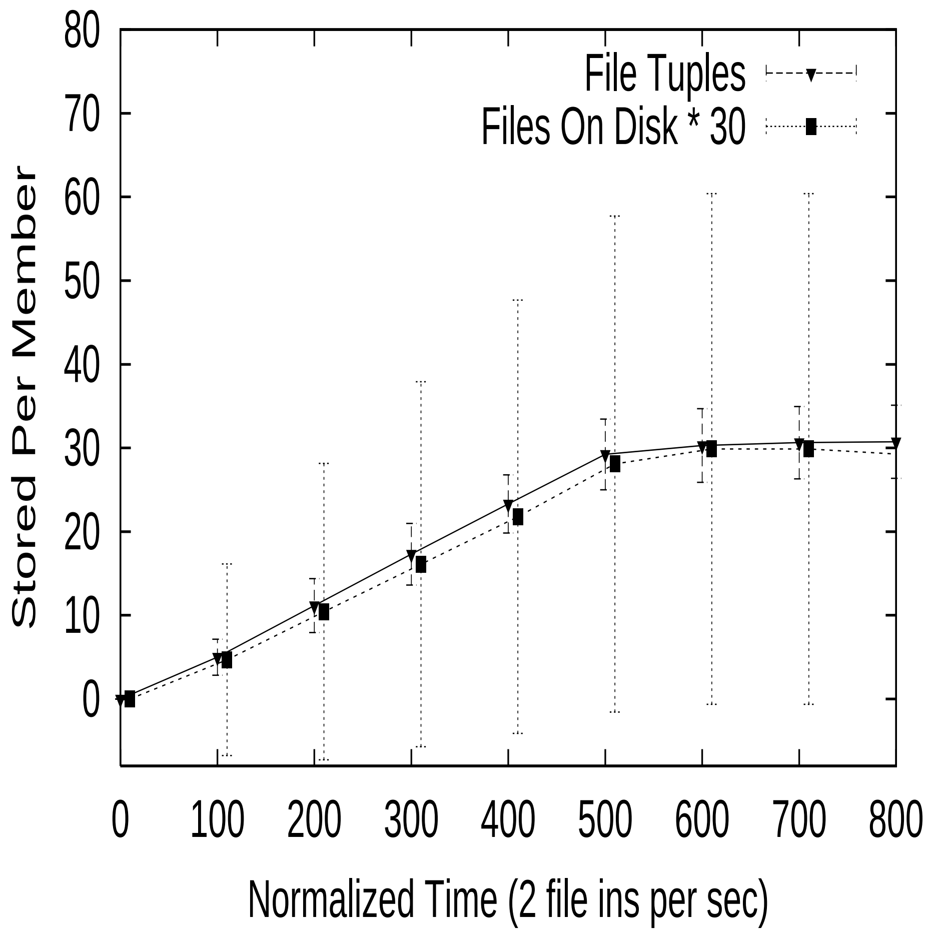


Figure 3.3: **Load Balancing II:** *Files are inserted into a 1000 node system (30 affinity groups), 2 insertions per sec between t=0 and t=500. Plot shows variation, over time, of number of files and filetuples at a node (average and one standard deviation).*

contact chosen as victim for replacement.

Multi-hop Query routing: When a file lookup or insert query fails, the querying node retries the query. Query (re-) tries may occur along several axes: a) the querying node could ask multiple contacts, b) contacts could be asked to forward the query within their affinity group (up to a specified TTL), c) the querying node could request the query to be executed at another node in its own affinity group (if this is different from the file's affinity group). Query routing occurs as a random walk within the file affinity group in (b), and within the querying node's affinity group in (c). TTL values on multi-hop routed queries and the maximum numbers of tries define a tradeoff between lookup query success rate and maximum processing time. The normal case lookup processing time and message complexity stay $O(1)$.

File insertion occurs through a similar multi-hop multi-try scheme, except the file is inserted exactly at the node where the TTL expires. This helps achieve good load balancing, although it increases the normal case insertion time to grow as $O(log(\sqrt{n}))$. However, this is competitive with existing systems.

3.4 Experimental Results

We are evaluating a C WinAPI prototype implementation of Kelips. This section reveals preliminary numbers from the system. Multiple nodes were run on a single host (1 GHz CPU, 1GB RAM, Win2K) with an emulated network topology layer. Unfortunately, limitations on resources and memory requirements have restricted currently simulated system sizes to thousands of nodes.

Background overhead in the current configuration consists of each node gossiping once every 2 (normalized) seconds. Rations limit gossip message size to 272 B. 6 gossip targets are chosen, 3 of them among contacts.

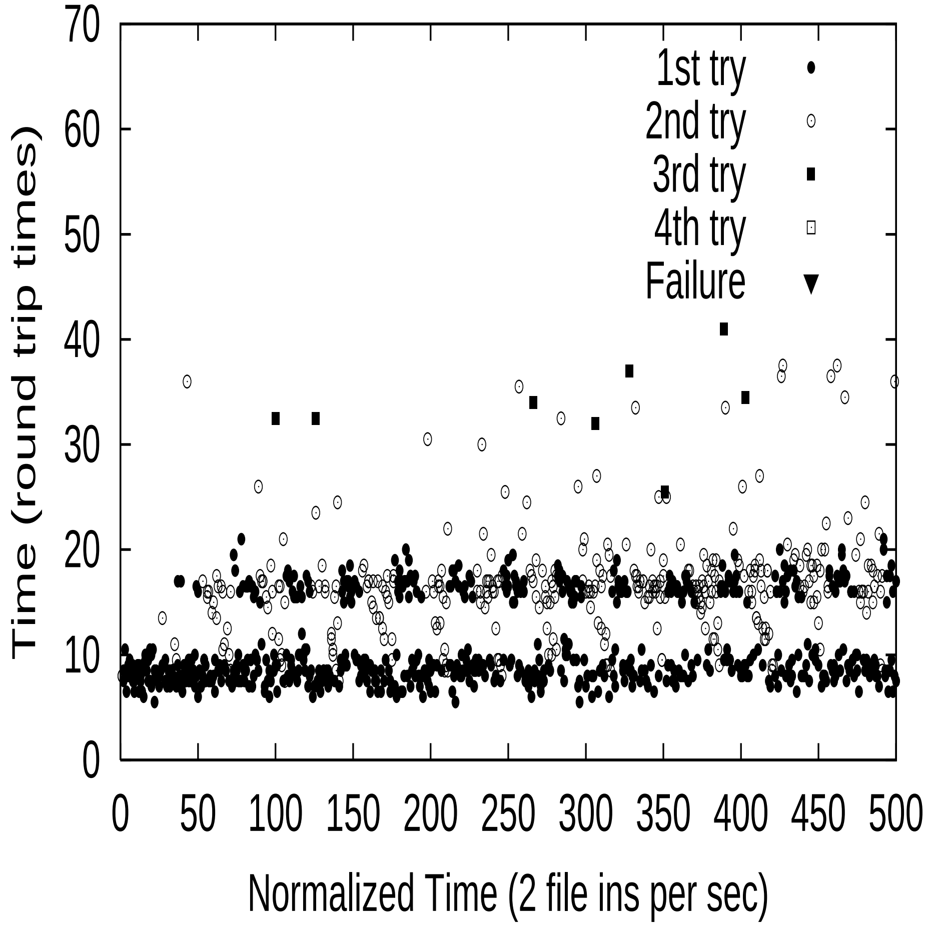


Figure 3.4: **File Insertion:** *Turnaround times (in round-trip time units) for file insertion in a 1000-node Kelips system (30 affinity groups).*

Load Balancing: Files are inserted into a stable Kelips system. The file name distribution used is a set of anonymized web URLs obtained from the Berkeley Home IP traces at [URLd]. The load balancing characteristics are better than exponential (Figure 3.2). File and filetuple distribution as files are inserted (2 insertions per normalized second of time) is shown in Figure 3.3; the plot shows that filetuple distribution has small deviation around the mean.

File Insertion: This occurs through a multi-try (4 tries) and multi-hop scheme (TTL set to $3 * logN$ virtual hops). Figure 3.4 shows the turnaround times for insertion of 1000 different files. 66.2% complete in 1 try, 33% take 2 tries, and 8% take 3 tries. None fail or require more than 3 tries. Views were found to be fully replicated in this instance. In a different experiment with 1500 nodes and views only 55.8% of the maximum size, 47.2% inserts required 1 try, 47.04% required 2 tries, 3.76% required 3 tries, 0.96% needed 4 tries, and 1.04% failed. Multi-hop routing thus provides fault-tolerance to incompleteness replication of soft state.

Fault-tolerance: Figures 3.5 and 3.6 show the fault-tolerance achieved through the use of background overhead (gossip stream). Lookups were initiated at a constant rate and were found to fail only if the homenode had also failed (Figure 3.5). In other words, multi-hop rerouting and redundant membership information ensures successful lookups despite failures. Responsiveness to failures is good, and membership and filetuple entry information stabilize quickly after a membership change (Figure 3.6).

3.5 Conclusions

We are investigating a new design point for DHT systems, based on increased memory usage (for replication of filetuple and membership information) and a constant and low background overhead at a node, in order to enable $O(1)$ file

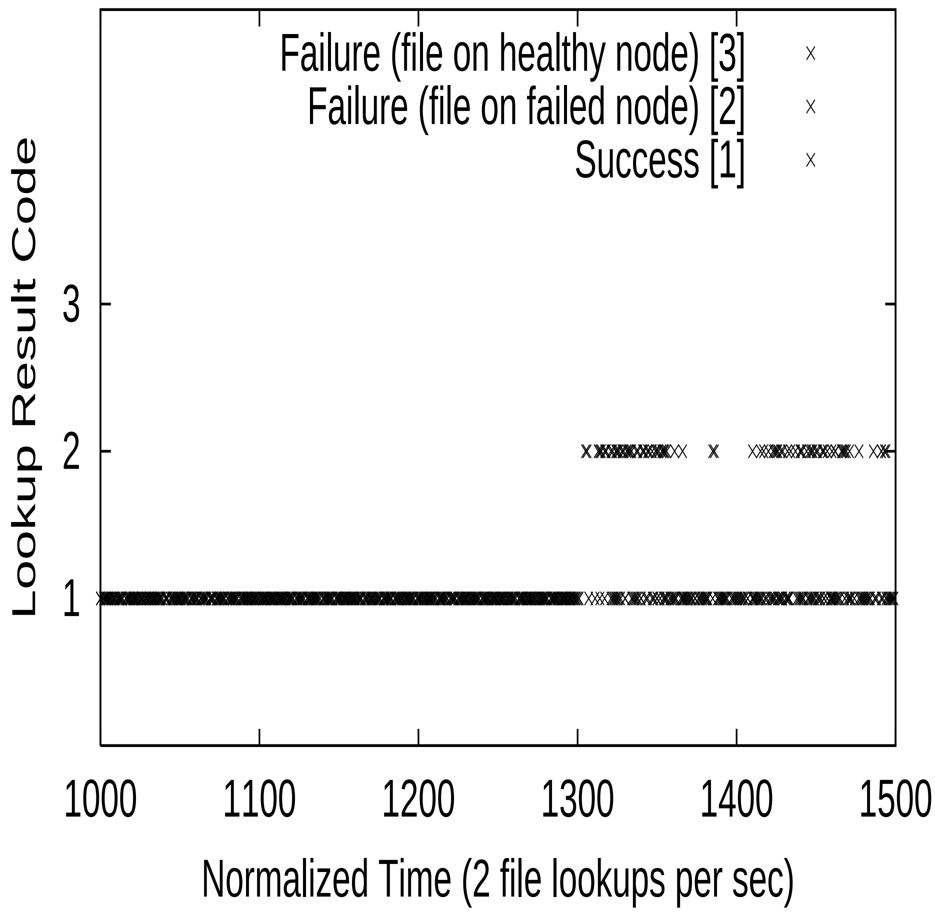


Figure 3.5: **Fault Tolerance of Lookups I:** *In a 1000 node (30 affinity groups) system, lookups are generated 2 per sec. At time* $t = 1300$*, 500 nodes are selected at random and caused to fail. This plot shows for each lookup if it was successful* $[y - axis = 1]$*, or if it failed because the homenode failed* $[2]$*, or if it failed in spite of the homenode being alive* $[3]$*.*

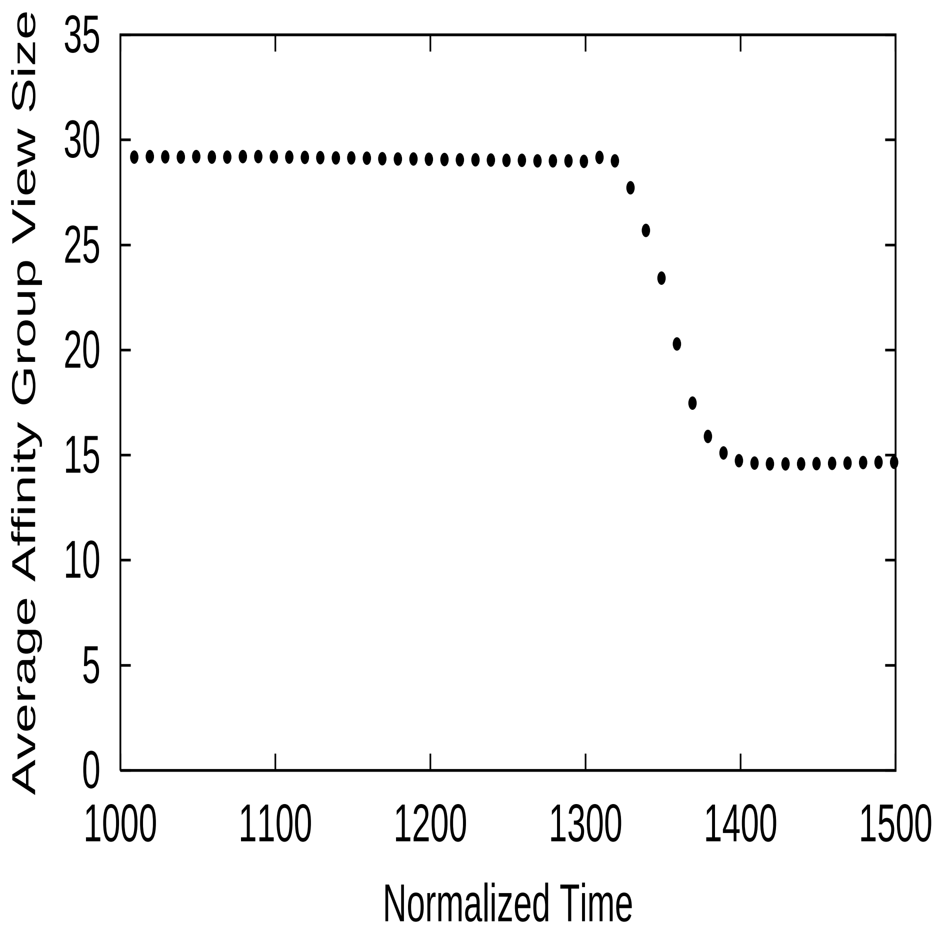


Figure 3.6: **Fault Tolerance of Lookups II:** *At time t=1300, 500 out of 1000 nodes in a 30 affinity group system fail. This plot shows that failure detection and view (and hence filetuple) stabilization occurs by time t=1380.*

lookup operations and stability despite high failure and churn rates. Per-node memory requirements are small in medium-sized systems (less than 2 MB with 10 million files in a 100,000 node system). Multi-hop (and multi-try) query routing enables file lookup and insertion to succeed even when bandwidth limitations or network disconnectivity lead to only partial replication of soft state. We observe satisfactory load balancing.

Chapter 4

Kache: A Churn Resistant Cooperative Caching Scheme

4.1 Introduction

Many systems today are built upon the assumption that if they are secure and available, they are robust. However, security violations such as denial of service attacks can be initiated by sources otherwise considered non-malicious. We consider one such source of possible disruptions in the service of a peer-to-peer system - churn and continuously occurring failures of nodes and communication. We investigate this within the context of a peer to peer system for cooperative caching of web services and objects, and show that probabilistic techniques can be used to build a design that heals proactively in the face of system instabilities. The system also adapts itself to the underlying network topology to provide access to nearby cached copies of web objects.

An *external caching* scheme stores copies of web objects or meta-data so that clients can avoid requests to a web server. The best known example of an external cache is a web proxy, but with the rapid spread of web technologies, many other caching scenarios can be identified. Caching is central to performance in the web, both because it reduces load on servers, and because cached data is normally cheaper to access from another machine that is closer in the network. Although cooperation between cache managers is not a common feature of existing web architectures, cooperative caching supported by peer-to-peer architectures has great potential [IRD02, PS02]. When sets of client systems have similar interests and fast low-latency interconnections, shared caching could bring significant benefits.

The peer-to-peer mechanisms referred to above emerged from interest in file sharing, and they are a technique with applicability to sharing web caches [IRD02, PS02, WNO+02]. We focus on cached web pages, primarily because detailed traces are available for this case and because such an analysis permits direct comparison with other systems.

However, cooperative caching is more generally applicable, and it could improve performance for web services and applications. Mohan studies a variety of these scenarios in [Moh02]. For example, dynamic web content serving can be accelerated by caching html fragments, web applications such as EJBs can be cached, and database operations such as queries can be speeded up by caching their results. Peer-to-peer indexing is a good match for such settings, and the longer term goal arising from this work is to develop a powerful caching solution useful in a diversity of web caching settings.

A primary concern about peer-to-peer cooperative caching is that while protocols in this class scale well, they are also sensitive to "churn", whereby hosts that join and leave the system trigger high overheads as the system restabilizes [PS02]. Churn-related overhead is more than just a nuisance, since an attacker seeking to disable a system could provoke churn to mount a distributed denial of service attack, potentially crippling the sharing mechanism while also subjecting participating machines to high loads. Resistance to churn is a crucial objective if this type of mechanism is to be successful.

The Kelips system [GBL+03], is unusual in employing probabilistic schemes and a self-regenerating data structure that adapts automatically and with bounded loads (independent of system size) as machines join, leave, or fail, or other disturbances occur. Here we show that when Kelips is employed for shared web caching, the system maintains rapid lookups and low overheads even when subjected to high churn rates, and even if new cache entries are simultaneously added.

Our analysis focuses on server-bandwidth, lookup time and access latency both when a cache hit occurs and when a miss is detected, load balancing and robustness to failures and churn. Our work is experimental, and includes (a) a microbenchmark study for small clusters, and (b) a trace-drive simulation study for larger-sized systems. Our evaluation uses web access traces from the Berkeley Home IP network [Dav], transit-stub network topology maps obtained through the Georgia-Tech generator [URLa], and churn traces from the Overnet deployment (obtained from the authors of [BSV03]). We show that loads are low and independent of the rate of churn and failure events, and the system adapts itself so that peers that tend to join and leave frequently are unlikely to be used as targets in lookups - in effect, requests are directed towards more reliable peers and, within this set, towards those with lower expected latency. Coupled with the extremely good scalability of the technique, we believe that Kelips is a strong candidate for caching in web systems of all kinds, including traditional web sites, databases, and web services.

4.2 Related Work

Normal HTTP Request Processing A client's request for a web object is first serviced from the local cache on the client's machine. This might fail because either the object is uncacheable, or not present in the cache, or the local copy is stale [1].

[1] Freshness is determined through the use of an expiration policy in the web cache. The expiration time is either specified by the origin server or is computed by the web cache based on the last modification time.

In the first case, the object's web server is contacted by issuing an HTTP GET application level request. In the second case (client cache miss), an HTTP GET is issued to the external web cache. For the third case (client cache copy stale), an HTTP conditional CGET is issued to the external web cache. If the external web cache is unable to service the GET (CGET), it could either fall-through to the web server or inform the client to contact the server directly. The reply to an external web cache request is the object or, in case of a CGET, a *not-modified* reply indicating that the stale copy is indeed the latest version of the object.

The design of an external web cache falls into one of the following three categories : (1) a hierarchy of proxies, (2) distributed proxies, or (3) peer-to-peer caches. We present a truncated survey below - the interested reader is referred to [Wan99] for a comprehensive study.

1. Hierarchical Schemes: Harvest [CDN$^+$96] and Squid [Wes] connect multiple web proxy servers at the institutional, wide area network and root levels in a virtual hierarchy. Servers store caches of objects, and an external web cache request is serviced through these multiple levels by traversing the parent, child and sibling pointers. Chankhunthod et al [CDN$^+$96] found that up to three levels of proxy servers could be maintained without a latency loss compared to that of direct web server access. Wang [Wan99] outlines some of the drawbacks of the hierarchy - proxy placement, redundant cache copies, and load on servers close to the root, etc.

2. Distributed Caching: Provey and Harrison [PH97] store only cache hints (not objects) at proxy servers. Cachemesh [WC97] partitions out the URL space among cache servers using hashing. A cache routing table among the servers is then used to route requests for objects.

3. Peer-to-peer Caching: The above schemes still require a proxy infrastructure. The elimination of proxy servers completely implies that the meta-information that would normally be stored inside the hierarchy must instead be stored at the individual clients or the server.

Padmanabhan et al [PS02] examine a server redirection scheme that uses IP prefixes, network bandwidth estimates, and landmarks to redirect a client request at the web server to a nearby client. Peer-to-peer web caching schemes such as COOPnet, BuddyWeb, Backslash and Squirrel organize network clients in an overlay within which object requests are routed. Stading et al [DLN02] propose institutional level special DNS and HTTP servers, called "Backslash" nodes. Backslash nodes are organized within the Content Addressable Network (CAN) overlay, and an external web cache request is routed from a client to the nearest Backslash node, and then into the CAN overlay itself. BuddyWeb [WNO$^+$02] uses a custom p2p overlay among the clients themselves to route object requests. Squirrel [IRD02] builds a cooperative web cache on top of the Pastry p2p routing substrate.

Padmanabhan et al contended in [PS02] that the use of peer-to-peer routing substrates for web caching may be too "heavy-weight because individual clients may not participate in the peer to peer network for very long, necessitating constant updates of the distributed data structures". Work on the p2p cooperative web cache designs described above has not addressed this criticism. Although the above p2p overlays are self-reorganizing, we believe our work is the first systematic study of cooperative caching under the form of "churn attack" discussed earlier.

There is a preliminary theoretical study by the authors of article [LNBK02] on how the Chord peer-to-peer system uses a periodic stabilization protocol to combat the effect of concurrent node arrival and failure. Their theoretical analysis however revealed that such a protocol would be infeasible to run - either the time to stabilization or the bandwidth consumed grow super-linearly with the number of nodes. The Kelips web caching solution does not require supplementary stabilization protocols; constant-cost and low-bandwidth background communication suffices to combat significant rates of churn while ensuring favorable and robust performance numbers. Our study in the current chapter is also the first to demonstrate a practicable and efficient solution to the problem of churn and experimentally study its working under realistic conditions.

4.3 The Kelips Peer-to-peer Overlay

Peer-to-peer overlays for object insertion and retrieval such as Pastry, Tapestry, and Chord define how each participant node chooses peers to which they maintain pointers. In contrast, Kelips uses softer rules to determine sets of possible peer pointers, permitting a node to pick any peer within the set according to end-user considerations and constraints such as topology awareness, trust, security concerns, etc. The choice can vary over time, and this gives Kelips the "self-regenerating" behavior mentioned earlier.

More precisely, a Kelips system with n nodes consists of $\sqrt{n}$ virtual subgroups called *affinity groups*, numbered 0 through $(\sqrt{n} - 1)$. Each node lies in an affinity group, determined by using a consistent hashing function to map the node's identifier (IP address and port number) into the integer interval $[0, \sqrt{n} - 1]$. Using SHA-1 as the hash function, each affinity group size will lie in an interval around $\sqrt{n}$ w.h.p. The value of n should be consistently known at all nodes, but can be an estimate of the actual system size. Please refer to chapter 3 for further details of Kelips.

Kelips Flexibility: While designing a Kelips-based p2p application (such as web caching), the designer as well as end nodes are equipped with a flexible choice of policies and tuning knobs.

• **Background Overhead** can be increased to lower dissemination latency.

• **Peer Maintenance** can be done through flexible end-to-end policies, e.g., based on network proximity, preference for peers not connected through a firewall, trusted peers, etc.

• **Multiple tries and Routing of queries** enables it to reach an appropriate node (i.e., one with a copy of the resource tuple) in the resource's affinity group when the 1 RPC lookup fails. TTL (time-to-live) and the number of retries can be used to trade latency with likelihood of success.

• **Replication Policies** for the resource (not the resource tuple) can be chosen orthogonal to the base operation of Kelips.

For Kelips web caching, the policy choices used are described in Section 4.4, and Section 4.5 gives experimental results to show the effect of cranking the knobs.

4.4 Design of a P2P Web Caching Application with Kelips

We study the design of a decentralized web caching application with Kelips. There are two options to designing an application over a p2p DHT (such as Pastry or Kelips) : (a) layering, through the use of the standard `get(object, ...)`, `put(object, ...)` API exported by the DHT layer (as in [ZDKS03]), and (b) pushing the application down into the DHT layer. Our work adopts the latter. The current section describes the required modifications in the Kelips base design - the details of the soft state at each node, the handling of lookups, and finally, where and how the soft state is refreshed. Section 4.5 studies, through cluster-based experiments and trace-based simulations, how well this design supports the initially stated goals for decentralized web caching (viz., tolerance to churn, topologically local access, good hit ratios for low latency and low server bandwidth, and load balancing).

Soft State at A Node For our web caching application, Kelips is used to replicate a *directory table* for each cached object. A directory table is a collection of a small set of addresses of topologically proximate nodes that hold a valid copy of the object. This is depicted in Figure 4.1.

Concretely, a directory entry contains the following fields: node address *n_addr*; round-trip-time estimate *rtt*; timestamp record *tstmps*. *n_addr* is the address of the node hosting a valid copy of the object; *rtt* is the round-trip-time estimate to this node; *tstmps* is a collection of different timestamps w.r.t. the web object such as time-to-live, time of last modification etc. The *tstmps* fields are used to decide if this copy of the object is fresh at a given point of time.

Web Object Lookup A request for web object from the browser at a node is handled in the following manner. If a fresh copy of the object exists in the requesting node's local cache, it is returned to the browser. If a stale copy is found, the node sends a CGET request to one of its contacts for the object's affinity group. If the requesting node has not accessed the object previously, a GET request is sent to one of its contacts for the object's affinity group. The requesting node is itself used as the contact in the case when the requesting node's affinity group is same as that of the object's. At any node n, contacts for a foreign affinity group are maintained using a *peer maintenance* policy that constantly measures the round trip time to contacts, and listens to the membership heartbeat stream seeking to

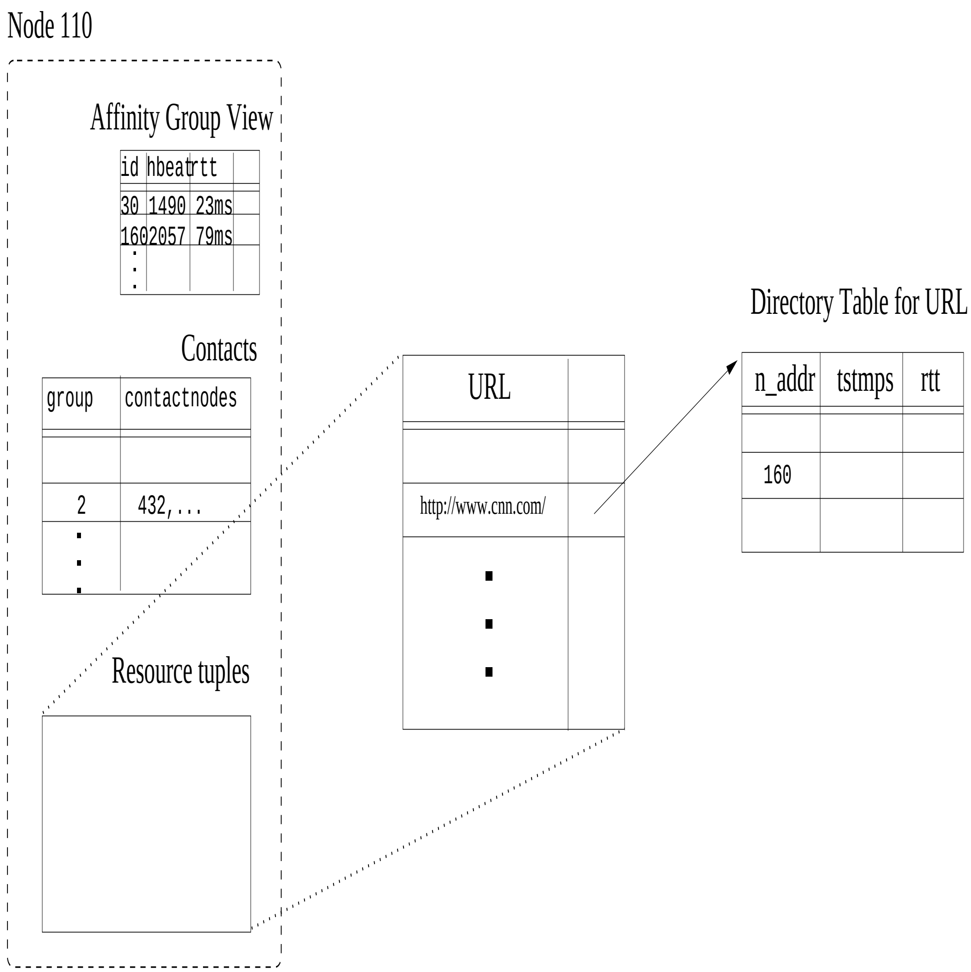


Figure 4.1: **Kache**: *Modified soft state at a Kelips node.*

replace the known contact that is farthest from node n with the newly heard-of candidate. Such a peer maintenance policy means that the GET request for an object will be sent to a contact that is topologically nearby to the requesting node.

When the contact receives a CGET request for an object, it first searches for the appropriate directory entry.

If the directory table contains at least one valid entry, the contact forwards the request to the topologically closest node among the entries (using the *rtt* field). This node in turn sends either a *not-modified* message or a copy of the object back to the requesting node. The topological proximity of the contact to both this node and the requesting node ensures access to a nearby cache of the requested object. If the triangle inequality for network distances is satisfied, the distance to the cached copy is at most the sum of the requester-contact and contact-cache distances.

If the directory table contains no valid entries, the contact has two choices - either to return a failure to the requesting node, or to forward the request for object to a peer in its own affinity group. For the former option, the requesting node subsequently contacts the web server directly with a GET/CGET. The latter request forwarding scheme can be generalized to a *query routing* scheme that uses multiple hops for routing a query try and multiple tries per query. The comparative performance of this multi-hop, multi-try (MM) scheme and the basic single-hop (SH) scheme is evaluated experimentally in Section 4.5.2.

Where Soft State is Maintained and How it is Updated When a node n successfully fetches a copy of an object o not accessed previously by it, the node creates a directory entry $< o, n >$ and communicates it to the contacts for o's affinity group. The contact first searches for object o's directory table, creating one if necessary. If the table is empty, a new entry is created for $< o, n >$. An expired duplicate entry for $< o, n >$ is replaced by a new one. If the table is full, the contact measures the round trip time to node n - if this exceeds the highest rtt field among directory table entries, the latter entry is replaced by a new $< o, n >$ entry.

Similar to the unmodified Kelips protocol, all object tuples are subject to selection for inclusion in a gossip message, in order to disseminate the new tuple $< o, n >$ within o's affinity group. However, recall from Section 4.3 that gossip targets are chosen through a topologically aware distribution (spatial distribution based on round trip times). Thus, gossip messages tend to flow between nodes that are topologically close.

Now, when only a few nodes in the entire system have accessed the given object o, one would ideally want all the nodes in o's affinity group to point to these nodes. However, when the number of cached copies of o rises, and as directory tables begin to fill up, a new tuple $< o, n >$ not previously inserted would replace entries in directory tables of nodes close to node n only. Thus, spreading tuple $< o, n >$ through gossip to nodes that are topologically far from node n will have low utility. This is achieved by associating a hops-to-live htl field with the disseminated tuple being disseminated through gossip.

The first contact spreading $< o, n >$ initializes the htl field to a small number `HTLMAX` (set to 3 in our experiments). htl is decremented at a node if $< o, n >$ is not inserted into the directory table for o. A tuple $< o, n >$ received with $htl = 0$ is not gossiped further.

When there are a large number of clients caching a valid copy of a given object, the effect of the combination of the above scheme and spatial gossiping is twofold.

Firstly, the directory entries maintained by a contact are topologically nearby to the contact. Secondly, as the number of cache copies of a given object rises, the background bandwidth used to propagate information about a new node hosting a copy of the object decreases.

4.5 Experimental Evaluation

We evaluate the performance of a C prototype implementation of Kelips web caching. The evaluation consists of (a) cluster-based microbenchmarks to examine the memory usage of the application and the soft state consistency, and (b) trace-drive experiments to study the system on a larger scale. The latter study is based on a combination of three traces/maps - client access web traces obtained from the Berkeley Home IP network [Dav], transit-stub network topology maps obtained through the Georgia-Tech generator [URLa], and churn traces from the Overnet deployment (obtained from the authors of [BSV03]).

4.5.1 Microbenchmarks: Small PC Cluster

This section presents microbenchmarks of the core Kelips component of the web caching application running within a commodity PC cluster. The cluster consists of commodity PCs each with a single 450 MHz - 1 GHz CPUs (PII or PIII), 256 MB - 1 GB RAM, and running Win2KPro over a shared 100 Mbps ethernet. A single node called the "introducer" is set aside to assist new nodes to join by initializing their membership lists.

We investigate actual memory utilization of the Kelips application and the consistency of membership soft state for a small cluster.

Memory Utilization Figure 4.2 shows the memory utilization at the introducer (triangles) and other nodes (x's) for different group sizes. The base memory utilization is low: less than 4 MB for the introducer at a group size of 1, and less than 2 MB for other nodes at a group size of 4. The rise in memory usage due to an increase in group size is imperceptible for all nodes. We conclude that memory usage in Kelips is modest.

Soft State Consistency In the experiment of Figure 4.3, 17 nodes join a one-affinity group system. The background gossiping bandwidth is configured so that at each node, 2 heartbeat entries (each 10 B long) is sent to 5 gossip targets chosen uniformly at random every 2 s. The heartbeat time-out is set to be 25 s. The solid line shows the view size measured at one particular node in the system. The crosses depict the distribution of heartbeat ages received at this node from the gossip stream. The numbers are clustered around less than 10 s for group sizes of up to 14. However, there are a few outliers - the ones beyond 25 s lie at times t=220 s, t=290 s, and t=345 s. On closer observation of the solid line, each of these leads to one node being deleted from the view. This explains why there are 14 =(17-3) nodes in the affinity group at time t=350 s.

4.5.2 Trace-Based Experiments

We study the performance of Kelips web caching through trace-based simulations. Multiple client nodes were run on a single host (1 GHz CPU, 1 GB RAM, Win2K) with an emulated network topology layer [2]. The experiments in this section combine three traces - network topologies, web access logs and p2p host availability

[2]Limitations on resources and memory requirements restricted current simulation sizes to a few thousand nodes.

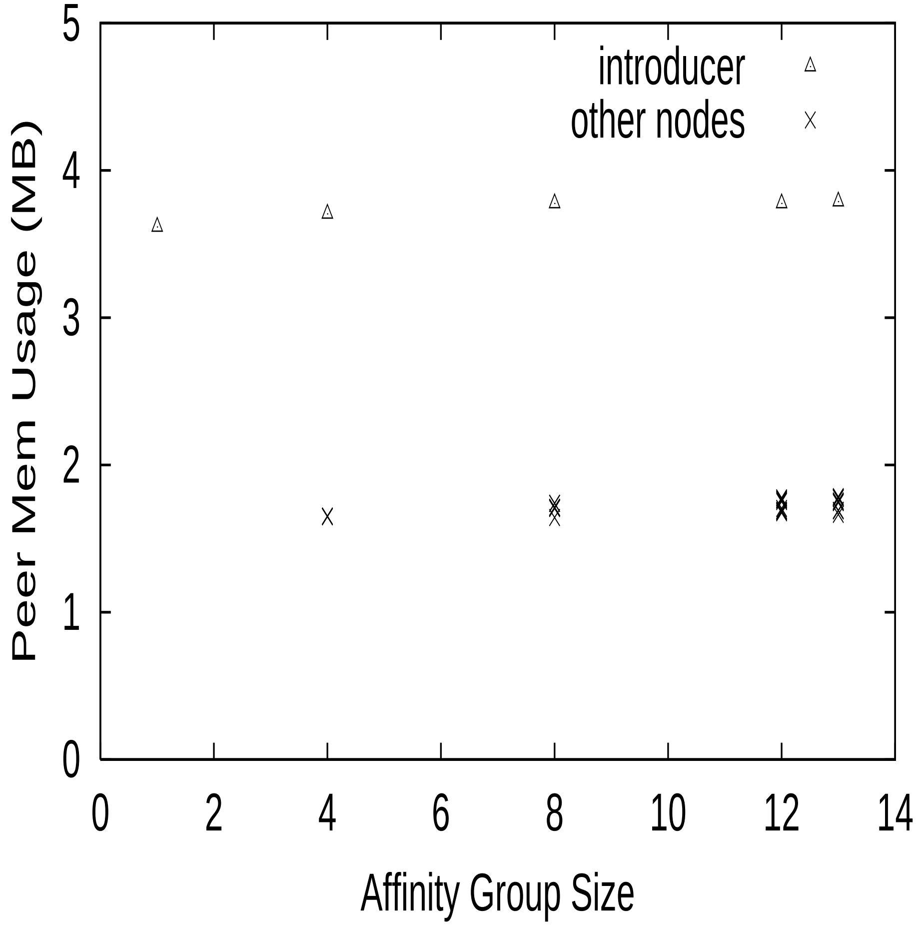


Figure 4.2: **Cluster Microbenchmark - Memory Usage of the Kelips Application in a Cluster:** *Memory usage in Win2KPro-based hosts, at the introducer node and other nodes*

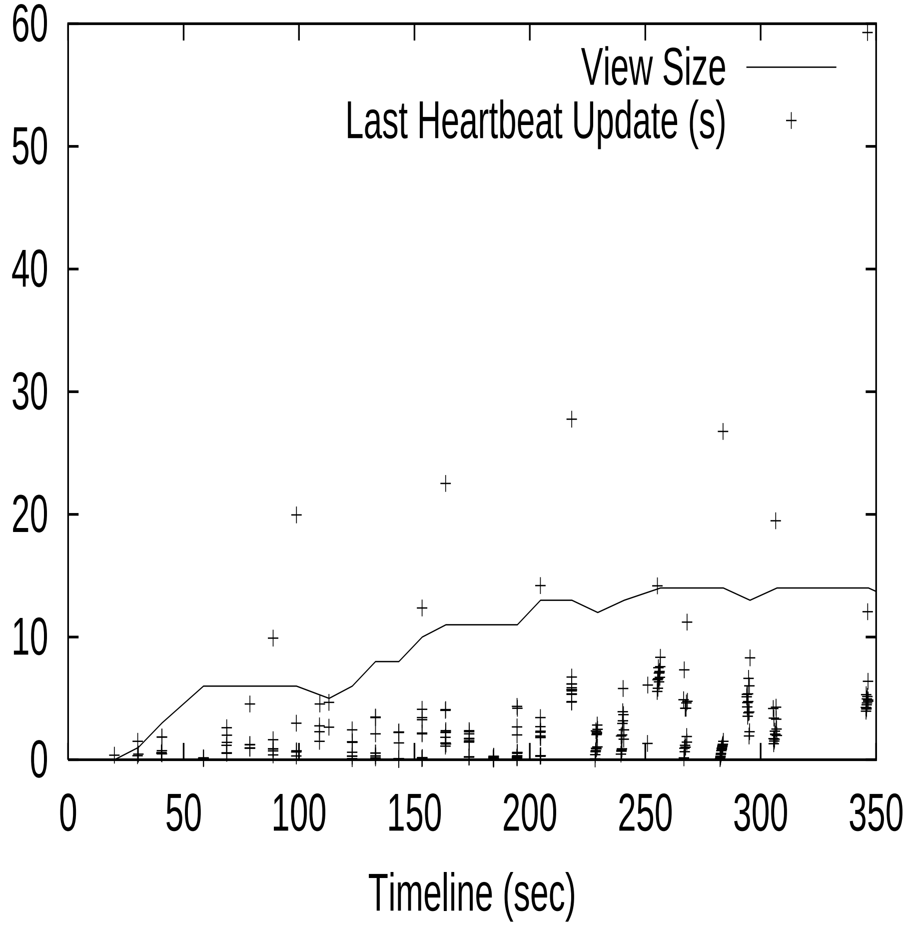


Figure 4.3: **Cluster Microbenchmark - Distribution of heartbeat ages and view size at a particular node**

Workload Traits	
Number of Clients	916
Total reqs	82142
Total cacheable reqs	75363
Total reqs size	558.9 MB
Total cacheable reqs size	523.3 MB
Total objs	47585
Total cacheable objs	43041
Trace duration	12200 s
Mean req rate	6.73 reqs/s
Perf. of Central Cache	
Total external bandwidth	393.3 MB
Avg. Ext. b/w per req.	4.78 KB
Hit ratio	0.331

Table 4.1: **Workload Characteristics and Centralized Cache Performance on the Berkeley HomeIP web access traces used.**

traces. We enumerate on the first two, and defer a description of the p2p host availability trace until later in the section.

The underlying network topology is generated using the well-known GT-ITM transit stub network model [URLa]. The default topology consists of 3 transit domains, with an average of 8 stub domains each, and an average of 25 routers per stub domain. Each Kelips node is associated with one host, and this host is connected to a router that is selected uniformly at random from among the 600 in the topology. Stubs are connected to each other with probability 0.5, and routers are connected to each other with probability 0.5. Network links are associated with routing delays, but congestion is not modeled.

The Berkeley HomeIP web access traces [Dav] are used to model object access workloads at Kelips nodes. Each web trace client is mapped to one Kelips node. The characteristics of the traces used are presented in Table 4.1. The last two rows in this table contain numbers corresponding to a single centralized proxy cache with infinite storage. These two numbers are the optimum achievable for this particular trace, with any caching scheme.

Finally, the Kelips group is configured as follows. The default number of participants (nodes) is 1000, and the default number of affinity groups is 31. The single-hop (SH) query routing scheme is the default. Background gossip communication was calculated to consume a maximum of 3 KBps per node. The number of directory entries per web page is limited to 4. We do not limit the cache size at each node, but we study the variation of maximum cache size with time and show that the maximum cache size stays low for the access trace considered.

External Bandwidth Figure 4.4 shows, over 500 s intervals, the aggregate bandwidth sent out to web servers due to misses within the p2p web cache. The external bandwidth due to Kelips web caching (dashed line) is comparable to that obtained through a central cache (dotted line).

Hit Ratio Hit ratio is the fraction of requests served successfully by the p2p cache. Define "oaf" as the number of times an object is accessed throughout the entire trace. As expected, the hit ratio rises with oaf (Figure 4.5). The plot appears to level out beyond a value of oaf=20.

Single Hop (SH) versus Multihop (MH) In the multi-hop scheme, a request is retried at most 4 times. Out of the four retries at most 2 retries are sent out directly to a contact. The rest are first forwarded to a node in its own affinity group in search of other potential contacts. Each request is routed for at most 3 hops in the requesting nodes affinity group and for at most 3 hops in the target affinity group.

Table 4.2 compares the hit ratio and average external bandwidth per request for the single hop (SH) and multi-hop (MH) query routing schemes. A comparison with Table 4.1 shows that the performance of both SH and MM Kelips web caching schemes are only slightly worse than that of the centralized cache scheme. The single hop query routing suffices to achieve as good hit rate as multi-hop, multi-try query routing. The only condition under which MM would be advantageous over SH is if either (a) an insertion of a web object tuple are followed so closely by queries for it (from other nodes) that the resource tuples might not be fully replicated, or (b) high churn rates cause staleness of membership tuples so that the single contact tried by the SH scheme is down. It is evident from Table 4.2 that

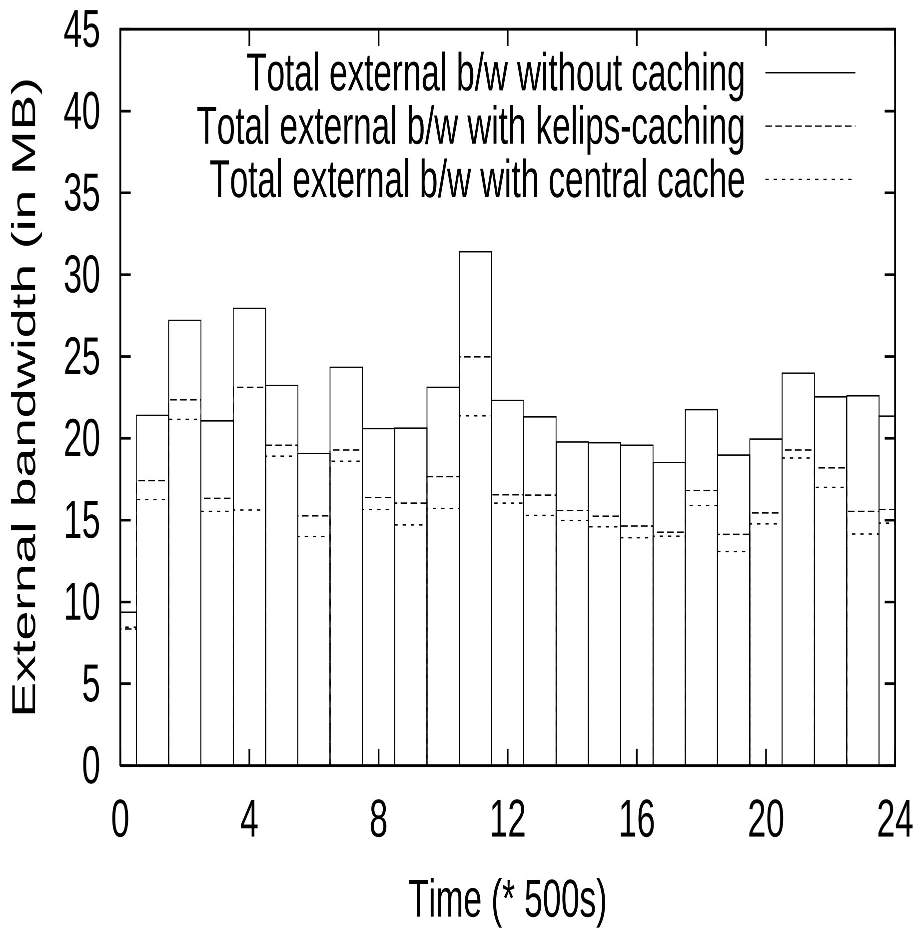


Figure 4.4: **External bandwidth vs Time**: *Kelips web caching is comparable to that obtained through a central cache*

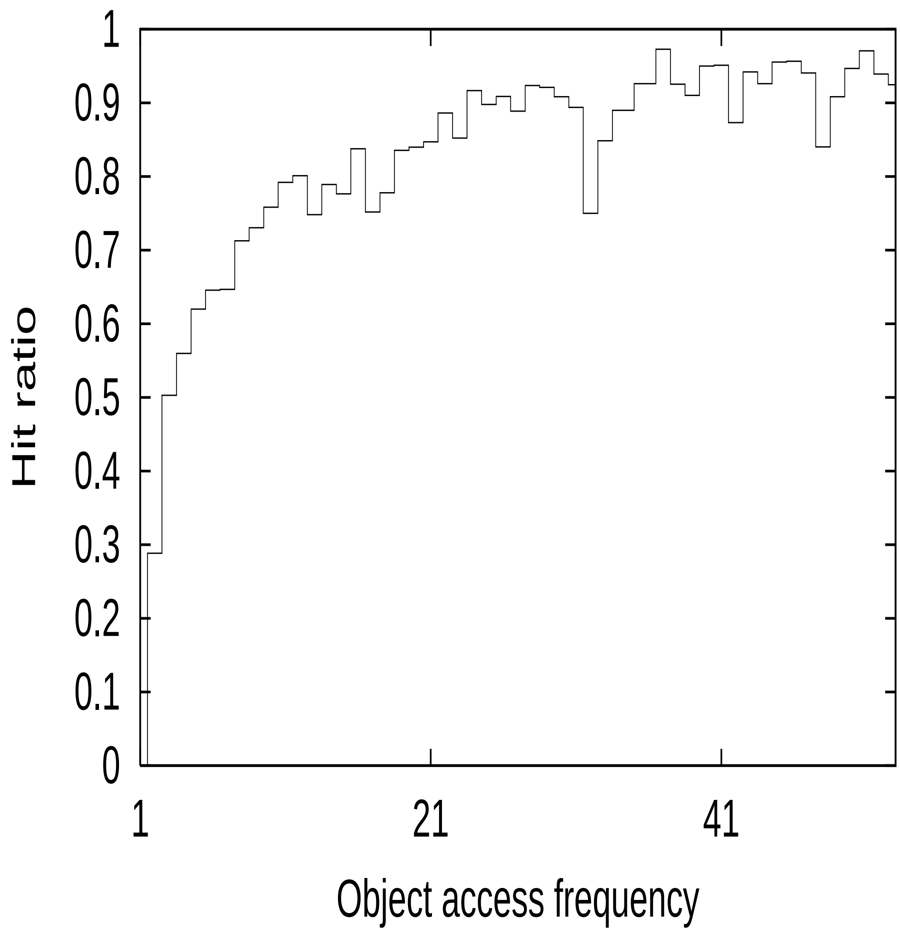


Figure 4.5: **Hit ratio vs Object access frequency**: *Objects accessed more frequently have higher hit ratios, saturating out at beyond oaf=20.*

Scheme	Ext. b/w	Hit Ratio
SH	5.63 KB	0.317
MM	5.5 KB	0.323
SH + churn	5.65 KB	0.313
MM + churn	5.46 KB	0.323

Table 4.2: **Average external bandwidth per request and hit ratio for single hop (SH) and multi-hop multi-try (MH) query routing schemes.**

(a) is not true for the web trace workload under study. The reasons why churn rates considered do not affect the hit ratio is explained later in Section 4.5.2.

Access Latency We measure two types of latency: (a) (*Time to find a target node address*) the time taken to resolve a request and return the address of a cache or report a cache miss to the requesting node, and (b) (*Time to reach a target node*) in the case of an external cache hit, the total time for the request to reach a node with a valid copy of the object (from the time when the request has been first issued at the requesting node). We do not measure the total time to fetch the object as this is a function of the object size. Figure 4.6 and 4.7 show these two numbers for the SH query routing scheme. The plots have a bimodal distribution, with a lower peak at a zero latency (local cache hit). Most requests are resolved within 1000 ms, and the total time taken to reach the target cache is within 1200ms for most requests. These plots demonstrate that access latencies are low and confirm the locality awareness of Kelips-caching.

Load Balancing We investigate the load balancing of requests for web objects in Figure 4.8. We consider one popular cacheable object. The requests received

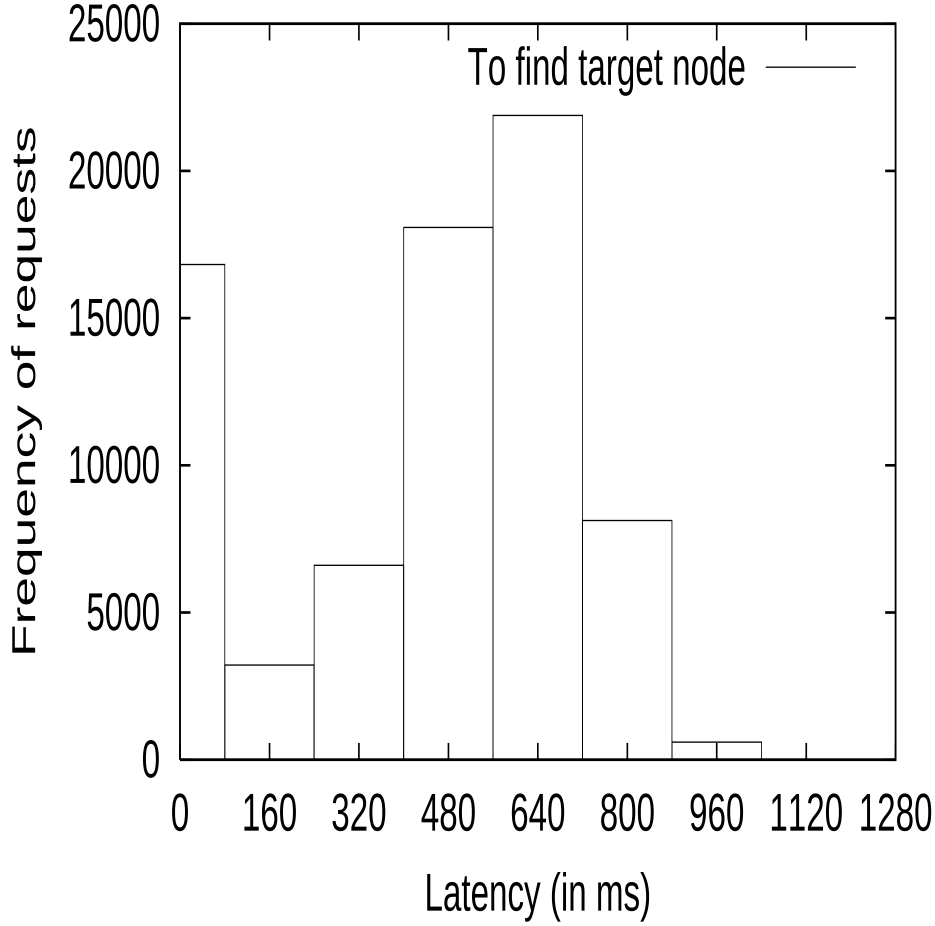


Figure 4.6: **Request frequency vs. Latency to search for a cache copy**: *Plotted for all requests to the external cache*

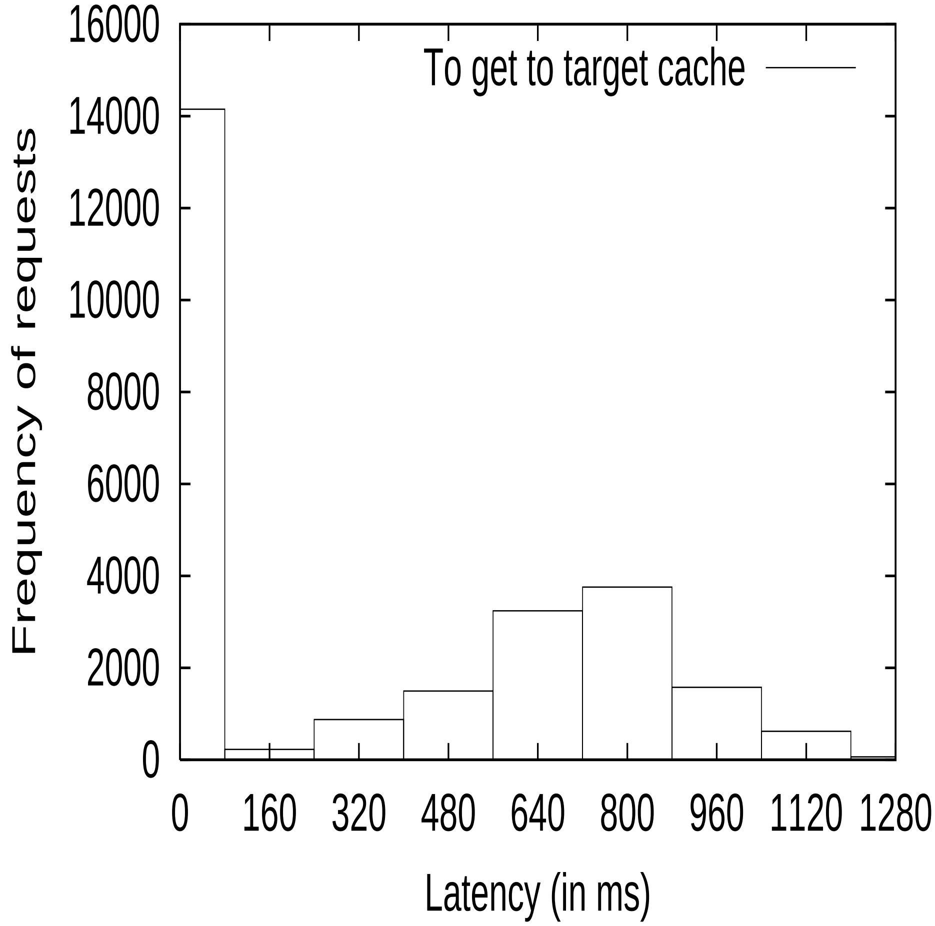


Figure 4.7: **Request frequency vs Latency to access cached copy**: *Plotted for requests that result in external cache hits*

for this object that are served successfully by the p2p cache system are assigned a global sequence number and plotted on the x-axis. The accessing nodes are ordered globally by their time of access on the y-axis. Each data point (x, y) shows that request number x was served at the node with global sequence number y. If points on this plot were clustered along horizontal lines, it would mean that a few nodes were taking most of the hits. An examination of Figure 4.8 shows that this is indeed not the case. Kelips web caching thus achieves good load balancing w.r.t. object requests.

Cache Size Figure 4.9 shows the variation of cache size with time during the simulation, and validates our infinite cache size assumption since the maximum cache size measured was smaller than 10 MB over the trace of duration 12,200 s.

Background Bandwidth We investigated the effect of varying the background gossip bandwidth (i.e, bandwidth used at end nodes) on the performance of the web caching scheme. We observe from Figure 4.10 that hit ratio decreases with decreasing background bandwidth since web object tuples are replicated less widely, and thus fewer queries hit a node with fresh tuples. Yet, the decrease is not substantial - from 4.35 KBps to 0.84 KBps, the hit ratio decreases by 0.005.

Effect of Churn: Constant Node Arrival and Departure Rates

The experiments in reference [GBL$^+$03] studied the effect of multiple node failures, measured the time for membership convergence, and showed that Kelips continues to ensure that lookups succeed efficiently under such stresses. In this section, we study the effects of a more general class of stresses arising from "churn" in the system - rapid arrival and failure (or departure) of nodes - on our implementation

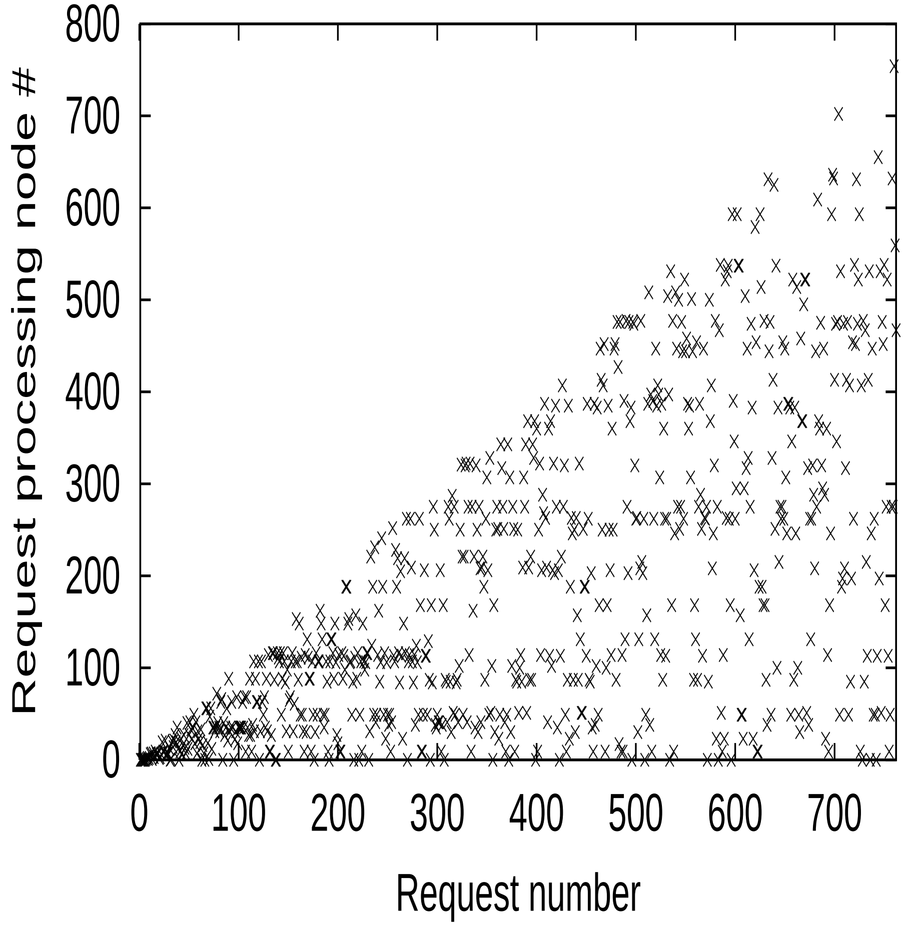


Figure 4.8: **Req processing node# vs Req num (see text in "Load Balancing" for explanation)**

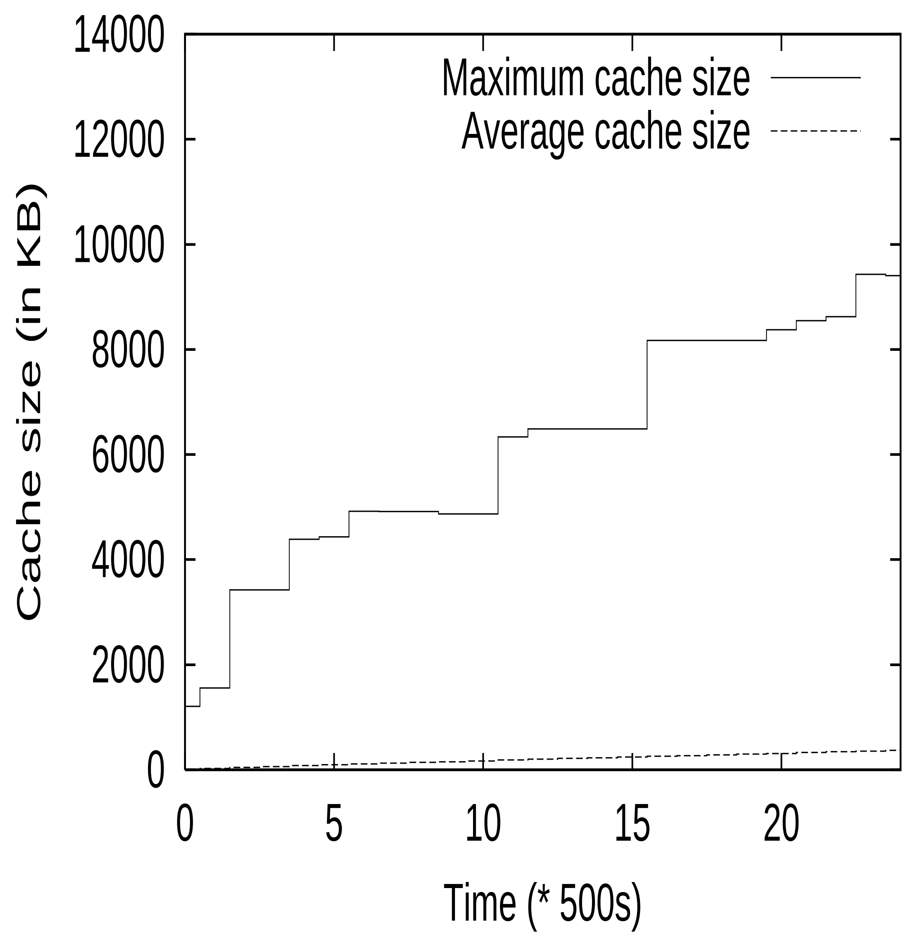


Figure 4.9: **Cache size vs Time**: *Average and Maximum cache sizes are smaller than 10 MB throughout the trace of duration 12,200 s*

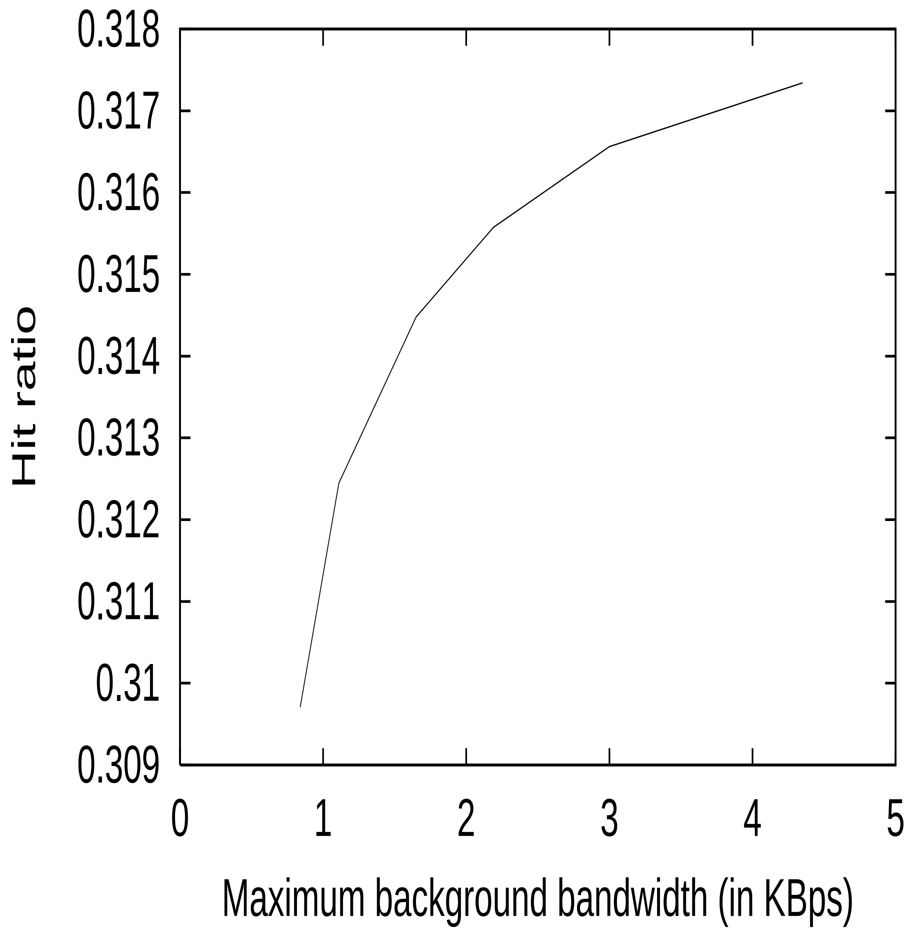


Figure 4.10: **Hit ratio vs Max background bandwidth**: *Increased background gossip communication cost affects the hit ratio by increasing the number of fresh web object tuples*

of web caching.

Our study uses client availability traces from the Overnet p2p system, obtained through the authors of reference [BSV03]. These traces specify at hourly intervals which clients (from a population of 990) are logged into the system. Typically, about 20% of the 990 clients are up at the start of each hour, and the hourly turnover rate varies between 10% - 25% of the total number of clients that are up.

Effect on Membership Each Kelips node in a 990-node system (with 31 affinity groups) is mapped to a node in this trace. Hourly availability traces are then injected into the system periodically at the start of *epochs* (rather than continuously) - given the hourly availability traces, this injection models the worst case behavior of Kelips from the churn.

Figure 4.11 shows the average affinity group view size when a new churn trace is injected every 200 s (in other words, 1 hour in the availability traces is mapped to 200 s). This epoch is more than the average stabilization period of the current Kelips configuration. As a result, one sees that soon after the trace injection at the beginning of an epoch, there is first a surge in membership size as information about returning nodes is spread through the system. This is followed by an expiry of nodes that have become unavailable due to the trace injection. In most epochs, the membership stabilizes a little before the end of the start of the next epoch [3].

Figure 4.12 shows the same experiment with churn traces injected every 40 s. The effect of such a low injection epoch is dramatic – the system suffers considerable pressure and is unable to cope with rapid membership changes. Before the membership changes from the last trace injection can be spread or detected by

[3] The epoch starting at 1800 time units is an exception. In this case 200s was not quite enough time for the system to stabilize

the system, a new trace is injected. As a result, the size of the membership lists thrash. Even after churn traces have been stopped being injected at time t=4000 s, the system takes considerable time to recover.

Effect on Hit Ratio, Access Latency Deployments of peer-to-peer applications tend to invite both nodes that are long lived and thus available most of the time, as well as nodes that exhibit churn behavior [SGG02]. For this experiment, we choose an operation point where 50% of the nodes in the Kelips system are available, and the remaining 50% are churned. More specifically, in a system of 1000 nodes, 500 nodes were churned by mapping to the first 500 entries in the Overnet availability traces[4]. The default churn trace injection epoch was set to 200 simulation time units. The other 500 nodes were kept alive throughout the trace, and requests were issued to the trace through these.

Figures 4.13 and 4.14 show the request latency distributions under the effect of churn. A comparison with Figures 4.6 and 4.7 respectively, and a glance at Table 4.2, show that churn has an *a negligible effect on the hit ratio and access latency distributions.* This happens in spite of membership tuples varying as shown in Figure 4.11, and the use of only single hop (and not multi-hop multi-try) query routing.

This churn-resistant behavior arises from the proactive contact maintenance policies used in Kelips. Recollect that when a Kelips node hears about another node in a foreign affinity group, it uses this node to replace the farthest known contact for the foreign affinity group. In addition, recollect that when a contact entry expires (as might happen when the contact node is being churned), the

[4]This is justified by the results of [BSV03] showing that availability characteristics tend to be uncorrelated across clients.

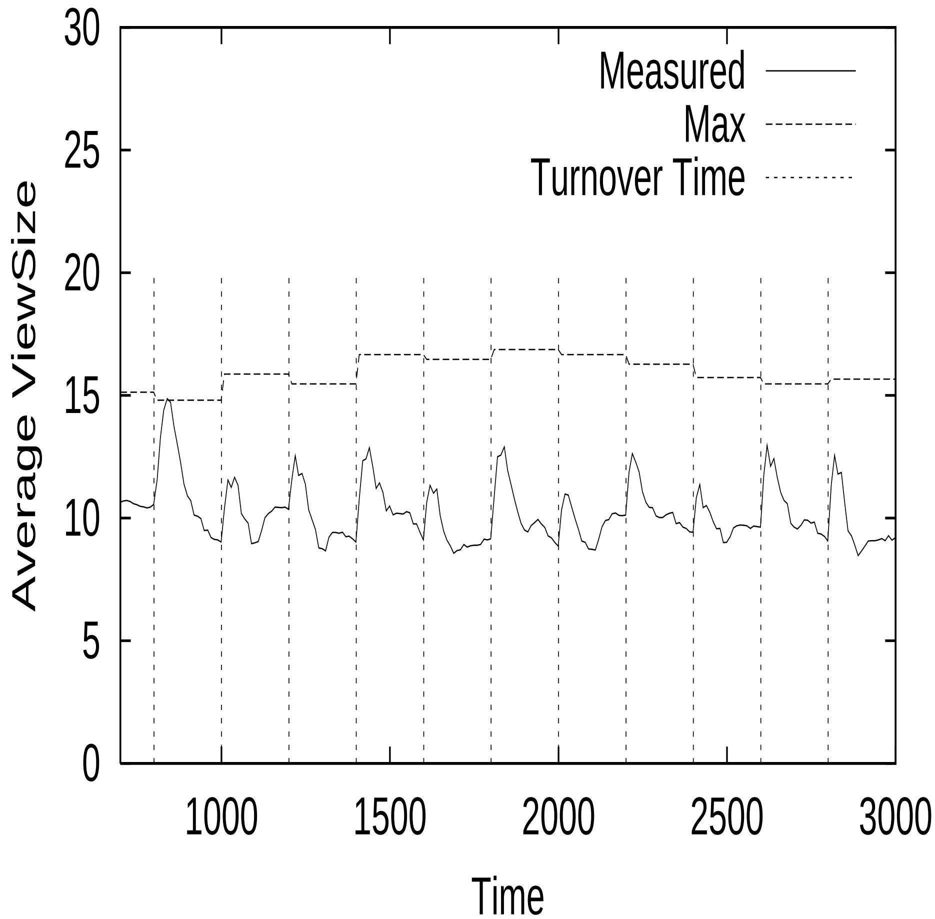


Figure 4.11: **Effect of churn on Affinity Group View Size at a node**: *Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 200 s*

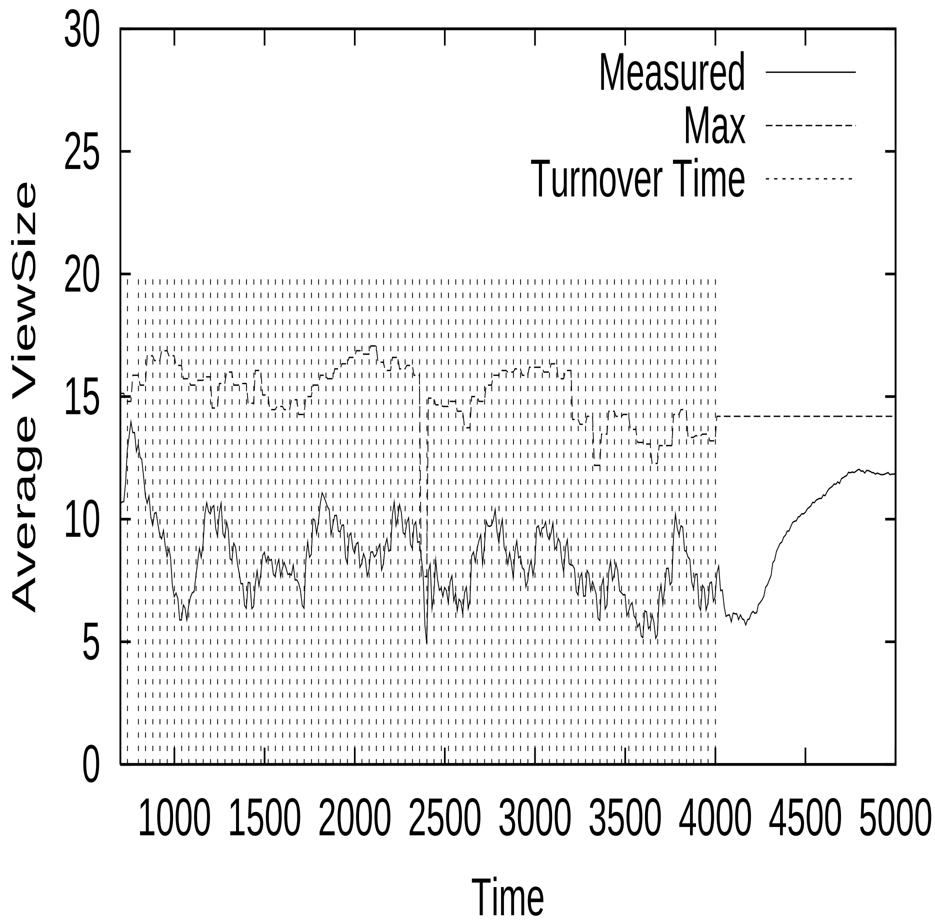


Figure 4.12: **Effect of churn on Affinity Group View Size at a node**: *Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 40 s*

expired entry is retained for a time duration to prevent stale copies for that node from being reinserted into the contact list within the specified timeout. Since the retention timeout is set to an excess of 200 time units in this experiment, the above two algorithms result in each Kelips node settling on a set of contacts that are nearest to it, as also highly available (not churned). Queries thus get routed mostly among the nodes that are stable, thus succeeding as often as in the simulation runs without churned nodes.

Figure 4.15 shows that hit ratio decreases by an insignificant amount (0.006, 2% decrease) as the churn trace injection epoch is decreased from 240 s to 20 s. Even when affinity group membership entries are thrashing at a churn trace injection epoch of 40 s (as shown in Figure 4.12), the hit rate is 30.9%, only 0.4% below the hit rate with a churn trace injection epoch of 200 time units. The reasoning behind this plot follows along the same lines as in the previous paragraph.

Note from Figure 4.7 that the number of requests which are local hits is about 18.8% of all the cacheable requests. Although a large portion of the cache hits from Figure 4.7 (54.4%) are local, we focus on the stability of the non-local p2p cache hits (the "remaining 45.6%"). From Figure 4.15, we see that a large fraction of these hits are retained even when there is excessive churn in the system.

This study thus substantiates our claim that Kelips web caching survives high rates of churn attack on the system.

4.6 Summary

Peer-to-peer applications may be subject to denial of service attacks from extreme stresses with origins typically considered non-malicious. We have studied one such source called churn, that arises from rapid arrival and failure (or departure) of a

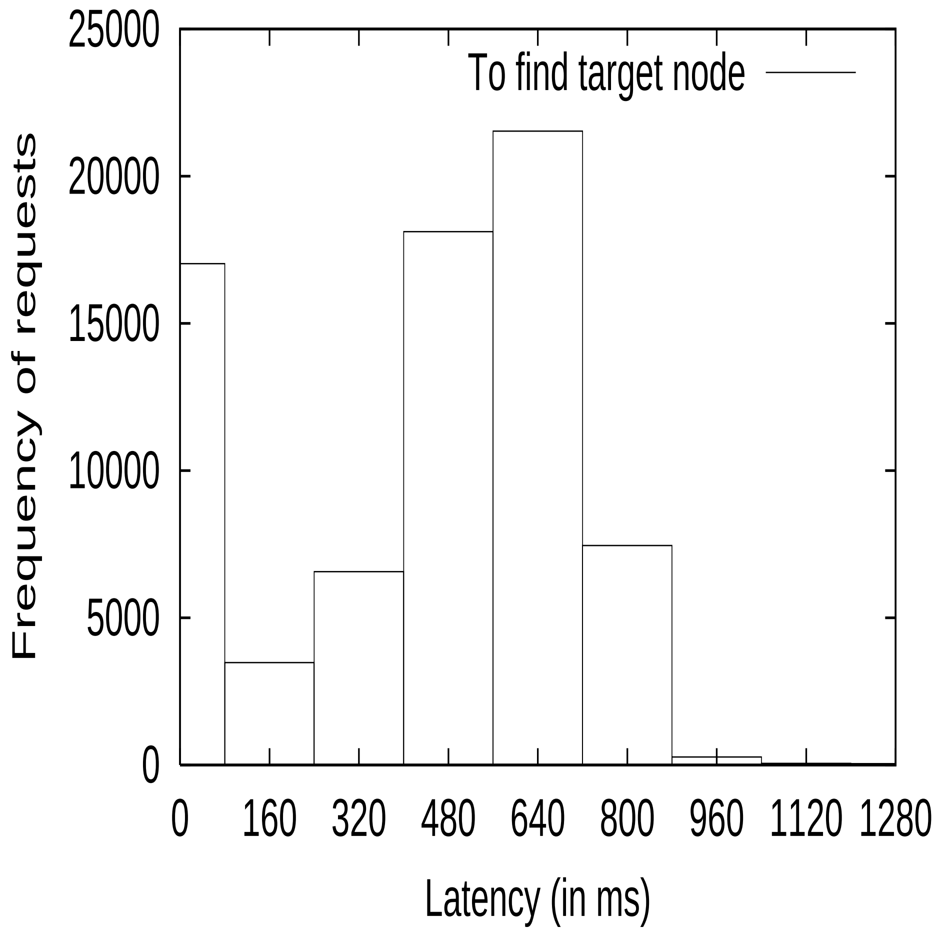


Figure 4.13: **Effect of Churn - Request frequency vs. Latency to search for a cache copy**: *Plotted for all requests to the external cache. Churn trace injection epoch is 200 s*

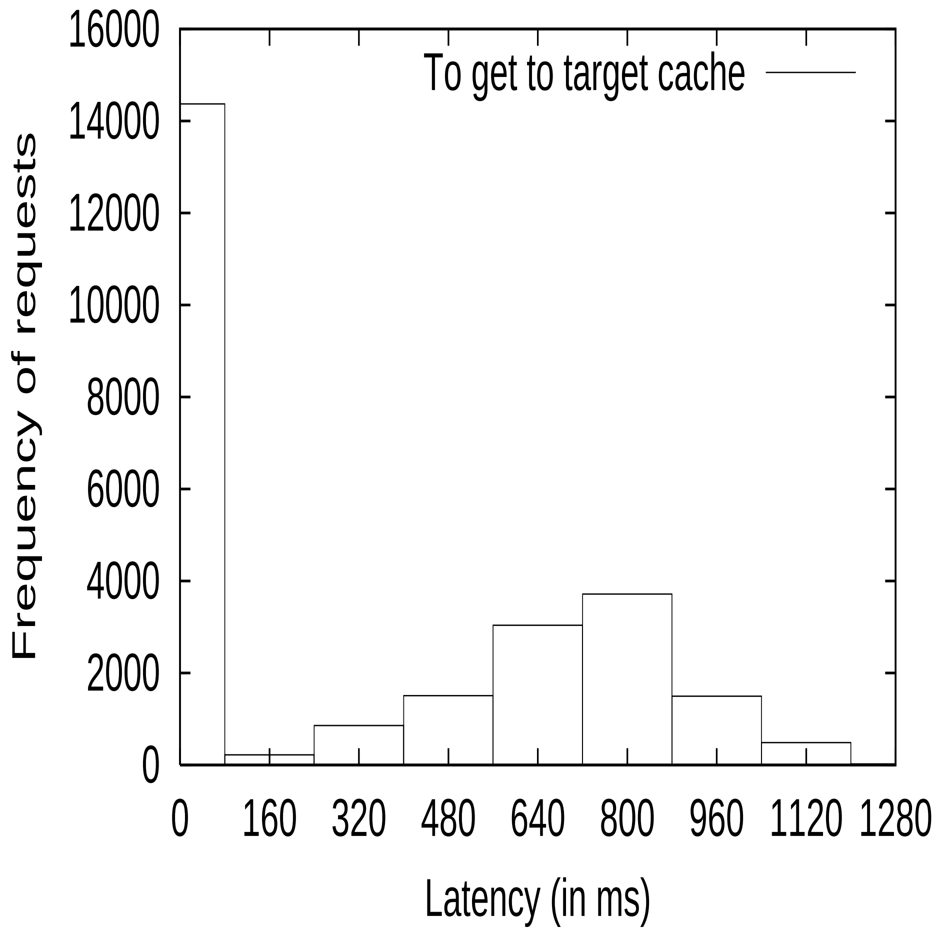


Figure 4.14: **Effect of Churn - Request frequency vs Latency to access cached copy**: *Plotted for requests that result in external cache hits. Churn trace injection epoch is 200 s*

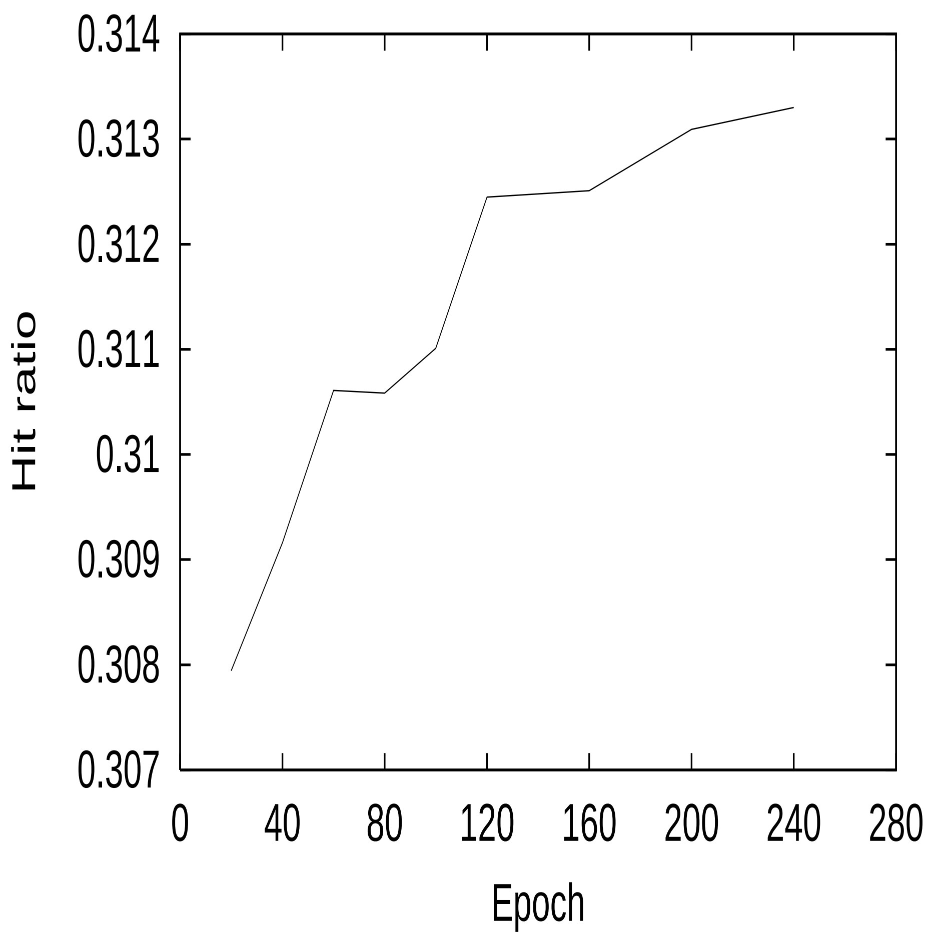


Figure 4.15: **Effect of Churn: Hit rate vs Churn trace injection epoch**

large number of participants in the system, and we have done so in the context of a peer-to-peer web caching application. Other malicious attacks are possible but we do not address these in this chapter. This chapter has shown how to design a churn-survivable peer-to-peer application. Our study has focused on the caching of web objects, and our solution has relied on the use of probabilistic techniques in the framework of the Kelips peer-to-peer overlay. Evaluation through microbenchmarking on commodity clusters, as well as experiments done through a combination of web access logs, transit-stub topologies, and p2p host availability traces, reveal significant advantages of locality and load balancing over previous designs for p2p web caching. Hit ratios and external bandwidth usage are both comparable to that in centralized web caching. In a system with a 1000 nodes, background communication costs as low as 3 KBps per peer suffice to ensure favorable and stable hit ratio, latency, external bandwidth use, and load balancing for access of web objects in the presence of system churn that causes 10%-25% of the total number of nodes to turn over within a few tens of seconds.

The investigation in this chapter can be extended to studies in several interesting directions - (1) the hit ratio and latency behavior of Kelips web caching at other operation points than the "50% available - 50% churned" above, (2) the effect of churn on caching scenarios other than web page browsing, and (3) the feasibility of the Kelips constant-cost low-bandwidth solution to other applications and other stressful networking environments.

Chapter 5

rKelips: An Efficient P2P Range Index

Peer-to-peer (P2P) systems provide a robust, scalable, and decentralized way to share and publish data. In this chapter, we propose a peer-to-peer data store that supports inserting and rapidly querying large amounts of geographic data or similar forms of data that can be inserted as $< type, key, value >$ tuples into the system. *key* may be the location, perhaps in terms of address or latitude and longitude. For example, the address "190 Pleasant Grove Rd, Ithaca, NY" can be represented as a one-dimensional key as "USA, NY, Ithaca, Pleasant Grove Rd, 190". Similarly, using latitude and longitude, it might be represented as a two-dimensional key as "(42, -76)". *value* is the data associated with this location. *type* specifies the type of the data, like "housing information" or "crime-rate data".

We are interested in supporting queries like:

- Find housing information for the location USA, NY, Ithaca
- Find crime-rate data for the location 80-90 deg latitude and 90-100 deg longitude (Users can select such a location region on one of the geo-centric web interfaces like Google maps.)

The example queries given above are range queries over the *key* (or location) attribute. We will assume that we have separate indices for different types of data and hence only $< key, value >$ tuples are indexed.

There are many proposals in the literature for peer-to-peer range index structures that will find $< key, value >$ tuples in the system that satisfy a given range query $[key1, key2]$. All these range indices are slow because they provide search performance of $O(log_d N)$, where N is the number of peers in the system and d is a tunable parameter.

We propose rKelips, a new peer-to-peer multi-dimensional range index that finds answers very quickly ($O(1)$ search performance). rKelips also guarantees that load on the nodes is roughly uniformly spread over all nodes, even in the presence of skewed insertion, non-uniform insertion rate across nodes, and non-uniform query rate across nodes. Moreover, rKelips is robust and works well even when half the nodes in the system fail. To the best of our knowledge, this is the first range index that provides $O(1)$ search performance.

In addition to querying geographic data, there are many other applications that can benefit from a rapid, robust, fault-tolerant and load balanced peer-to-peer range index. Some examples are:

- Service discovery on mobile devices: With increasing popularity of mobile devices, we envision a world where services will be advertised on mobile devices. rKelips enables quick service discovery in such an inherently p2p and high churn setting.

- Data retrieval in a data center: Google's BigTable and Amazon S3 store massive amounts of data on a cluster of computers and retrieve data using equality and range queries. In such a setting, rKelips can be used to retrieve data quickly while maintaining balanced load on the cluster nodes.

The rest of the chapter is organized as follows: In Section 5.1 we present the system model and the assumptions made. In Section 5.2 we describe the core design of rKelips. We present some additional algorithms in Section 5.3. In Section 5.4 we present the experimental results. In Section 5.5 we present related work and conclude in Section 5.6.

5.1 System Model

A *P2P system* is a collection of large number of nodes that share similar responsibility and hence are peers to each other. A *peer* is a processor with shared storage space and private storage space. The shared space is used to store the distributed data structure for speeding up the evaluation of user queries. We assume that peers in the system are cooperative, i.e, they follow a common protocol as long as they are part of the system. We assume that each peer can be identified by a physical id (for example, its IP address). A peer can join a P2P system by contacting some peer that is already part of the system. A peer can leave the system at any time without contacting any other peer. At any instant, we assume that clock skew between peers is bounded by δ where $2 * \delta < T_g$. Consider any time interval of $k + 1$ epochs on some peer n. Every other peer m will have at least k and at most $k + 1$ epochs during the same time interval. We assume that the underlying network is unreliable but characterized by probabilistic failure rates of peers and message deliveries.

We assume that each (data) item stored in a peer exposes a *search key* that is indexed by the system. The search key value for a item i is from an totally ordered domain $\mathcal{K}$ and is denoted by $i.key$. Without loss of generality, we assume that search key values are unique (duplicate values can be made unique by appending the physical id of the peer where the value originates and a version number; this transformation is transparent to users). Peers inserting items into the system can retain ownership of their items. In this case, the items are stored in the private storage partition of the peer, and only pointers to the items are inserted into the system. In the rest of the chapter we make no distinction between items and pointers to items.

We consider only equality or range queries. Range queries are of the form $[lb, ub]$ where $lb, ub \in \mathcal{K}$. Queries can be issued from any peer. For simplicity, we assume that the query distribution is uniform over all items, so the load of a peer is determined by the number of data items stored per peer. The algorithms introduced in this chapter can be extended to other definitions of load. We define the *load imbalance* in a system to be the ratio between the load of the most loaded peer and the load of the least loaded peer in the system.

5.2 rKelips : Core Design

Peer-to-peer overlays for object insertion and retrieval such as Chord [SMK+01], Pastry [RD01b], SkipGraph [AS03] and PRing [CLM+04a] define how each participant node chooses peers to which they maintain pointers. In contrast, inspired by Kelips [GBL+03], rKelips uses softer rules to determine sets of possible peer pointers, permitting a node to pick any peer within the set according to end-user considerations and constraints such as topology awareness, trust, security concerns, etc. The choice can vary over time, and this gives rKelips a "self-regenerating" behavior.

Like in existing p2p range indices [BRS04, CLM+04a], rKelips supports multi-dimensional range queries by using indices on single attributes. Multiple single attribute rKelips indices are overlayed over the same set of nodes. Given a multi-dimensional range query, selectivity estimates are used to pick the dimension to route through. The index corresponding to this dimension (or attribute) is then used to answer the multi-dimensional query. In this rest of the chapter, we will consider the problem of answering one-dimensional range queries.

5.2.1 Affinity Groups

A rKelips system with N nodes consists of $\sqrt{N}$ virtual subgroups called *affinity groups*, numbered 0 through $(\sqrt{N}-1)$. Each node lies in an affinity group, determined by using a consistent hashing function to map the node's identifier (IP address and port number) into the integer interval $[0, \sqrt{N}-1]$. Using SHA-1 as the hash function, each affinity group size will lie in an interval around $\sqrt{N}$ w.h.p. The value of N should be consistently known at all nodes, but can be an estimate of the actual system size.

As there are $\sqrt{N}$ groups, the key space is divided into $\sqrt{N}$ partitions, with partition boundaries at $R_0 \leq R_1 \leq \cdots R_{\sqrt{N}}$. Moreover, to make sure that the entire key space is covered with non-overlapping partitions, we require $R_0 = R_{\sqrt{N}}$. Every node in Group i maintains the range $[R_i, R_{i+1})$, $\forall 0 \leq i < \sqrt{N}$, and knows about the ranges maintained by other groups. In particular, all items with keys in range $[R_i, R_{i+1})$ are stored with all nodes in Group i.

At each node, rKelips replicates item and membership information. Membership information includes (a) the affinity group *view*, a (partial) set of other nodes lying in the same affinity group, and (b) for each foreign affinity group, a small (constant-sized) set of *contact* nodes lying in it. Each membership entry (affinity group view or contact) carries additional fields such as round-trip time estimates and heartbeat counts.

Figure 5.1 illustrates the state of node in a 10 group rKelips system. Node 110 is in Group 0 and is responsible for the range $[0, 50)$. Node 110 knows about all the other nodes in its group. It also knows some contacts from every other group and the range maintained by every other group. For instance, node 432 is a contact from Group 2 and the range maintained by Group 2 is $[50, 199)$. Node 110 knows the items that are in its range. Item with key 45 is one such item.

5.2.2 Inserting an Item

A node inserting an item i first determines the item's affinity group. Item's affinity group is g, where $i.key \in [R_g, R_{g+1})$. The item is communicated to a contact in the item's affinity group, which in turn disseminates it, perhaps partially, within the affinity group. With full tuple replication, a node can access an item given the key by one RPC to its contact for the item's affinity group.

The freshness of items is determined through the use of an integer heartbeat count. The inserting node periodically disseminates an updated heartbeat count to the item's affinity group. Similarly, each node n_i periodically disseminates a heartbeat count for itself, refreshing membership entries that are maintained for n_i at other nodes in the system. Thus, failure of n_i leads to purging of membership tuples for n_i (and items inserted by n_i) at other nodes.

Membership heartbeat counts need to be disseminated throughout the entire system (since contacts are maintained across affinity groups). So does membership information such as about joining, leaving or failed members.

5.2.3 Maintenance using Gossip

All dissemination in rKelips occurs through a continuously active, low-cost background communication mechanism based on a p2p epidemic-style (or *gossip-style*) protocol [Bai75, DGH$^+$87]. Gossip-based communication in rKelips proceeds as follows. A node periodically picks a few peers (from among its list of contacts

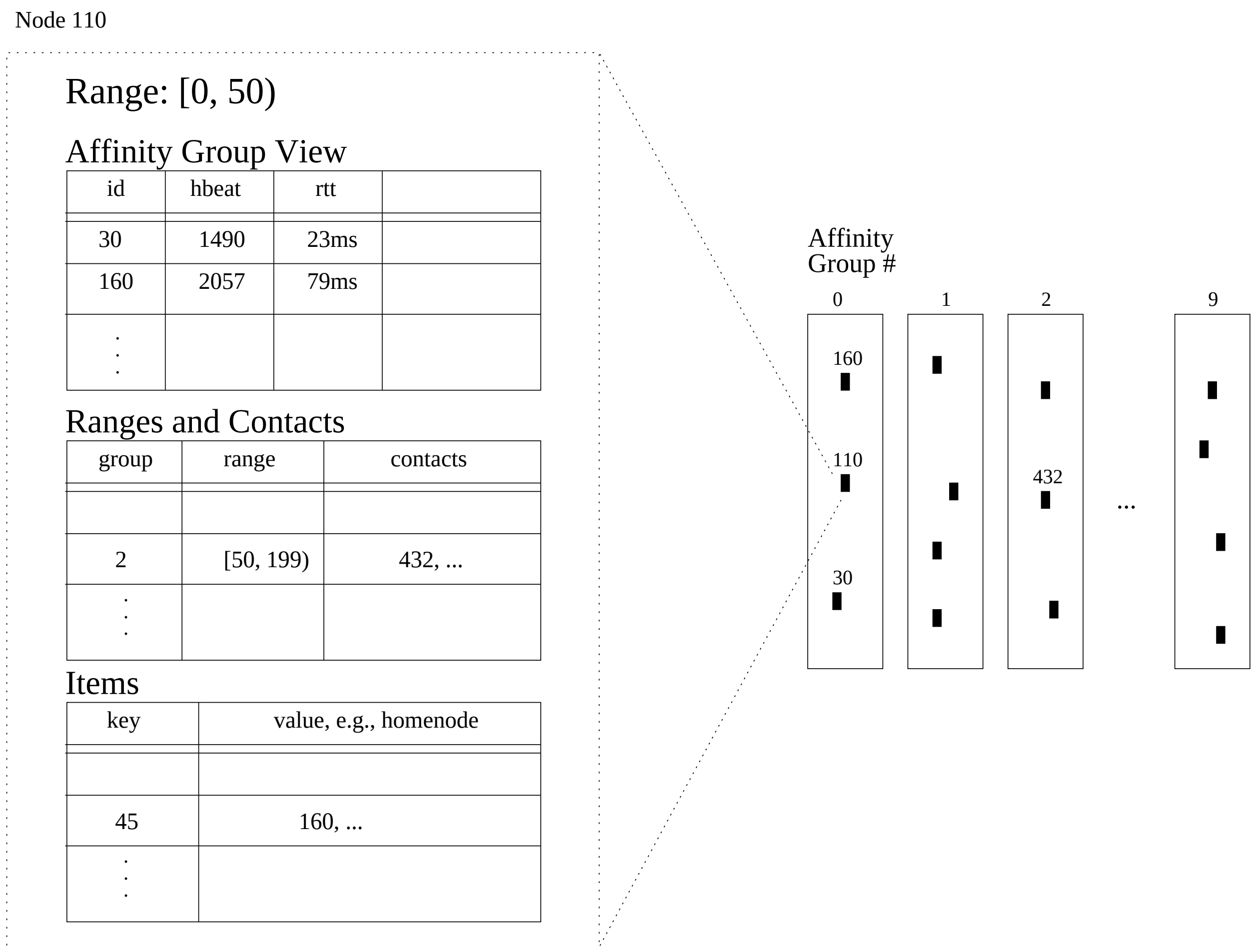


Figure 5.1: **rKelips soft state at a node:** A rKelips system with nodes distributed across 10 affinity groups, and soft state at a hypothetical node.

and its affinity group view) as gossip targets. These peers are picked based on a spatial distribution based on round trip times [KKD01], that as a consequence, prefers gossip targets topologically close to the node. The node then sends these nodes a constant sized gossip message (via UDP) containing membership and item entries selected uniformly at random from among those it maintains. These gossiped entries also contain the corresponding heartbeat counts. Tuples that are new or were recently deleted are explicitly added on to the gossip message for faster dissemination. Recipients update their soft state based on information obtained from the gossip message. No attempt is made to detect or resend lost messages.

The latency of dissemination within an affinity group depends on the gossip target selection scheme – it varies as $O(log^2(n))$ under the spatial gossiping scheme from [KKD01] and as $O(log(n))$ under uniform target selection. These latencies rise by a factor of $O(log(n))$ for dissemination throughout the system (i.e., across affinity groups) - see [GBL$^+$03]. Since gossip message sizes are limited, only a part of the soft state can be sent with each gossip message. This imposes an extra multiplicative factor of $O(\sqrt{n})$ for heartbeat updates. In fact, however, the constants are small for medium sized systems. In a system with a thousand nodes, a background bandwidth utilization of a few KBps per node suffices to have low dissemination latency ranges in tens of seconds [GBL$^+$03].

5.2.4 Answering Range Queries

Given a range query $Q = [r_1, r_2)$ at a node, the node sends this query to one of its contact for the group responsible for r_1. The contact then sends all items in its store that match query Q to the originating node. In addition, if there are potentially more items in the next group that match the query Q, it will forward the query to a contact from the next group. The forwarding will continue until Q reaches the group responsible for r_2.

Suppose range query $Q = [55, 65]$ is issued at node 110 in our example system from Figure 5.1. Node 110 will first find the group responsible for 55. In this example, it is Group 2. It will then forward the query to one of its contacts from Group 2. Suppose the query is forwarded to node 432. Node 432 will now return the items it has in the query range. Group 2 is responsible for 65 (right end of the query range) and hence the query processing is done.

5.2.5 Load Balancing

We would like data items to be uniformly distributed among peers so that the load of each peer is about the same (load imbalance is small). Most existing P2P indices achieve this goal by *hashing* the search key value of an item, and assigning the item to a peer based on this hashed value. Such an assignment is, with high probability, very close to a uniform distribution of entries [GBL+03, RFH+01, RD01b, SMK+01]. However, hashing destroys the value ordering among the search key values, and thus cannot be used to process range queries efficiently (for the same reason that hash indices cannot be used to handle range queries efficiently).

To help answer range queries efficiently, all current p2p range indices assigns data items to peers directly based on their search key value. The ordering of peer values is the same as the ordering of search key values. In rKelips, we use the same idea to order groups based on order of search key values and range queries are answered by scanning along consecutive groups.

Ordering data items on the search key value may lead to non-uniform distribu-

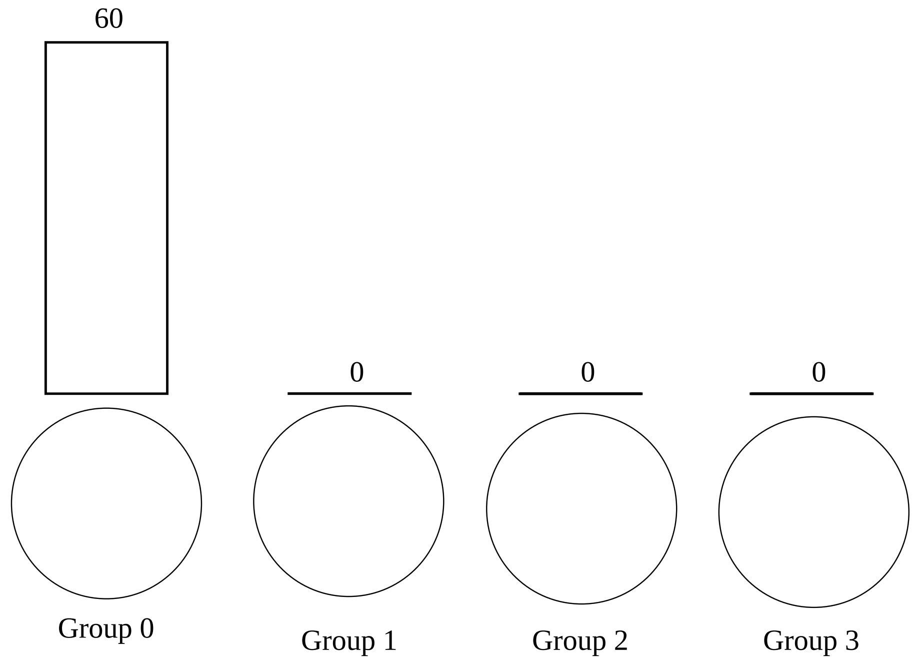


Figure 5.2: **Before load balance:** A 4 group rKelips system that is unbalanced due to skewed insertion of items

tion of items across groups, especially when the item distribution is non-uniform. We therefore need efficient techniques to ensure load balance across nodes.

If item distribution is uniform over the domain of the key attribute, assuming that partition boundaries $R_0 \leq R_1 \leq \cdots R_{\sqrt{N}}$ are chosen such that all partitions are of equal size, we are assured that each partition or group has the same number of items. Hence, each node has the same number of items i.e. has the same load.

If the item distribution is non-uniform, groups may have different number of items and hence non-uniform load on nodes. We will now describe how we perform load balancing to ensure that load on nodes is roughly the same.

Suppose we have an unbalanced rKelips system as shown in Figure 5.2. The basic idea is to determine new ranges for the groups such that (1) load imbalance of the system is small, and (2) not too many items are moved across groups. We

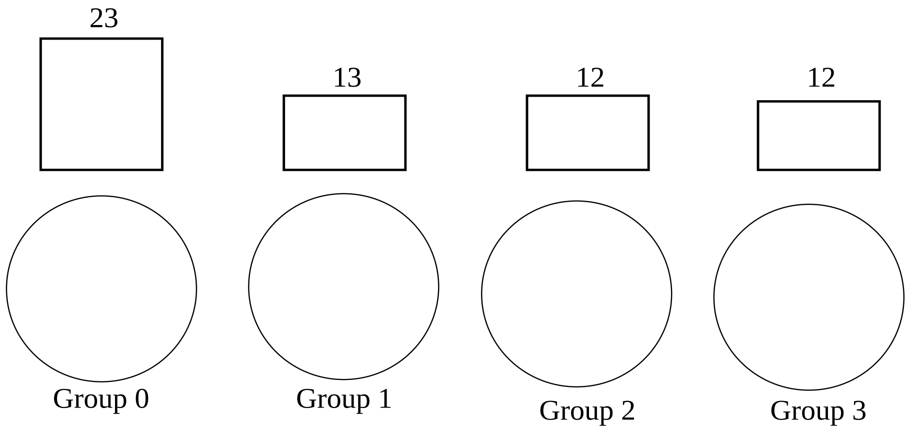


Figure 5.3: **After load balance:** rKelips system after load balance has a load imbalance of just 1.9

adapt the load balancing algorithm proposed in [GBGM04] for this purpose (see Section 5.3.1 for details). Items are then moved across groups to respect the new ranges. The example rKelips system after load balancing is shown in Figure 5.3.

To simplify our load balancing algorithm, we use a leader to compute a new partitioning of the key space that it then informs all the nodes in the system. This provides all the nodes in the system with a consistent view of the system. This is a very light weight operation. It is important to note that the leader is elected using a totally decentralized protocol and no node is dedicated to be a leader (see Section 5.2.6). Nodes take turns in assuming the role of a leader. Without going into the details, we would like to note that it is possible to totally decentralize our load balancing algorithm.

We will first describe the details of the load balancing scheme under the following assumptions:

1. All nodes have the same gossip time period T_g. We define the time between two gossip periods as one epoch.

2. At any instant, clock skew between nodes is bounded by δ where $2 * \delta < T_g$. Consider any time interval of $k+1$ epochs on some node n. Every other node m will have at least k and at most $k+1$ epochs during the same time interval.

3. Insert of any item i takes at most T_i epochs from the time the item is inserted at a node in the target affinity group to the time when all other nodes in the affinity group learn about the item i whp.

4. There exists a leader L which is alive during the entire load balancing step. L knows constant number of nodes from each group. This assumption is relaxed in Section 5.2.6.

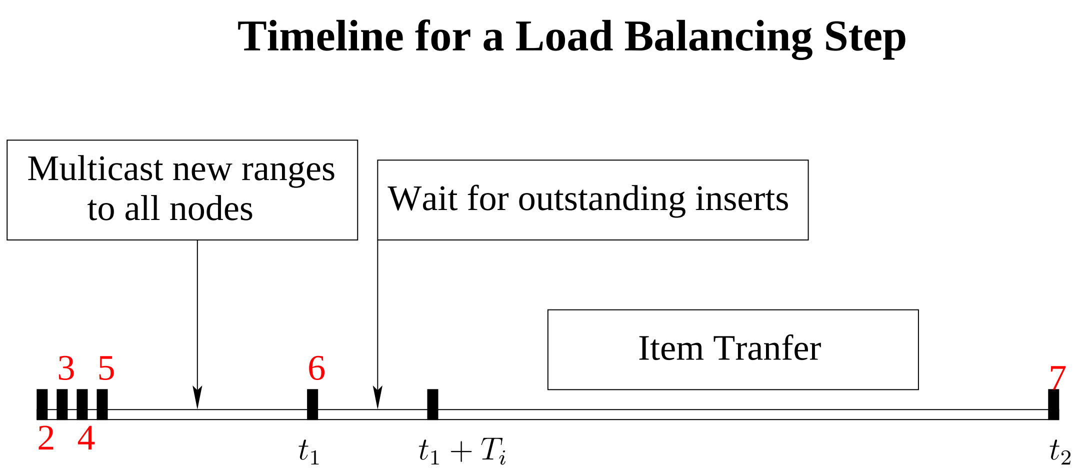
5. Load L_n on a node n is defined as the number of items at node n. This assumption is relaxed in Section 5.2.6.

As observed in Section 5.1, assumptions (1) and (2) guarantee that during any time interval of $k+1$ epochs on some node n, every other node m will have at least k and at most $k+1$ epochs. Assumption (3) guarantees that an inserted item is replicated to all nodes in the group whp in no more than T_i epochs. We expect item insert rate to be low for most applications and hence, $T_i = log(N)$ should suffice. Assumptions (3) and (4) are for the purpose of ease of exposition only. Later in this section, we will describe how these assumptions can be relaxed.

Under the above assumptions, we now outline our load balancing scheme (see Figure 5.4). We will use a toy example with four groups to explain our scheme.

1. Let s be the number of the current load balancing step. Initialize s to 0.

2. Leader L gets an estimate of the number of items in each group by contacting



2 Leader L gets number of items per group

3 L computes number of items that should be in each group

4 L will find the new ranges for groups to maintain load balance

5 L multicasts new ranges info to all nodes, including epochs t_1 and t_2

6 Nodes use the new ranges to route inserts
Lookups are resolved using both current and new ranges
Item transfer is initiated at epoch $t_1 + T_i$

7 Nodes replace current range with new range starting epoch t_2

Figure 5.4: **Load balancing in rKelips**

a member of each group. Remember that leader L knows constant number of contacts from each group. In our toy example, let's assume that the estimates the leader gets are: $30, 20, 5, 5$.

3. Leader L uses a load balancing algorithm to find the number of items that should be in each group to ensure load balance. We will use the load balancing algorithm proposed in [GBGM04] for this purpose (see Section 5.3.1). In our toy example, the load balancing algorithm will require the following number of items in each group: $23, 13, 12, 12$.

4. Leader L will now find the new ranges that the groups should have. L does this by getting an estimate of the value for the new range boundaries. In our example, leader will ask one of its Group 0 contacts to give the value of the 23^{rd} item it currently has. Similarly, it will ask its Group 1 contact for the value of $(23 + 13 - 30)$ i.e 6^{th} item it currently has and so on. Note that it does not really matter that the 23^{rd} item now may not be the 23^{rd} item when the number of items estimates were obtained.

5. Leader will now multicast to all nodes in the system the new ranges, the new step $s+1$, insert time T_i, an epoch t_1 indicating start time of next phase and an epoch t_2 indicating when nodes should switch to the new ranges. The leader will send the multicast to each of its contact in each of the groups. Gossip will spread this information to all nodes in the system. t_1 is calculated by the leader based on how long it takes for gossip to spread the information to all nodes. This is logarithmic in the number of nodes in each group. t_2 is calculated by the leader based on the time it takes for nodes to finish transferring items according to the new ranges. We will talk more about this

later in the section when we outline the strategies to transfer items across groups.

6. At epoch t_1 a node has either heard from the leader about the new ranges or it has not.

   Starting with epoch t_1, all inserts are routed according to the new ranges. Starting with epoch t_1, lookups are routed according to both current and new ranges i.e lookups will now reach one extra group and hence will take one additional hop.

   Case 1: Node n has heard about the new ranges: Node n will route start routing inserts according to new ranges at epoch t_1. Moreover, starting with epoch t_1, node n will route lookups according to both current and new ranges i.e lookups will now reach one extra group and hence will take one additional hop.

   Node will start transferring items at epoch $t_1 + T_i$. This enables all outstanding inserts in the group to finish replicating to all nodes in the group. Node will first learn the number of items that it is expecting before the item transfer begins.

   We propose to transfer items by piggybacking on gossip messages. For ease of exposition, lets assume that items in set I are being transferred from group i to group j. Nodes in group i and group j join a new group ij. Nodes in group i and ij insert items in I into group ij. These items will reach all the nodes in group ij (and hence group j) through gossip. Nodes can leave group ij once the item transfer is done.

   Case 2: A node has not heard about the new ranges: Note that with a very small probability, there might be a node that does not know about the new ranges. How is this taken care of? Gossip, insert and search messages have range consistency checks to detect nodes that have fallen behind. Whenever a message is sent from node n_1 to n_2, a check is performed to see if both the nodes are in the same step. If the two nodes are not from the same step, then the node (say n_l) which is lagging behind will catch up. Catching up involves the following operations:

- n_l will contact a node n_c in its group that is in the correct step.
- n_l will copy ranges, items and view/contact members of n_c.
- n_l will then insert all items in its store that should be in a different group into the appropriate group. This ensures that no items are permanently lost.

7. All nodes that have received the multicast from the leader will start using the new range at epoch t_2. They will also increment the step to $s+1$ at epoch t_2. Leader uses a conservative estimate for the number of epochs required to transfer all items to decide t_2.

5.2.6 Generalized Load Balance

We will now describe how assumptions 2 and 3 can be relaxed.

Leader Election

In this section, we show how assumption 2 can be relaxed. We discuss how and when a leader is elected in a decentralized fashion. We also discuss how the protocol works when the leader fails during the load balancing step.

We use techniques from [GRB00] to elect a leader. Every node, with a random delay, will perform steps 2 and 3 above to check if load balancing is required. A node performs this check only if is not aware of an outstanding load balancing step. If node n detects that load balancing is required, node n will assume leadership and will proceed with step 4. To further reduce the probability of many nodes performing the load balance check at the same time, we could have each node n calculate the hash (using a consistent hash function) of its own id concatenated with the load balancing step s, and perform the check only if this is lower than K/N (for some constant K known to all nodes).

It is possible that more than one leader is elected. If a node hears from more than one leader, it will use the multicast information from the leader with lowest t_1. Nodes that hear only from the other leader(s) will be equivalent to nodes that do not hear about the new ranges in the single leader case. Hence, multiple leaders is not a problem.

What happens when a leader L fails during load balancing? There are only two possibilities: (1) Leader L fails before it initiates the multicast so that no node knows about the new ranges: This case is equivalent to not initiating the current load balancing step. (2) Leader L initiates the multicast before it fails. In this case, $t_1 = \sqrt(N) + log(N)$ (instead of $log(N)$) is enough to guarantee that all nodes know about the new ranges with high probability in t_1 epochs: This case is therefore equivalent to the leader being alive during the entire load balancing step. Hence, a leader failing during the load balancing step has no effect on the correctness of the load balancing operation.

As we already observed, the leader is elected using a totally decentralized protocol and nodes take turns in assuming the role of a leader. Without going into the details, we would like to note that it is possible to totally decentralize our load balancing algorithm.

Balancing Query Load

So far we have seen how rKelips answers range queries efficiently and guarantees load balancing when the load L_p on a peer p is defined as the number of items stored at p.

The real measure of load is the number of queries answered by the node. This definitely depends on the number of items stored at the node but also depends a great deal on how the queries are routed. Like in other range indices [CLGS04, CLM$^+$04a, BRS04, GBGM04], we assume that number of queries per item is the same for all items.

In rKelips, number of items stored at a node is also a measure of the number of queries answered by the node if the queries to a group are uniformly distributed over all the nodes in the group. We will now describe how to ensure that queries to a group are uniformly distributed over all the nodes in the group. Note that a rKelips node can choose a contact for a group uniformly at random from all the nodes in the group. If the query rate is uniform over all nodes, then random contacts ensure that queries to a group are uniformly distributed over all the nodes in the group.

The question that remains is: even when most queries are issued by a small number of nodes in the system, how do we ensure that queries to a group are roughly uniformly distributed over all the nodes in the group. This can be achieved using the following scheme: In the response to a lookup query from node n to a group, its contact c for the group will include information about one of its group members (chosen at random). Node n may use this new contact to replace c. This will make sure that queries to a group from any node n are roughly uniformly distributed over all the possible contacts for that group.

5.2.7 Availability

A rKelips node has a lot of paths to reach a node in the target affinity group. rKelips is therefore highly available and can tolerate of lot of node failures. In particular, a node requires only three tries on average to reach a live node from a target affinity group even when half the nodes in the system fail.

Theorem 1 (Availability) *Suppose node n wants to route a given query q to a live node in Group g. Even when half the nodes in the system fail, n needs* 3 *tries on average to reach a node from Group g.*

Proof Sketch: As we use a hash function to assign nodes to groups, we assume that the node failures are not correlated. Let's also assume that node n has at least one contact (live or failed).

Node n will first forward the query to its contact for Group g. With probability 0.5 it contact is failed. Node n will then forward the query to nodes in its group and have them forward the query to their contacts for Group g. Therefore, expected number of tries $< T > \leq \frac{1}{2} + 2\frac{1}{2}\frac{1}{4} + 3\frac{1}{2}\frac{3}{4}\frac{1}{4} + \cdots$.

Hence, $< T > \leq 3$. Moreover, each try is at most 2 hops.

5.3 Additional Algorithms

In this section, we describe some additional algorithms. We first present our solution to the problem of rebalancing ranges. We then present an optimized algorithm to transfer items across groups.

5.3.1 Rebalancing Ranges

Setup: Consider a system with N nodes. Let $p_1, \cdots, p_N$ be the nodes in the system. We consider a relation divided into N range partitions on the basis of a key attribute, with partition boundaries at $R_0 \leq R_1 \leq \cdots R_N$. Node p_i manages the range $[R_{i-1}, R_i), \forall 0 < i \leq N$. We let $L(p_i)$ denote the load at p_i, defined to be the number of tuples stored by p_i. We assume a central site has access to the range-partition information $R_0 \leq R_1 \leq \cdots R_N$ and directs each query, insert and delete to the appropriate node(s). The goal of the central node is to perform load-balancing and in each round, the central node can communicate with the peers using $O(N)$ constant sized messages. The central node cannot, for example, find all the items currently in the system and then do a trivial load balancing.

Imbalance Ratio: A load-balancing algorithm guarantees an imbalance ratio σ if, after the completion of the load-balancing step, $max_i L(p_i) \leq \sigma\ min_i L(p_i) + c_0$, for some fixed constant c_0. Intuitively, σ is the asymptotic ratio between the largest and smallest loads.

Metric: We measure the cost of a load-balancing algorithm as the number of tuples that need to be moved between peers by the algorithm per insert or delete. Our interest is in the amortized cost per insert or delete, for adversarial (worst-case) sequences of insertions and deletions. The amortized cost of an insert or delete is said to be c if, for any sequence of t tuple inserts and deletes, the total number of tuples moved is at most tc.

Problem Statement: Develop a load balancing algorithm which guarantees a constant imbalance σ with low amortized cost per tuple insert and delete.

Rebalancing Ranges - Solution: Define *rank* of a tuple t, $rank(t)$, as the position in the sorted list of tuples (sorted on key attribute) currently in the system.

For any peer p_i, $0 < i \leq N$, the central node knows its range $[R_{i-1}, R_i)$ and can find its load $L(p_i)$. The central node then uses the algorithm proposed in [GBGM04] to do the load balancing by simulating N peers and simulating key values of tuples at these peers with the rank of the tuple. This gives the new partition boundaries in terms of the rank of the tuples. The central node then learns the key values corresponding to the new range boundaries. This can be done using $O(N)$ constant sized messages. This gives the central node the new range boundaries such that $max_i L(p_i) \leq \sigma\ min_i L(p_i) + c_0$ when the tuples are moved around to the right peer according to the new range boundaries. Using the guarantees provided by the load-balancing algorithm in [GBGM04], the amortized cost of insert or delete is constant.

Moreover, this load-balancing algorithm is asymptotically optimal since, for any load-balancing algorithm, there exist sequences of t operations that require $\Omega(t)$ tuple movements to ensure load balance.

5.3.2 Item Transfer

We implemented an optimized algorithm to transfer items from group i to group j quickly. Main features of the algorithm are as follows:

- Items are grouped into buckets of fixed size based on the available bandwidth in a gossip message.

- Nodes in group j pull different buckets from nodes in group i by piggybacking on gossip messages. This is done once in $\sqrt{N}$ rounds.

- These buckets are piggybacked on gossip messages within group j for the next $\sqrt{N}$ rounds (before the next set of buckets are pulled in).

- Nodes send bitmaps of size approximately $\sqrt{N}$ during the push phase of gossip. This is used to indicate which buckets the current node has and which it doesn't.

- Suppose node n picks node m (from its group) as the target for push-pull gossip. n will send a bitmap summarizing the buckets it has. Node m on receiving the gossip will send a bucket that it has and that node n is missing. Moreover, node m will also include a request for a bucket that it is missing and that is present with node n. Node n on receiving the gossip message from node m will store the new bucket received and will send the bucket requested by m.

Using this scheme, we observed in our experiments that transfer of M buckets of items to all nodes in the group takes at most $M + \sqrt{N}$ epochs. A naive push-pull gossip scheme to transfer M buckets of items to all nodes in the group will take $M\ log(M)$ epochs.

5.4 Experimental Results

We evaluated a real distributed implementation of rKelips. rKelips was implemented in C and was deployed on a cluster of 40 Linux machines. We present the results from this deployment in this section.

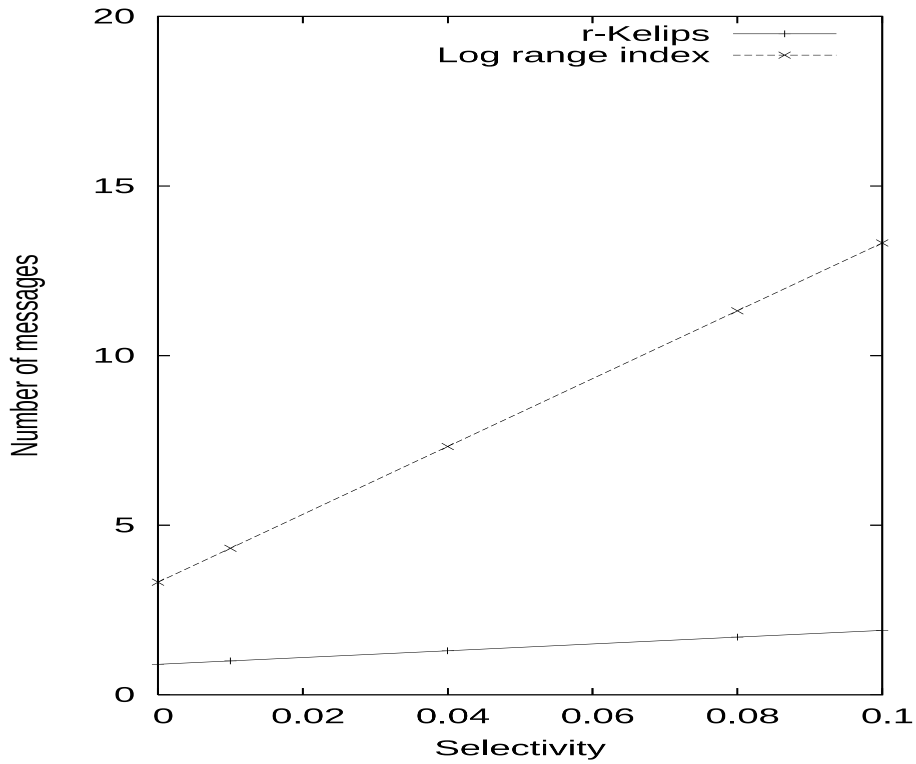


Figure 5.5: **Efficiency:** *rKelips system with 100 nodes needs only 2 hops for a range query with selectivity as high as 0.1*

Efficiency: In this section, we compare the efficiency of rKelips with that of logarithmic range index like P-Ring. In a system of 100 nodes, range queries with different selectivities are issued from nodes picked at random. Selectivity S of a range query is defined as the fraction of items in the system that are retrieved by the query. In this experiment, efficiency of a range index in answering a range query is measured using the average number of messages required to reach a set of nodes that contain items satisfying the query. Figure 5.5 shows that rKelips needs to contact only two nodes to answer a range query with selectivity as high as 0.1. On the other hand, a logarithmic range index likes P-Ring with the same number of nodes and items is expected to take 18 messages for such a range query.

Load Balance: In this section, we study the load balancing properties of rKelips.

In the first experiment, we study the time scale for a load balancing step. In Figure 5.6, we plot the variation of the storage load imbalance with time for a system of 100 nodes. Items are inserted at rate 1 item per node per second into Group 0 and no items are inserted into other groups. Figure 5.6 shows that the load balancing step takes only 38 gossip rounds or less than 8 seconds with a gossip period of 0.2 seconds. Moreover, the load imbalance before the load balancing step is about 1500 and the imbalance after the step is about 2.

We next show that the load balancing algorithm we use performs well. In Figure 5.7, we report a plot from Ganesan et al [GBGM04] showing that their load balancing algorithm guarantees a storage load imbalance of at most 4.2. They plot the storage load imbalance for the Fibbing Algorithm against a run on the ZIPFIAN workload [GBGM04].

Next, we study the effectiveness of our optimized item transfer algorithm. We

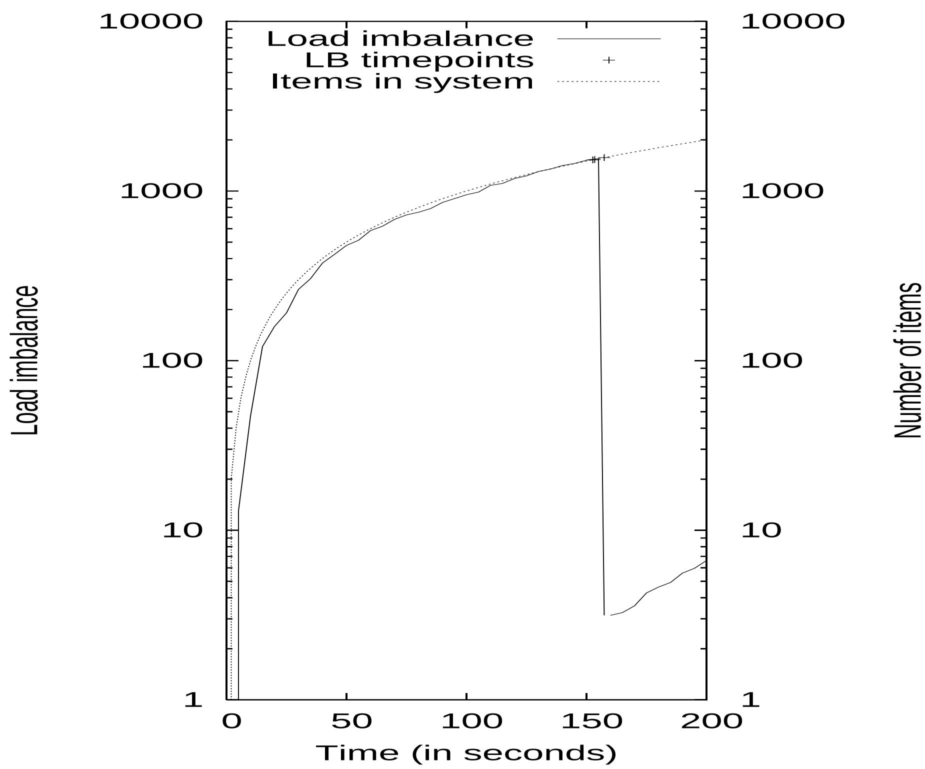


Figure 5.6: **Load imbalance:** *Periodic load balancing to bound load imbalance. Load balancing step takes less than 8 seconds with a gossip period of 0.2 seconds*

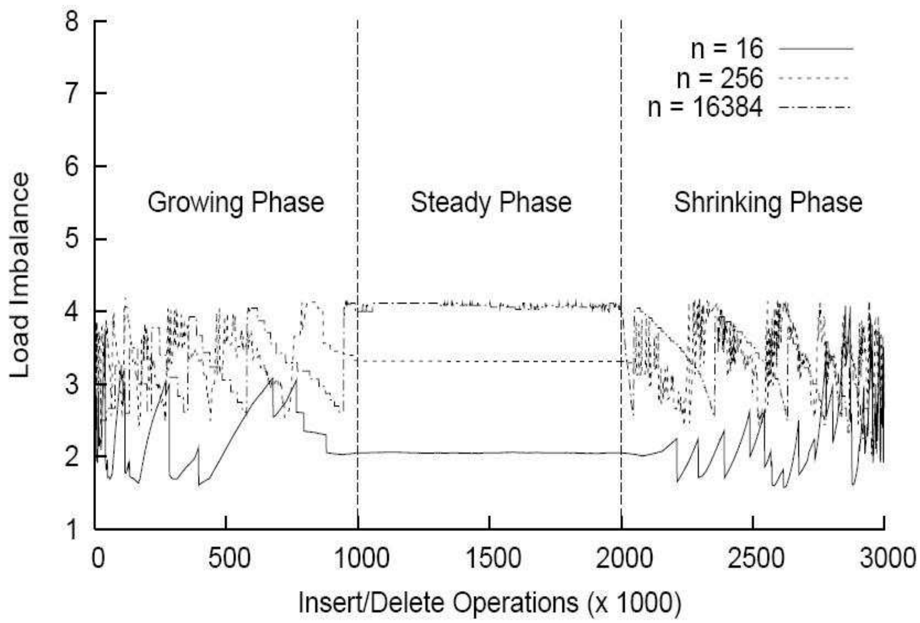


Figure 5.7: **Load imbalance:** *Load balancing algorithm by Ganesan et al guarantees a storage load imbalance of at most 4.2*

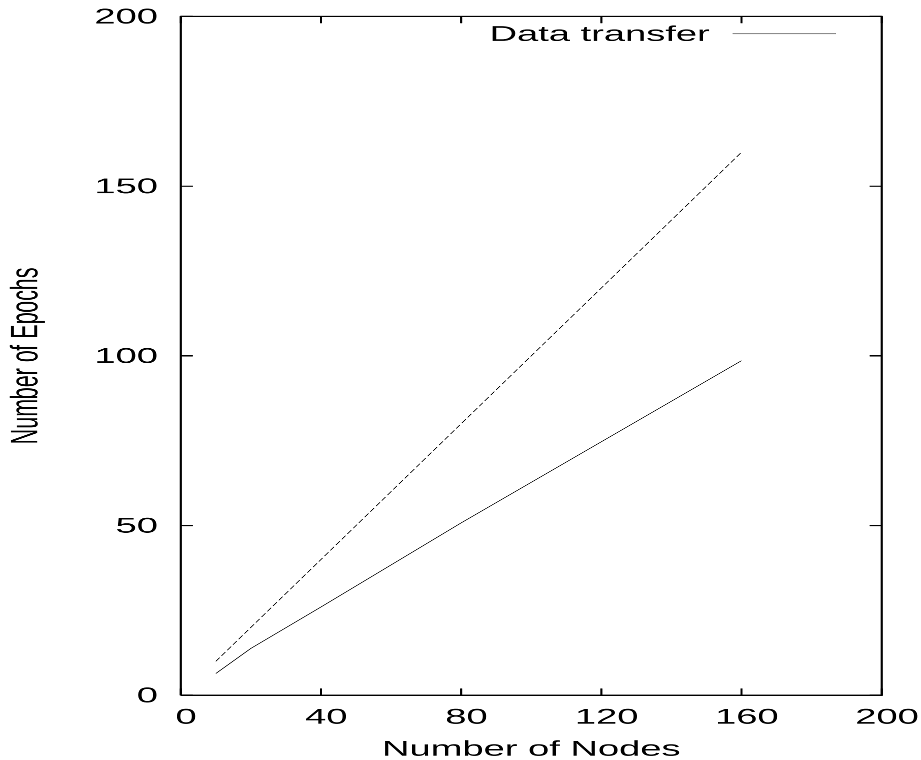


Figure 5.8: **Item transfer to a target group:** *Transfer rate as high as M items per epoch, where M is the number of items that fit in a gossip message*

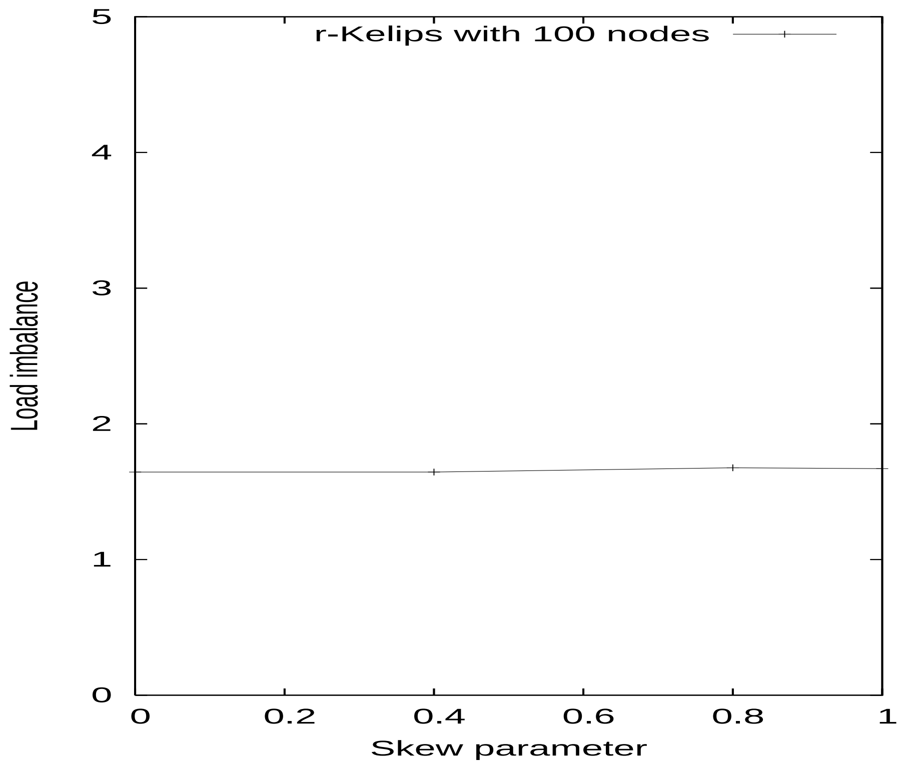


Figure 5.9: **Load Balance:** *Load imbalance is small (1.6) even when all the requests are issued by one node*

consider a system with N nodes and each node has a piece of information to start with. We run our protocol and measure the number of epochs required before all the nodes know about all the N pieces of information. Figure 5.8 shows that the number of epochs required is well below N. Therefore, if gossip bandwidth is such that there are M items per gossip message, we can perform item transfer at the rate of M items per epoch i.e. every node learns about M new items per epoch.

In our final plot in this section, we study query load balance in rKelips. In this plot, we assume that all items receive requests at the same rate. Requests can originate from different nodes at different rates. We capture this with a skew parameter s: $s = 0$ implies all the request originate at one node and $s = 1$

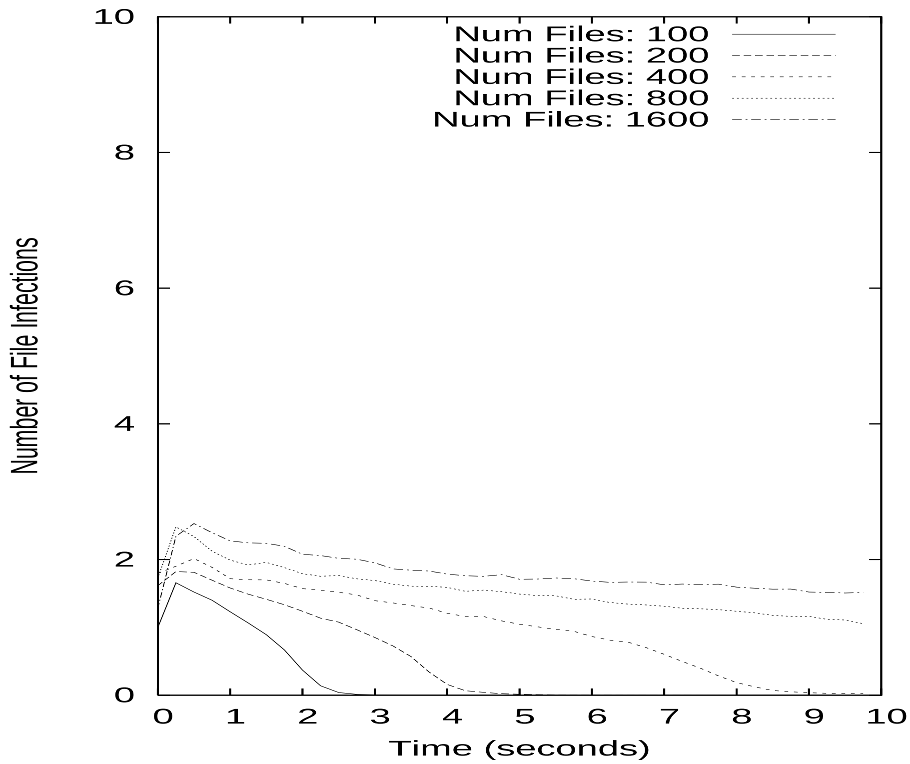


Figure 5.10: **Inserts:** *Insert rate of 40 items per sec can be sustained with a b/w of 22KBps*

implies all the requests originate at an equal rate from all the nodes in the system. Figure 5.9 shows the load imbalance as a function of skew s. We see that the load imbalance is close to 1.6 even when all the requests originate at one node. We also observed that, using the piggybacking of lookup responses with contact information, we obtain a load imbalance very close to 1.0 even when all the requests originate at one node (data not shown).

Maintenance Cost: In this section, we study the effectiveness of gossip in disseminating item information. In a 25 node rKelips affinity group, we insert M items at a node in the group. We fixed the gossip background bandwidth at 22 KBps allowing 10 items to be included in the gossip message (assuming

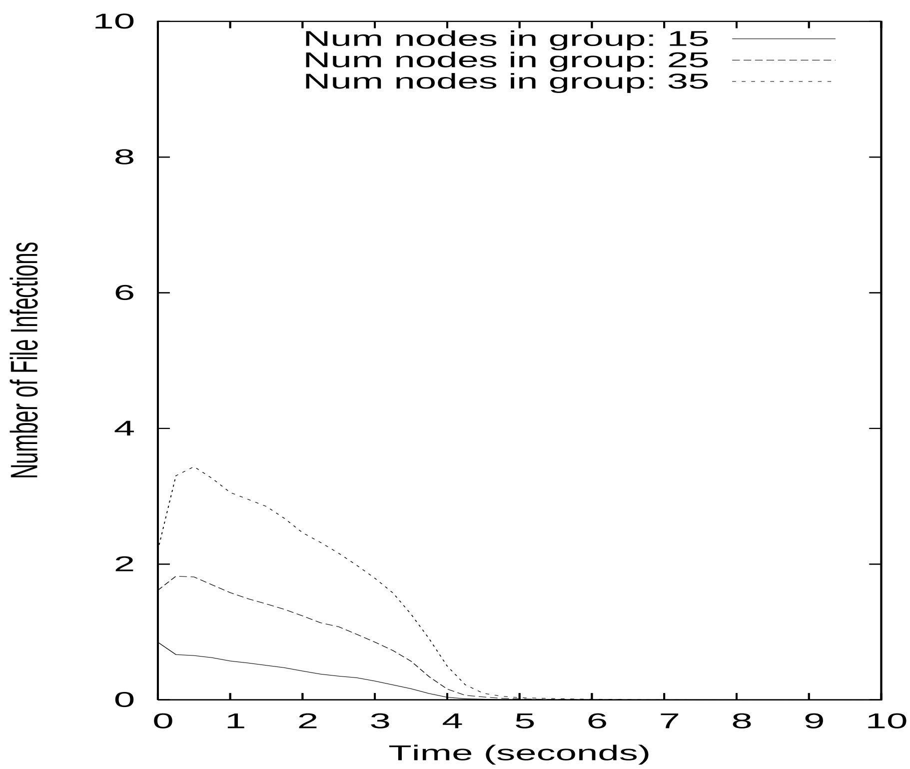


Figure 5.11: **Insert rate:** *For a fixed bandwidth, increase in the number of nodes causes a marginal decrease in the maximum insert rate that can be sustained*

$100B$ per item). We plot the average number of items per node as a function of time. Figure 5.10 shows that about 400 items can be inserted into the group in 10sec. We conducted an independent experiment to confirm that rKelips sustains an insert rate as high as 40 items per second with a background bandwidth as low as $22KBps$.

We next study the effectiveness of gossip in disseminating item information as a function of the number of nodes. We fix the number of items initially inserted into a node at 200 and measure the average number items per node as a function of the number of nodes. From Figure 5.11, we can deduce that increase in number of nodes from 15 to 35 does not cause a significant change in the insert rate that can be sustained with a bandwidth of 22KBps.

Availability: In this section, we study the performance of rKelips when half the nodes in the system are failed. We start with a system of N nodes and fail half the nodes in the system that are picked uniformly at random. We measure the average number of messages required to reach an alive node in a particular target group. In this experiment, each node has at most one contact per affinity group. From Figure 5.12 we see that number of messages to reach a live node (and hence answer a range query) is independent of the number of nodes, even when half the nodes in the system fail.

5.5 Related Work

There has been a lot of work on indexing in distributed databases [LNS93, LNS94, KW94, Lom96]. Many of the indexing techniques [LNS93, LNS94] developed in the distributed databases community maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability

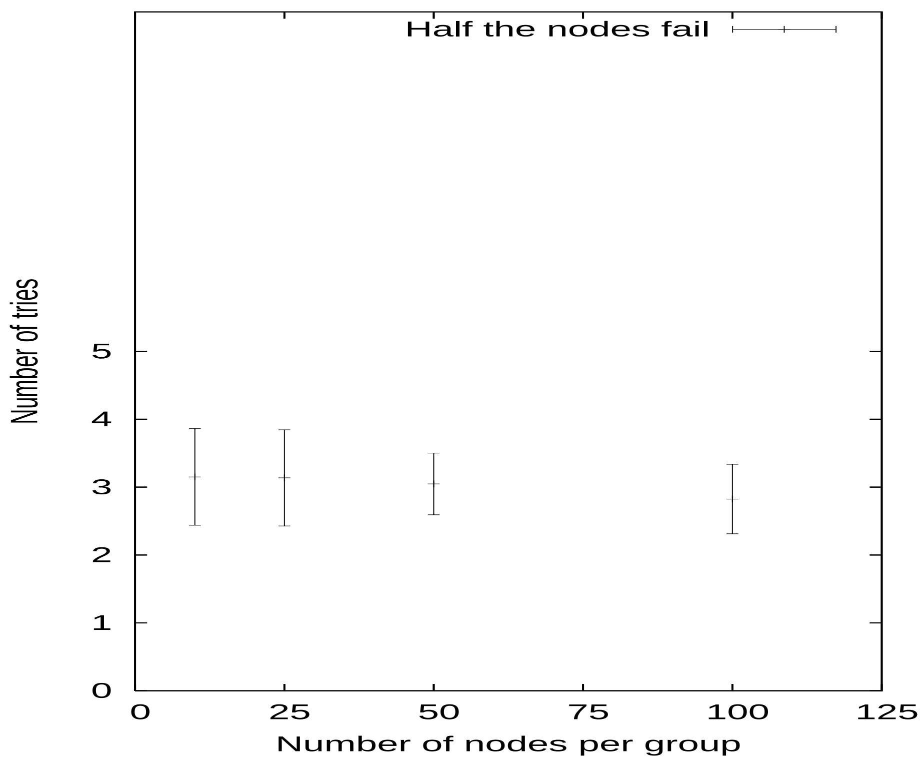


Figure 5.12: **Availability:** *Need only 3 tries to reach a live node even when half the nodes fail*

problems when dealing with thousands of peers. In contrast, rKelips is designed to be resilient to failures of arbitrary peers. The DRT [KW94] and dPi-tree [Lom96] maintain replicas lazily, but these schemes are not designed for peers that can leave the system, which makes them inadequate in a P2P environment.

Chord [SMK+01], Pastry [RD01b], Tapestry [ZKJ01], CAN [RFH+01] and Kelips [GBL+03] implement distributed hash tables to provide efficient lookup of a given key value. However, these structures cannot process range queries efficiently because a hash function destroys the ordering in the key value space. rKelips is similar in design to and uses ideas from Kelips [GBL+03].

There have been a number of approaches to answer range queries in a p2p setting [GAE03a, AS03, CLGS04, GBGM04, BRS04, CLM+04a, Abe01, JOV05]. All these indices give logarithmic guarantee for search in a stable system. They do not provide search guarantees in the presence of peer failures. On the other hand, rKelips guarantees answering equality queries and low selectivity range queries in just one hop. Moreover, even when half the nodes in the system fail, rKelips takes only 6 messages (3 tries) on average to reach a node containing a subset of the results to a given range query.

5.6 Conclusions and Future Work

We have introduced rKelips, a novel p2p index structure that efficiently supports both equality and range queries in a dynamic p2p environment. To the best of our knowledge, this is the first p2p range index that guarantees answering equality queries and low selectivity range queries in just one hop. Moreover, rKelips handles churn well and is the only p2p range index that guarantees good search performance even in the presence of high churn. It effectively balances items among peers even in the presence of skewed data insertions and deletions. Even when half the nodes in the system fail, rKelips takes only 6 messages (3 tries) on average to reach a node containing a subset of the results to the range query. Our experiments confirm that rKelips outperforms all existing range indices in terms of performance, while keeping the maintenance costs low.

As part of future work, we want to extend rKelips to handle multi-dimensional range queries efficiently. We also want to adapt load balancing techniques in rKelips to deal with non-uniform popularity of items.

Chapter 6

Correctness and Availability in P2P Range Indices

6.1 Introduction

Different applications have different requirements for a P2P index. We can characterize the index requirements of most P2P applications along the following three axes (we shall formally define these requirements later in the chapter):

- **Expressiveness of predicates**: whether simple equality predicates suffice in a P2P index, or whether more complex predicates such as range predicates are required.
- **Query correctness**: whether it is crucial that the P2P index return all and only the data items that satisfy the predicate.
- **System and Item Availability**: whether it is crucial that the availability of the P2P index and the items stored in the index, are not reduced due to the reorganization of peers.

For example, simple file sharing applications only require support for equality predicates (to lookup a file by name), and do not have strict correctness and availability requirements (it is not catastrophic if a search occasionally misses a file, or if files are occasionally lost). Internet storage applications require only simple equality predicates, but have strict requirements on correctness and availability (so that data is not missed or lost). Digital library applications require complex predicates such as range predicates (to search for articles within a date range), but do not have strict correctness and availability requirements. The most demanding applications are transaction processing and military applications, which require both complex range predicates (to search for objects within a region) and strong correctness/availability guarantees.

As an example, consider the Joint Battlespace Infosphere (JBI) [URLb], a military application with the requirement to be highly scalable and fault-tolerant. One of the potential uses of the JBI is to track information objects, which could include objects in the field such as enemy vehicles. A natural way to achieve the desired scalability and fault-tolerance is to store such objects as (value,item) pairs in a P2P index, where the value could represent the geographic location of the object (in terms of its latitude and longitude), and the item could be a description of that object. Clearly, the JBI requires support for range queries in order to find objects in a certain region. The JBI also requires strong correctness guarantees (so that objects are not missed by a query) and availability guarantees (so that stored objects are not lost).

Current P2P indices, however, cannot satisfy the above application needs: while there has been some work on devising P2P indices that can handle expressive range predicates [BRS04, CLM+04a, GBGM04], there has been little or no work on guaranteeing correctness and availability in such indices. Specifically, we are not aware of any P2P range index that *guarantees* that a query will not miss items relevant to a query. In fact, we shall later show scenarios whereby range indices [BRS04, CLM+04a, GBGM04] that are based on the Chord ring [SMK+01] (originally devised for equality queries) can miss query results for range queries, even when the index is operational. Similarly, we are not aware of any range index that can provide provable deterministic guarantees on system and item availability. rKelips (see Chapter 5) only provides probabilistic guarantees for correctness and availability.

In this chapter, we devise techniques that can provably guarantee query correctness and system and item availability in P2P range indices. At a high level, there are two approaches for guaranteeing correctness and availability. The first approach is to simply let the application handle the correctness and availability issues – this, for instance, is the approach taken by CFS [DKK+01] and PAST [RD01a], which are applications built on top of the P2P equality indices Chord [SMK+01] and Pastry [RD01b], respectively. However, this approach does not work in general for range indices because the application does not (and should not!) have control over various concurrent operations in a P2P range index, including index reorganization and peer failures. Moreover, this approach exposes low-level concurrency details to applications and is also very error-prone due to subtle concurrent interactions between system components.

We thus take the alternative approach of developing new correctness and availability primitives that can be directly implemented in a P2P index. Specifically, we build upon the P2P indexing framework proposed by Crainiceanu et al. [CLM+04b], and embed novel techniques for ensuring correctness and availability directly into this framework. The benefits of this approach are that it abstracts away the dynamics of the underlying P2P system and provides applications with a consistent interface with provable correctness and availability guarantees. To the best of our knowledge, this is the first attempt to address these issues for both equality and range queries in a P2P index.

One of the benefits of implementing our primitives in the context of a P2P indexing framework is that our techniques are not just applicable to one specific P2P index, but are applicable to all P2P indices that can be instantiated in the framework, including [BRS04, CLM+04a, GBGM04]. As a specific instantiation, we implement P-Ring [CLM+04a], a P2P index that can support both equality and range queries, and show how it can be extended to provide correctness and availability guarantees. We also quantitatively demonstrate the feasibility of our proposed techniques using a real distributed implementation of P-Ring.

The rest of the chapter is organized as follows. In Section 6.2, we present some background material, and in Section 6.3, we outline our correctness and availability goals. In section 6.4 we present techniques for guaranteeing query correctness, and in Section 6.5, we present techniques for guaranteeing system and item availability. In Section 6.6, we present our experimental results. In section 6.7, we discuss related work, and in Section 6.8, we present our conclusions.

6.2 Background

In this section, we first introduce our system model, and then we briefly review the indexing framework proposed by Crainiceanu et al.[CLM+04b], and give an example instantiation of this framework for completeness. We use this instantiation in the rest of the chapter to discuss problems with existing approaches and to illustrate our newly proposed techniques. We use the framework because it presents a clean way to abstract out different components of a P2P index, and it allows us to confine concurrency and consistency problems to individual components of the framework.

6.2.1 System Model

A *peer* is a processor with shared storage space and private storage space. The shared space is used to store the distributed data structure for speeding up the evaluation of user queries. We assume that each peer can be identified by a physical id (for example, its IP address). We also assume a crash-failure model for peer failures. A *P2P system* is a collection of peers. We assume there is some underlying network protocol that can be used to send messages reliably from one peer to another with known bounded delay. A peer can join a P2P system by contacting some peer that is already part of the system. A peer can leave the system at any time without contacting any other peer. For ease of exposition in this chapter, we assume a globally synchronized clock at the different peers, although our correctness proofs (and our implementation) do not require a globally synchronized clock. Our formal proofs of correctness for all theorems are based on histories of actions on objects. [1]

We assume that each (data) item stored in a peer exposes a *search key value* from a totally ordered domain $\mathcal{K}$ that is indexed by the system. The search key value for an item i is denoted by $i.skv$. Without loss of generality, we assume that search key values are unique (duplicate values can be made unique by appending the physical id of the peer where the value originates and a version number; this transformation is transparent to users). Peers inserting items into the system can retain ownership of their items. In this case, the items are stored in the private storage partition of the peer, and only pointers to the items are inserted into the system. In the rest of the chapter we make no distinction between items and pointers to items.

[1] Due to space constraints, we had to omit all proofs from this chapter.

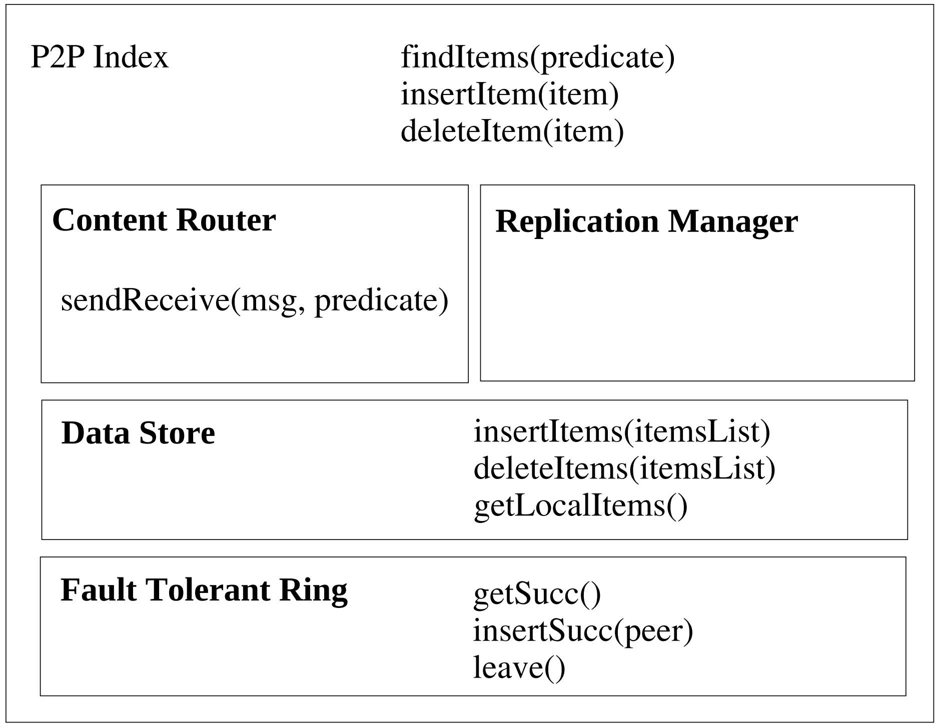


Figure 6.1: **Indexing Framework**

The queries we are considering are range queries of the form $[lb, ub]$, $(lb, ub]$, $[lb, ub)$ or (lb, ub) where $lb, ub \in \mathcal{K}$. Queries can be issued at any peer in the system.

6.2.2 The P2P Indexing Framework From [5]

A P2P index needs to reliably support the following operations: search, item insertion, item deletion, peers joining, and peers leaving the system. We now shortly survey the modularized indexing framework, which is designed to capture most structured P2P indices. Figure 6.1 shows the components of the framework, and their APIs. Note that the architecture does not specify *implementations* for these components but only specifies *functional requirements*.

Fault Tolerant Torus. The Fault Tolerant Torus connects the peers in the system on a torus, and provides reliable connectivity among these peers even in the face of peer failures. For the purposes of this chapter, we focus on a Fault Tolerant Ring (a one-dimensional torus). On a ring, for a peer p, we can define the *successor* `succ`(p) (respectively, *predecessor* `pred`(p)) to be the peer adjacent to p in a clockwise (resp., counter-clockwise) traversal of the ring. Figure 6.2 shows an example of a Fault Tolerant Ring. If peer p_1 fails, then the ring will reorganize such that $\texttt{succ}(p_5) = p_2$, so the peers remain connected. In addition to maintaining successors, each peer p in the ring is associated with a value, $p.val$, from a totally ordered domain $\mathcal{PV}$. This value determines the position of a peer in the ring, and thus increases clockwise around the ring (wrapping around at the highest value). The values of the peers in Figure 6.2 are shown in brackets.

Figure 6.1 shows the Fault Tolerant Ring API. When invoked on a peer p,
p.`getSuccessor` returns the address of `succ`(p).
p.`insertSuccessor`(p') makes p' the successor of p.
p.`leaveRing` allows p to gracefully leave the ring (of course, p can leave the ring without making this call due to a failure). The ring also exposes events that can be caught at higher layers, such as successor changes (not shown).

Data Store. The Data Store is responsible for distributing and storing items at peers. The Data Store has a map $\mathcal{M}$ that maps the search key value $i.skv$ of each item i to a value in the domain $\mathcal{PV}$ (the domain of peer values). An item i is stored in a peer p such that $\mathcal{M}(i.skv) \in (\texttt{pred}(p).val, p.val]$. In other words, each peer p is responsible for storing data items mapped to a value between $\texttt{pred}(p).val$ and $p.val$. We refer to the range $(\texttt{pred}(p).val, p.val]$ as $p.range$. Figure 6.3 shows an example Data Store that maps some search key values to peers on the ring. For example, peer p_3 is responsible for search key values 16 and 18. One of the main re-

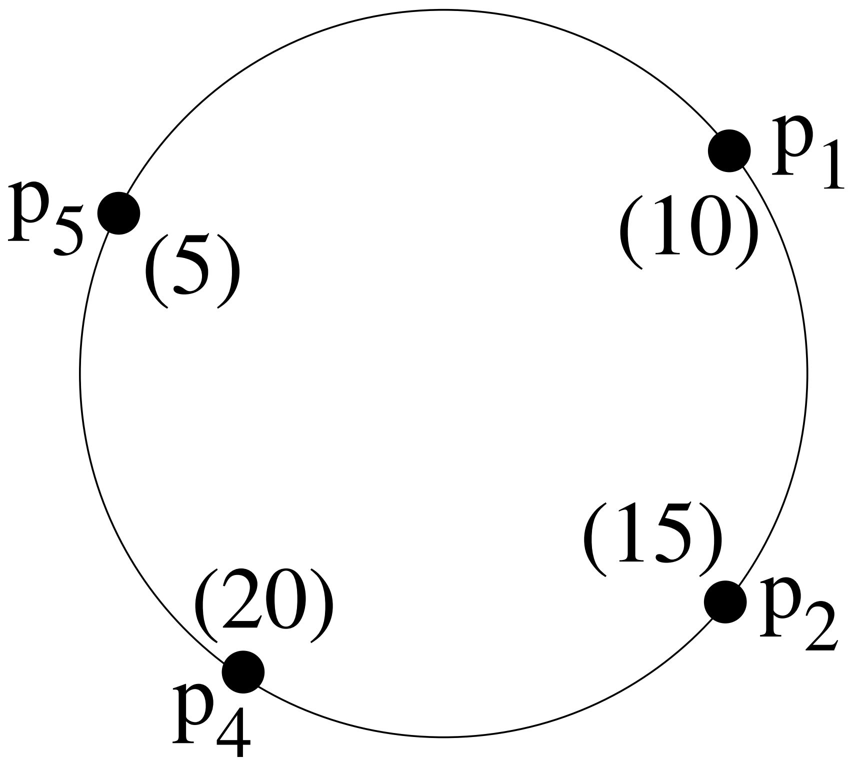


Figure 6.2: **Ring**

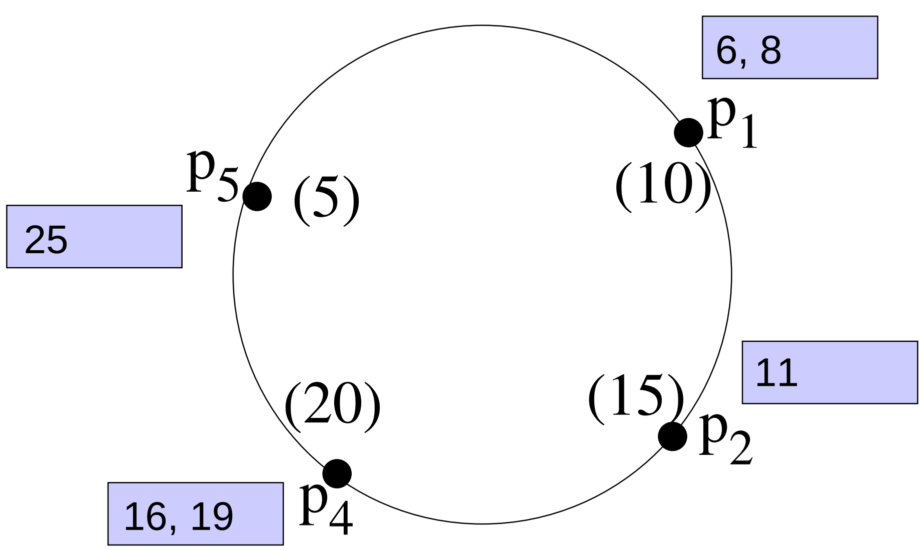


Figure 6.3: **Data Store**

sponsibilities of the Data Store is to ensure that the data distribution is uniform so that each peer stores about the same number of items. Different P2P indices have different implementations for the Data Store (e.g., based on hashing [SMK+01], splitting, merging and/or redistributing [CLM+04a, GBGM04]) for achieving this storage balance. As shown in Figure 6.1, the Data Store provides API methods to insert items into and delete items from the system.

Replication Manager. The Replication Manager is responsible for reliably storing items in the system even in the presence of failures, until items are explicitly deleted. As an example, in Figure 6.5, peer p_1 stores items i_1 and i_2 such that $\mathcal{M}(i_1.skv) = 8$ and $\mathcal{M}(i_2.skv) = 9$. If p_1 fails, then these items would be lost even though the ring would reconnect after the failure. The goal of the replication manager is to handle such failures for example by replicating items so that they can be "revived" even if peers fail.

Content Router. The Content Router is responsible for efficiently routing messages to relevant peers in the P2P system. As shown in the API (see Figure 6.1), the relevant peers are specified by a content-based predicate on search key values, and not by the physical peer ids. This abstracts away the details of storage and index reorganization from higher level applications.

P2P Index. The P2P Index is the index exposed to the end user. It supports search functionality by using the functionality of the Content Router, and supports item insertion and deletion by using the functionality of the Data Store.

6.2.3 An Example Instantiation

We now discuss the instantiation of the architecture with range query indices, taking as a specific example P-Ring [CLM+04a], an index structure designed for range queries in P2P systems. P-Ring, uses the Fault Tolerant Ring of Chord and the Replication Manager of CFS, and only devises a new Data Store and a Content Router for handling data skew in range queries. While the full details of P-Ring are presented in [CLM+04a], we concentrate only on features of P-Ring that are common to *all* P2P range query index structures from the literature: splitting, merging, and redistributing in order to balance the number of items at each peer [BRS04, GBGM04]. In the remainder of this section, we will discuss P-Ring in the context of the P2P Indexing Architecture. However, we would like to emphasize that we only use P-Ring as a running example to illustrate query correctness, concurrency, and availability issues in subsequent sections, and our discussion applies to all other P2P range indices from the literature.

Fault Tolerant Ring. P2P range indices need to maintain connectivity, and most use the Chord Ring to achieve this [SMK+01]. The Chord Ring achieves fault-tolerance by storing a *list* of successors at each peer, instead of storing just a single successor. Thus, even if the successor of a peer p fails, p can use its successor list to identify other peers to re-connect the ring and to maintain connectivity. Figure 6.4 shows an example Chord Ring in which successor lists are of length 2 (i.e., each peer p stores $\texttt{succ}(p)$ and $\texttt{succ}(\texttt{succ}(p))$ in its successor list). The successor lists are shown in the boxes next to the associated peers. Chord also provides a way to maintain these successor lists in the presence of failures by periodically *stabilizing* a peer p with its first live successor in the successor list. P-Ring also uses Chord to maintain connectivity.

Data Store. Ideally, we would like data items to be uniformly distributed among peers so that the storage load of each peer is about the same. Most existing P2P indices achieve this goal by *hashing* the search key value of an item, and

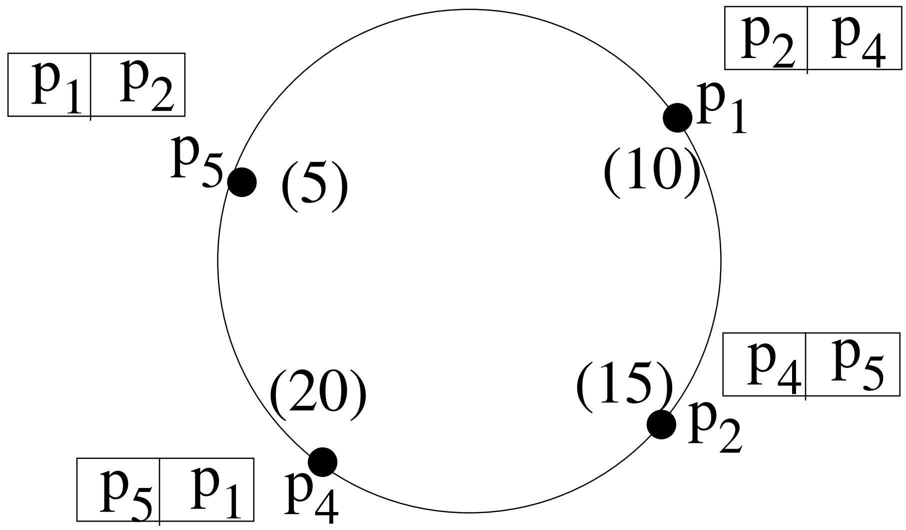


Figure 6.4: **Chord Ring**

assigning the item to a peer based on this hashed value. Such an assignment is, with high probability, very close to a uniform distribution of entries [RFH+01, RD01b, SMK+01]. However, hashing destroys the value ordering among the search key values, and thus cannot be used to process range queries efficiently (for the same reason that hash indices cannot be used to handle range queries efficiently).

To solve this problem, range indices assign data items to peers directly based on their search key value (i.e., the map $\mathcal{M}$ is order-preserving, in the simplest case it is the identity function). In this case, the ordering of peer values is the same as the ordering of search key values, and range queries can be answered by scanning along the ring. The problem is that now, even in a stable P2P system with no peers joining or leaving, some peers might become overloaded or underloaded due to skewed item insertions and/or deletions. There is a need for a way to dynamically reassign and maintain the ranges associated to the peers. Range indices achieve this goal by `splitting`, `merging` and `redistributing` for handling item overflows

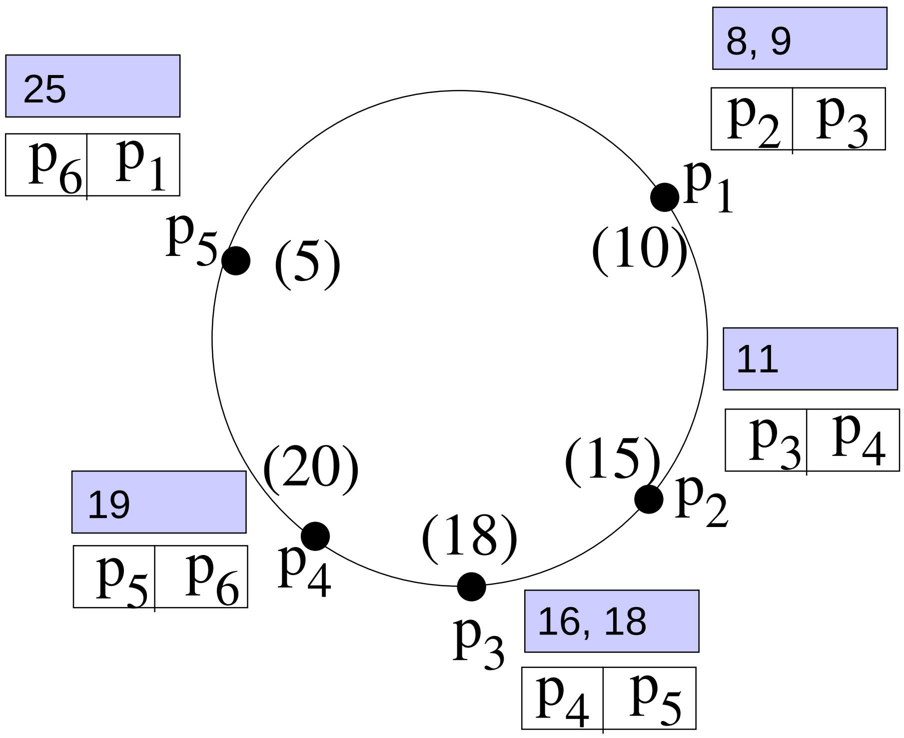


Figure 6.5: **P-Ring Data Store**

and underflows in peers. Let us give an example in the context of P-Ring.

The P-Ring Data Store has two types of peers: *live* peers and *free* peers. Live peers are part of the ring and store data items, while free peers are maintained separately in the system and do not store any data items.[2] The Data Store ensures that the number of items stored in each live peer is between `sf` and $2 \cdot \texttt{sf}$ in order to balance storage between peers.

Whenever the number of items in a peer p's Data Store becomes larger than $2 \cdot \texttt{sf}$ (due to many insertions into $p.range$), it is said that an *overflow* occurred. In this case, p tries to *split* its assigned range (and implicitly its items) with a free peer, and to give a fraction of its items to the new peer. Whenever the number of entries in p's Data Store becomes smaller than $storageFactor$ (due to deletions from $p.range$), it is said that an *underflow* occurred. In this case, p tries to *merge* with its successor in the ring to obtain more entries. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a free peer.

As an illustration of a split, consider the Data Store shown in Figure 6.3. Assume that `sf` is 1, so each peer can have 1 or 2 entries. Now, when an item i such that $i.skv = 18$ is inserted into the system, it will be stored in p_4, leading to an overflow. Thus, $p_4.range$ will be split with a free peer, and p_4's items will be redistributed accordingly. Figure 6.5 shows the Data Store after the split, where p_4 split with the free peer p_3, and p_3 takes over part of the items p_4 was originally responsible for (the successor pointers in the Chord Ring are also shown in the figure for completeness). As an illustration of merge, consider again Figure 6.5

[2] In the actual P-Ring Data Store, free peers also store data items temporarily for some live peers. The ratio of the number of items between any two peers can be bounded, but these details are not relevant in the current context.

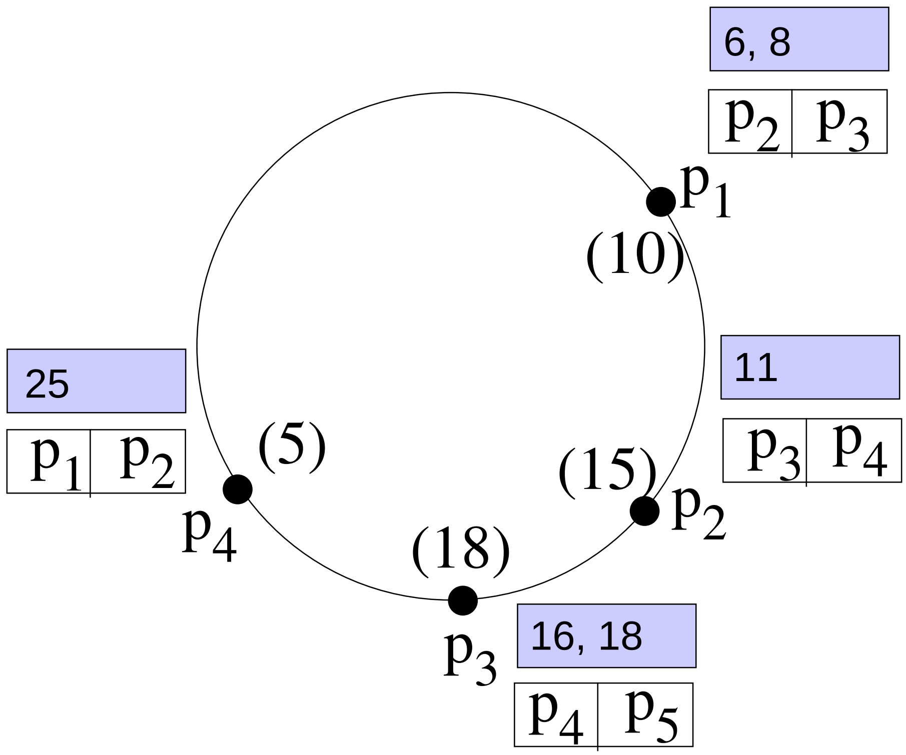


Figure 6.6: **Data Store Merge**

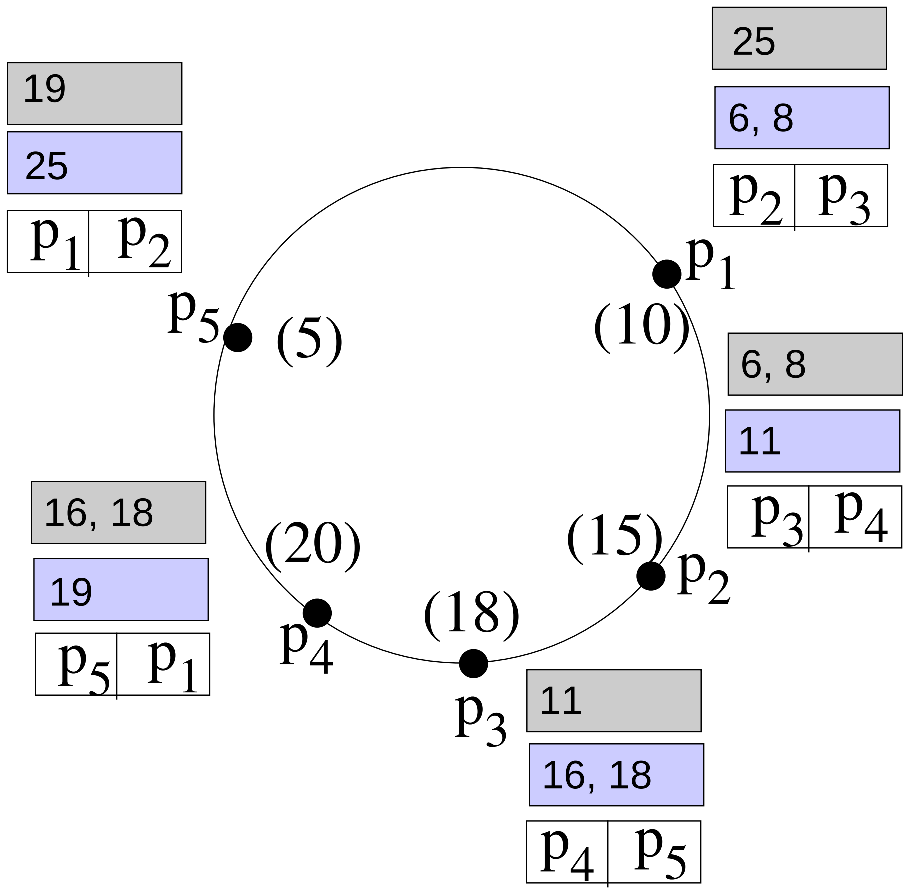


Figure 6.7: **CFS Replication**

and assume that item i with $t.skv = 19$ is deleted from the system. In this case, there is an underflow at p_4, and p_4 merges with its successor, p_5 and takes over all of p_5's items; p_5 in turn becomes a free peer. Figure 6.6 shows the resulting system.

Replication Manager. Most index structures use the CFS Replication Manager, and so does P-Ring. CFS Replication works as follows. Consider an item i stored in the Data Store at peer p. The Replication Manager replicates i to k successors of p. In this way, even if p fails, i can be recovered from one of the successors of i. Larger values of k offer better fault-tolerance but have additional overhead. Figure 6.7 shows a system in which items are replicated with a value of $k = 1$ (the replicated values are shown in the top most box next to the peer).

Content Router. The P-Ring Content Router is based on idea of constructing a hierarchy of rings that can index skewed data distributions. The details of the content router are not relevant for this chapter; see [CLM+04a] for details.

6.3 Goals

We now turn to the main focus of this chapter: guaranteeing correctness and availability in P2P range indices. At a high level, our techniques enforce the following design goals.[3]

- **Query Correctness:** A query issued to the index should return all and only those items in the index that satisfy the query predicate.
- **System Availability:** The availability of the index should not be reduced due to index maintenance operations (such as splits, merges, and redistribu-

[3]We give formal definitions of correctness and availability in Sections 6.4 and 6.5.

tions).

- **Item Availability:** The availability of items in the index should not be reduced due to index maintenance operations (such as splits, merges, and redistributions).

While the above requirements are simple and natural, it is surprisingly hard to satisfy them in a P2P system. Thus, one approach is to simply leave these issues to higher level applications – this is the approach taken by CFS [DKK$^+$01] and PAST [RD01a], which are applications built on top of Chord and Pastry, respectively, two index structures designed for equality queries. The downside of this approach is that it becomes quite complicated for application developers because they have to understand the details of how lower layers are implemented, such as how ring stabilization is done. Further, this approach is also error-prone because complex concurrent interactions between the different layers (which we illustrate in Section 6.4) make it difficult to devise a system that produces consistent query results. Finally, even if application developers are willing to take responsibility for the above properties, there are no known techniques for ensuring the above requirements for P2P range indices.

In contrast, the approach we take is to cleanly encapsulate the concurrency and consistency aspects in the different layers of the system. Specifically, we embed some consistency primitives in the Fault Tolerant Ring and the Data Store, and provide handles to these primitives for the higher layers. With this encapsulation, higher layers and applications can simply use these APIs without having to explicitly deal with low-level concurrency issues or knowing how lower layers are implemented, while still being guaranteed query consistency and availability for range queries.

We note that our proposed techniques differ from distributed database techniques [Kos00] in terms of scale (hundreds to thousands of peers, as opposed to a few distributed database sites), failures (peers can fail at any time, which implies that blocking concurrency protocols cannot be used), and perhaps most importantly, dynamics (due to unpredictable peer insertions and deletions, the location of the items themselves are not known a priori and can change *during* query processing).

In the subsequent two sections, we describe our solutions to query correctness and system and item availability. As in the instantiation of the framework by Crainiceanu et al. , we use P-Ring as our running example and illustrate the issues and solutions in the context of the Chord Fault Tolerant Ring, the P-Ring Data Store, and the CFS Replication Manager. We note, however, that our solutions are generally applicable to P2P range indices, and they can be implemented in any instantiation of the framework that we are building upon.

6.4 Query Correctness

We focus on query consistency for range queries (note that equality queries are a special case of range queries). We first formally define what we mean by query correctness in the context of the indexing framework. We then illustrate scenarios where query correctness can be violated if we directly use existing techniques. Finally, we present our solutions to these problems.

6.4.1 Defining Correct Query Results

Intuitively, a system returns a correct result for a query Q iff the result contains all and only those items in the system that satisfy the query predicate. Translating this intuition into a formal statement in a P2P system requires us to define which items are "in the system"; this is more complex than in a centralized system because peers can fail, can join, and items can move between peers during the duration of a query. We start by defining an index I as a set of peers $I = \{p_1, \ldots, p_n\}$, where each peer is structured according to the framework described in Section 6.2.2. In particular, each peer p has a Data Store which contains the set of items that are stored at p. To capture what it means for an item to be in the system, we first introduce the notion of a live item at a given time instant.

Definition (Live Item): An item i is *live* at time t in index I, denoted by $live_I(i, t)$ iff there exists a peer p in I at time t such that p's Data Store contains item i at time t and $\mathcal{M}(i.skv) \in range_{p,t}$, where $range_{p,t}$ is the value of $p.range$ at time t.

In other words, an item i is live at time t iff some peer with the appropriate range contains i in its Data Store at time t. Given the notion of a live item, we can define a correct query result as follows. We use $satisfies_Q(i)$ to denote whether item i satisfies query Q's query predicate.

Definition (Correct Query Result): A set R of items is a *correct query result* for a query Q initiated at time t_{begin} and successfully completed at time t_{end} in index I iff the following two conditions hold:

1. $\forall i \in R(satisfies_Q(i) \wedge \exists t(t_{begin} \leq t \leq t_{end} \wedge live_I(i, t)))$
2. $\forall i(satisfies_Q(i) \wedge \forall t(t_{begin} \leq t \leq t_{end} \Rightarrow live_I(i, t))) \Rightarrow i \in R$.

The first condition states that only items that satisfy the query predicate and which were live at some time during the query evaluation should be in the query result. The second condition states that all items that satisfy the query predicate and which were live throughout the query execution must be in the query result.

6.4.2 Incorrect Query Results: Scenarios

Existing index structures for range queries evaluate a range query in the following two steps: (a) finding the peer responsible for left end of the query range, and (b) scanning along the ring to retrieve the items in the range. The first step is achieved using an appropriate Content Router, such as SkipGraphs [AS03] or the P-Ring [CLM$^+$04a] Content Router, and the related concurrency issues have been described and solved elsewhere in the literature [AS03, CLM$^+$04a]. We thus focus on the second step and show how using existing techniques can produce incorrect results.

Scanning along the ring can produce incorrect query results due to two reasons. First, the ring itself can be temporarily inconsistent, thereby skipping over some live items. Second, even if the ring is consistent, concurrency issues in the Data Store can sometimes result in incorrect results. We now illustrate both of these cases using examples.

Inconsistent Ring

Consider the Ring and Data Store shown in Figure 6.5. Assume that item i with $\mathcal{M}(i.skv) = 6$ is inserted into the system. Since $p_1.range = (5, 10]$, i will be stored in p_1's Data Store. Now assume that p_1's Data Store overflows due to this insertion, and hence p_1 splits with a new peer p and transfers some of its items to p. The new state of the Ring and Data Store is shown in Figure 6.8. At this point, $p.range = (5, 6]$ and $p_1.range = (6, 10]$. Also, while p_5's successor list is updated to reflect the presence of p, the successor list of p_4 is not yet updated because the Chord ring stabilization proceeds in rounds, and p_4 will only find out about p when it next stabilizes with its successor (p_5) in the ring.

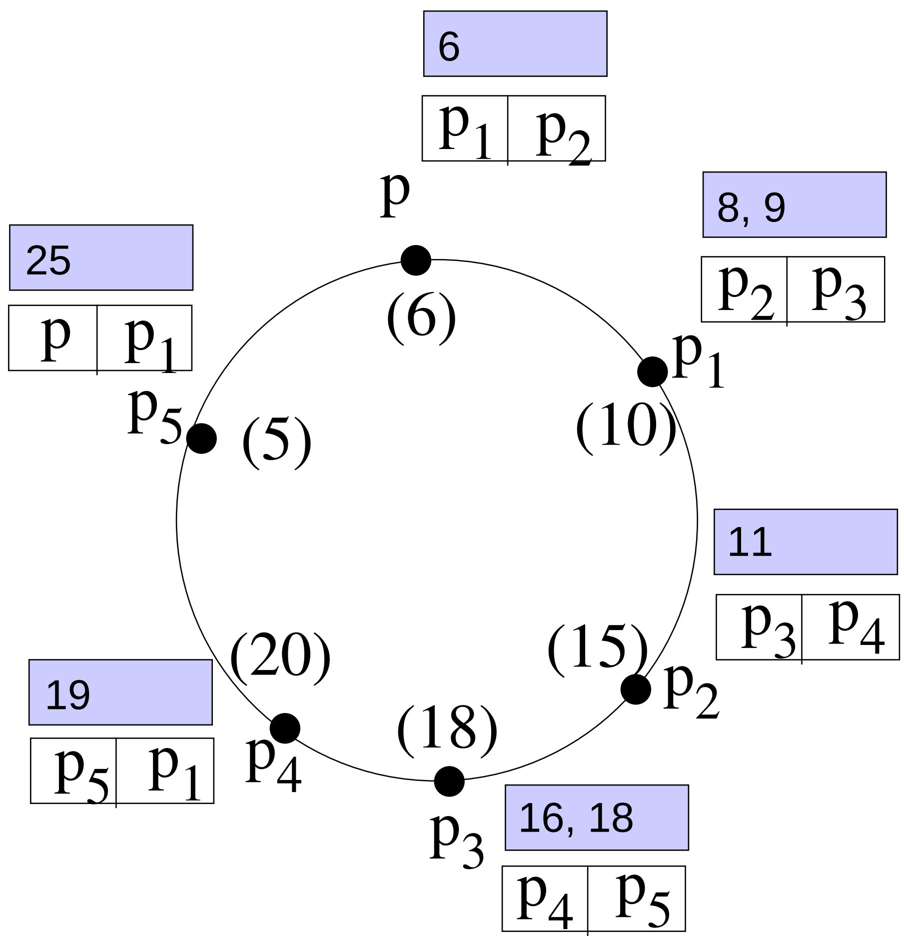


Figure 6.8: **After Insert**: *Peer p just inserted into the system*

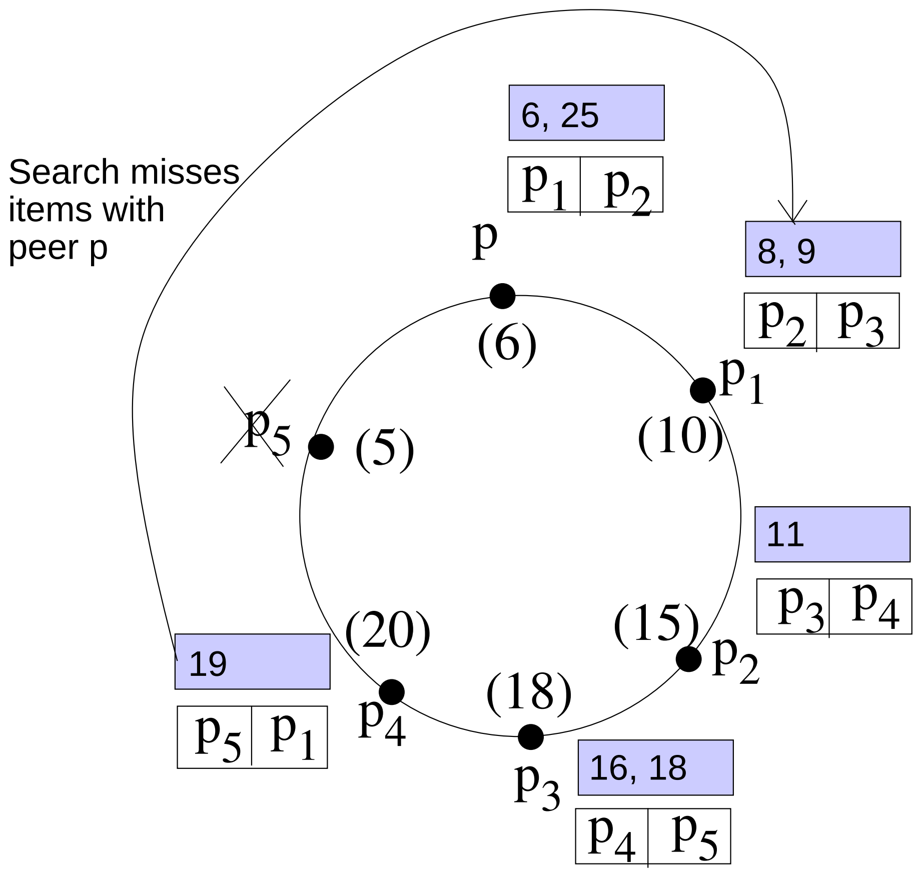


Figure 6.9: **Incorrect query results**: *Search Q originating at peer p_4 misses items in p*

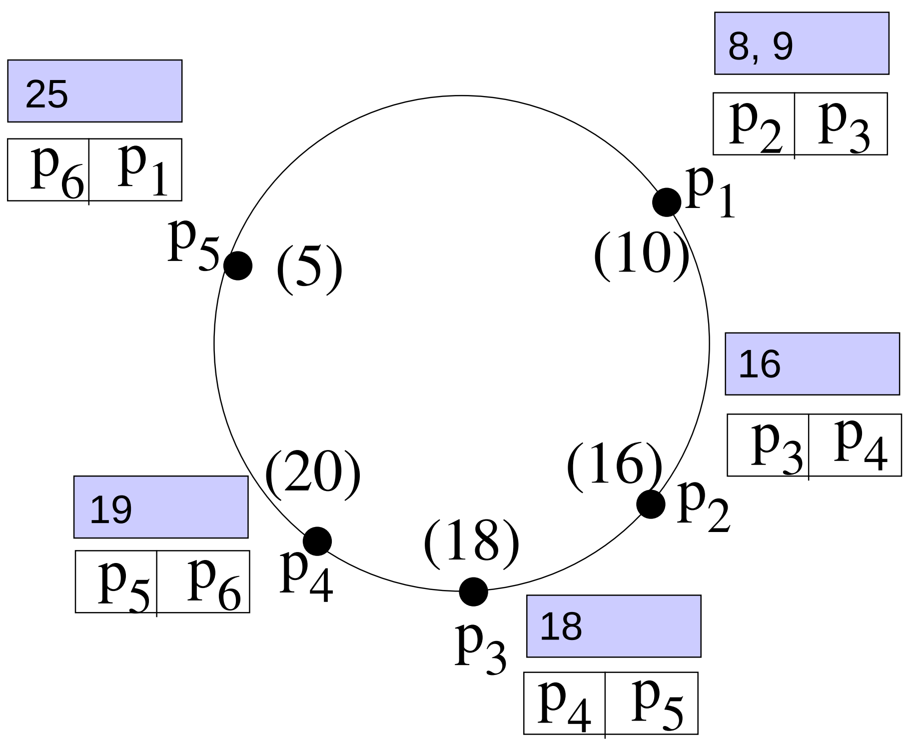


Figure 6.10: **After Redistribute**: *System after peer p_2 redistributes with peer p_3*

Now assume that p_5 fails. Due to the Replication Manager, p takes over the range $(20, 6]$ and adds the data item i such that $\mathcal{M}(i.skv) = 25$ into its Data Store. The state of the system at this time is now shown in Figure 6.9. Now assume that a search Q originates at p_4 for the range $(20, 9]$. Since $p_4.val$ is the lower bound of the query range, p_4 tries to forward the message to the first peer in its successor list (p_5), and on detecting that it has failed, forwards it to the next peer in its successor list (p_1). p_1 returns the items in the range $(6, 10]$, but the items in the range $(20, 6]$ are missed! (Even though all items in this range are live – they are in p's Data Store.) This problem arises because the successor pointers for p_4 are temporarily inconsistent during the insertion of p (they point to p_1 instead of p). Eventually, of course, the ring will stabilize and p_4 will point to p as its successor, but *before* this ring stabilization, query results can be missed.

At this point, the reader might be wondering whether a simple "fix" might address the above problem. Specifically, what if p_1 simply rejects the search request from p_4 (since p_4 is not p_1's predecessor) until the ring stabilizes? The problem with this approach is that p_1 does not know whether p has also failed, in which case p_4 is indeed p_1's predecessor, and it should accept the message. Again, the basic problem is that a peer does not know precise information about other peers in the system (due to the dynamics of a P2P system), and hence potential inconsistencies can occur. We note that the scenario outlined in Figure 6.9 is just one example of inconsistencies that can occur in the ring – rings with longer successor lists can have other, more subtle, inconsistencies (for instance, when p is not the direct predecessor of p_1).

Concurrency in the Data Store

We now show how concurrency issues in the Data Store can produce incorrect query results, *even if the ring is fully consistent.* We shall illustrate the problem in the context of a Data Store redistribute operation; similar problems also arise for Data Store splits and merges.

Consider again the system in Figure 6.5 and assume that a query Q with query range $(10, 18]$ is issued at p_2. Since the lower bound of $p_2.range$ is the same as the lower bound of the query range, the sequential scan for the query range starts at p_2. The sequential scan operation first gets the data items in p_2's Data Store, and then gets the successor of p_2 in the ring, which is p_3. Now assume that the item i with $\mathcal{M}(i.skv) = 11$ is deleted from the index. This causes p_2 to become underfull (since it has no items left in its Data Store), and it hence redistributes with its successor p_3. After the redistribution, p_2 becomes responsible for the item i_1 with $\mathcal{M}(i_1.skv) = 16$, and p_3 is no longer responsible for this item. The current state of the index is shown in Figure 6.10.

Now assume that the sequential scan of the query resumes, and the scan operation propagates the scan to p_3 (the successor of p_2). However, the scan operation will miss item i_1 with $\mathcal{M}(i_1.skv) = 16$, even though i_1 satisfies the query range and was live throughout the execution of the query! This problem arises because of the concurrency issues in the Data Store - the range that p_2's Data Store was responsible for changed while p_2 was processing a query. Consequently, some query results were missed.

6.4.3 Ensuring Correct Query Results

We now present solutions that avoid the above scenarios and provably guarantee that the sequential scan along the ring for range queries will produce correct query results. The attractive feature of our solution is that these enhancements are confined to the Ring and Data Store components of the architecture, and higher layers (both applications on top of the P2P system and other components of the P2P system itself) can be guaranteed correctness by accessing the components through the appropriate API. We first present a solution that addresses ring inconsistency, and then present a solution that addresses Data Store concurrency issues.

Handling Ring Inconsistency

As illustrated in Section 6.4.2, query results can be incorrect if a peer's successor list pointers are temporarily inconsistent (we shall formally define the notion of consistency soon). Perhaps the simplest way to solve this problem is to explicitly avoid this inconsistency by atomically updating the successor pointers of every relevant peer during each peer insertion. For instance, in the example in Section 6.4.2, we could have avoided the inconsistency if p_5's *and* p_4's successor pointers had been atomically updated during p's insertion. Unfortunately, this is not a viable solution in a P2P system because there is no easy way to determine the peers whose successor lists will be affected by an insertion since other peers can concurrently enter, leave or fail, and any cached information can become outdated.

To address this problem, we introduce a new and simple method for implementing `insertSuccessor` (Figure 1) that ensures that successor pointers are always consistent even in the face of concurrent peer insertions and failures (peer deletions are considered in the next section). Our technique works asynchronously and does not require any up-to-date cached information or global co-ordination among peers. The main idea is as follows. Each peer in the ring can be in one of two states: `JOINING` or `JOINED`. When a peer is initially inserted into the system, it is in the `JOINING` state. Pointers to peers in the `JOINING` state need not be consistent. However, each `JOINING` peer transitions to the `JOINED` state in some bounded time. We ensure that the successor pointers to/from `JOINED` peers are always consistent. The intuition behind our solution is that a peer p remains in the `JOINING` state until all relevant peers know about p – it then transitions to the `JOINED` state. Higher layers, such as the Data Store, only store items in peers in the `JOINED` state, and hence avoid inconsistencies.

We now formally define the notion of consistent successor pointers. We then present our distributed, asynchronous algorithm for `insertSuccessor` that satisfies this property for `JOINED` peers.

Defining Consistent Successor Pointers

We first introduce some notation. $p.succList_t$ is the successor list of peer p at time t. $succList.length$ is the length (number of pointers) of $succList$, and $succList[i]$ $(0 \leq i < succList.length)$ refers to the i'th pointer in $succList$. The value of a peer at time t is $p.val_t$. A peer is said to be non-failed at time t if it did not fail at any time $t' \leq t$.

Definition (Consistent Successor Pointers): A set of non-failed peers $\mathcal{P}$ at time t has *consistent successor pointers at time* t iff the following condition holds:

- $\forall p \in \mathcal{P}, \forall i\ (0 \leq i \leq p.succList_t.length \wedge p.succList_t[i] \in \mathcal{P}) \Rightarrow \forall p_1 \in \mathcal{P}\ (p_1.val_t \in (p.val_t, p.succList_t[i].val_t) \Rightarrow \exists j(0 \leq j < i \wedge p.succList_t[j] = p_1))$

In other words, the successor pointers of a set of peers $\mathcal{P}$ is consistent iff for every peer $p \in \mathcal{P}$, if p has a pointer to a peer $p' \in \mathcal{P}$ in its successor list, it also has pointers to all peers $p_1 \in \mathcal{P}$ such that $p_1.val \in (p.val, p'val)$. Intuitively, this means that p cannot have "missing" pointers to peers in the set P. In our example in Figure 6.8, the successor pointers are not consistent with respect to the set of all peers in the system because p_4 has a pointer to p_1 but not to p.

Proposed Algorithm

We first present the intuition behind our algorithm, before presenting the details. Assume that a peer p' is to be inserted as a successor of a peer p. Initially, p' will be in the `JOINING` state. Eventually, we want p' to transition to the `JOINED` state, without violating the consistency of successor pointers. According to the definition of consistent successor pointers, the only way in which converting p' from the `JOINING` state to the `JOINED` state can violate consistency is if there exists some `JOINED` peers p_x and p_y such that: $p_x.succList[i] = p$ and $p_x.succList[i+k] = p_y$ (for some $k > 0$) and for all $j, 0 < j < k$, $p_x.succList[i+j] \neq p'$. In other words, p_x has a pointer to p and p_y but not a pointer to p' whose value occurs between $p.val$ and $p_y.val$.

Our algorithm avoids this case by ensuring that at the time p' changes from the `JOINING` state to the `JOINED` state, if p_x has pointers to p and p_y (where p_y's pointer occurs after p's pointer), then it also has a pointer to p'. It ensures this property by propagating the pointer to p' to all of p's predecessors until it reaches the predecessor whose last pointer in the successor list is p (which thus does not have a p_y that can violate the condition). At this point, it transitions p' from the `JOINING` to the `JOINED` state. This propagation of p' pointer is piggyback on the

Algorithm 1 : p_1.`insertSuccessor(Peer p)`

1: // Insert p into lists as a `JOINING` peer

2: writeLock *succList*, *stateList*

3: *succList.push_front*(p)

4: *stateList.push_front*(`JOINING`)

5: releaseLock *stateList*, *succList*

6: // Wait for successful insert ack

7: wait for ack from some predecessor; on ack do:

8: // Notify p of successful insertion and update lists

9: writeLock *succList*, *stateList*

10: Send a message to p indicating it is now `JOINED`

11: *stateList.update_front*(`JOINED`)

12: *succList.pop_back*(), *stateList.pop_back*()

13: releaseLock *stateList*, *succList*

Chord ring stabilization protocol, and hence does not introduce new messages.

Algorithms 1 and 2 show the pseudocode for the `insertSuccessor` method and the modified ring stabilization protocol, respectively. In the algorithms, we assume that in addition to *succList*, each peer also has a list called *stateList* which stores the state (`JOINING` or `JOINED`) of the corresponding peer in *succList*. We now walk through the algorithms using an example.

Consider again the example in Figure 6.5, where p is to be added as a successor of p_5. The `insertSuccessor` method is invoked on p_5 with a pointer to p as the parameter. The method first acquires a write lock on *succList* and *stateList*, inserts p as the first pointer in p_5.*succList* (thereby increasing its length by one), and inserts a corresponding new entry into p_5.*stateList* with value `JOINING` (lines

Algorithm 2 : `Ring Stabilization`

```
1: // Update lists based on successor's lists
2: readLock succList, stateList
3: get succList/stateList from first non-failed p_s in succList
4: upgradeWriteLock succList, stateList
5: succList = p_s.succList; stateList = p_s.stateList
6: succList.push_front(p_s)
7: stateList.push_front(JOINED)
8: succList.pop_back(), stateList.pop_back()
9: // Handle JOINING peers
10: listLen = succList.length
11: if stateList[listLen − 1] == JOINING then
12:     succList.pop_back(); stateList.pop_back()
13: else if stateList[listLen − 2] == JOINING then
14:     Send an ack to succList[listLen − 3]
15: end if
16: releaseLock stateList, succList
```

2−4 in Algorithm 1). The method then releases the locks on *succList* and *stateList* (lines 5) and blocks waiting for an acknowledgment method from some predecessor peer indicating that it is safe to transition p from the `JOINING` state to the `JOINED` state (line 7). The current state of the system is shown in Figure 6.11 (`JOINING` list entries are marked with a "*").

Now assume that a ring stabilization occurs at p_4. p_4 will first acquire a read lock on its *succList* and *stateList*, contact the first non-failed entry in its successor list, p_5, to get p_5's *succList* and *stateList* (lines $2-3$ in Algorithm 2). p_4 then acquires a write lock on its *succList* and *stateList*, and copies over the *succList* and *stateList* it obtained from p_5 (lines $4-5$). p_4 then inserts p_5 as the first entry in *succList* (increasing its length by 1) and also inserts the corresponding state in *stateList* (the state will always be `JOINED` because `JOINING` nodes do not respond to ring stabilization requests). p_4 then removes the last entries in *succList* and *stateList* (lines $6-8$) to ensure that its lists are of the same length as p_5's lists. The current state of the system is shown in Figure 6.12.

p_4 then checks whether the state of the last entry is `JOINING`; in this case it simply deletes the entry (lines $11-12$) because it is far enough from the `JOINING` node that it does not need to know about it (although this case does not arise in our current scenario for p_4). p_4 then checks if the state of the penultimate peer (p) is `JOINING` – since this is the case in our scenario, p_4 sends a acknowledgment to the peer preceding the penultimate peer (p_5) in the predecessor list indicating that p can be transitioned from `JOINING` to `JOINED` since all relevant predecessors know about p (lines $13-14$). p_4 then releases the locks on its lists (line 16).

The `insertSuccessor` method of p_5, on receiving a message from p_4, first send a message to p indicating that it is now in the `JOINED` state (line 10). p_5 then changes the state of its first list entry (p) to `JOINED` and removes the last entries from its lists in order to shorten them to the regular length (lines $11-12$). The final state after p is inserted into the ring and multiple ring stabilizations have occurred is shown in Figure 6.13.

One optimization we can do to the above method is to *proactively* contact the predecessor in the ring whenever `insertSuccessor` is in progress, to trigger ring stabilization. This will expedite the operation since it will no longer be limited by the frequency of the ring stabilization process.

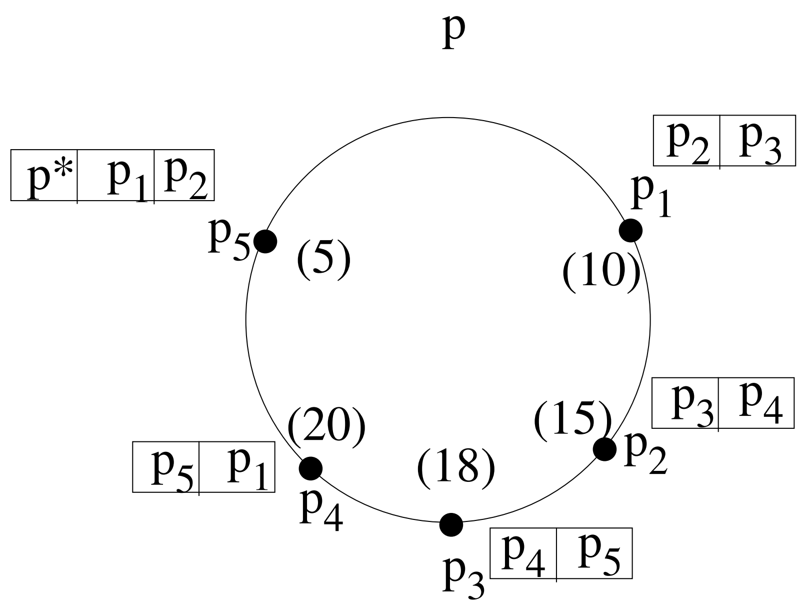


Figure 6.11: **After p_5.insertSuccessor call**

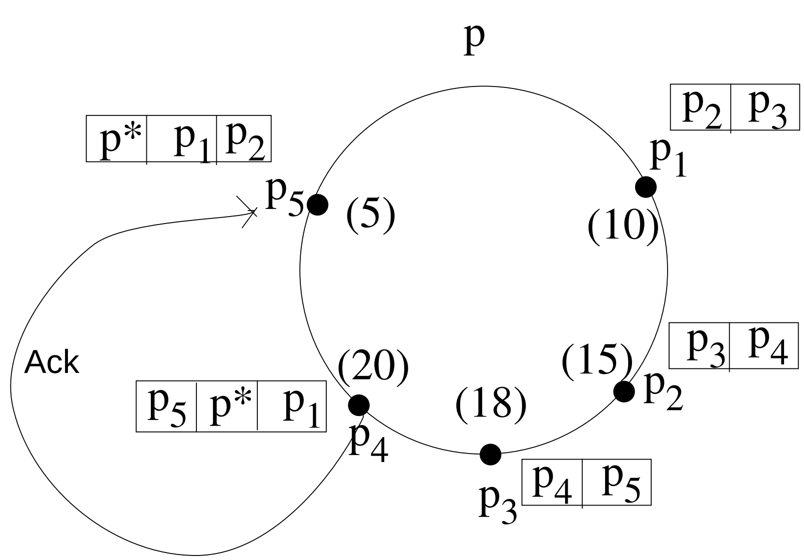


Figure 6.12: **Propagation and final ack**

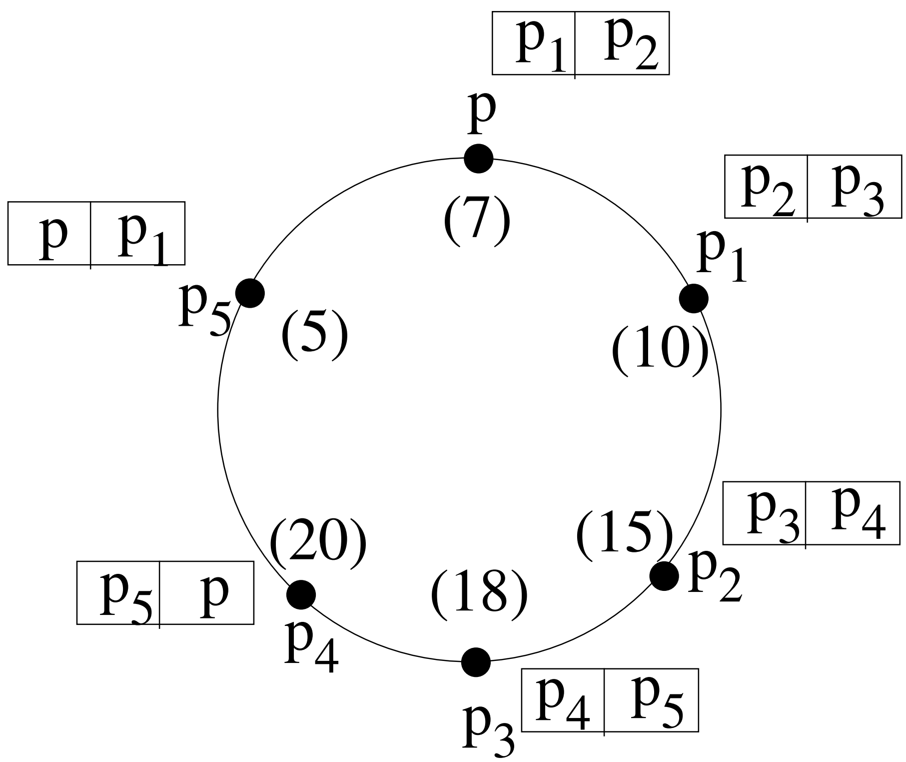


Figure 6.13: **Completed** `insertSuccessor`

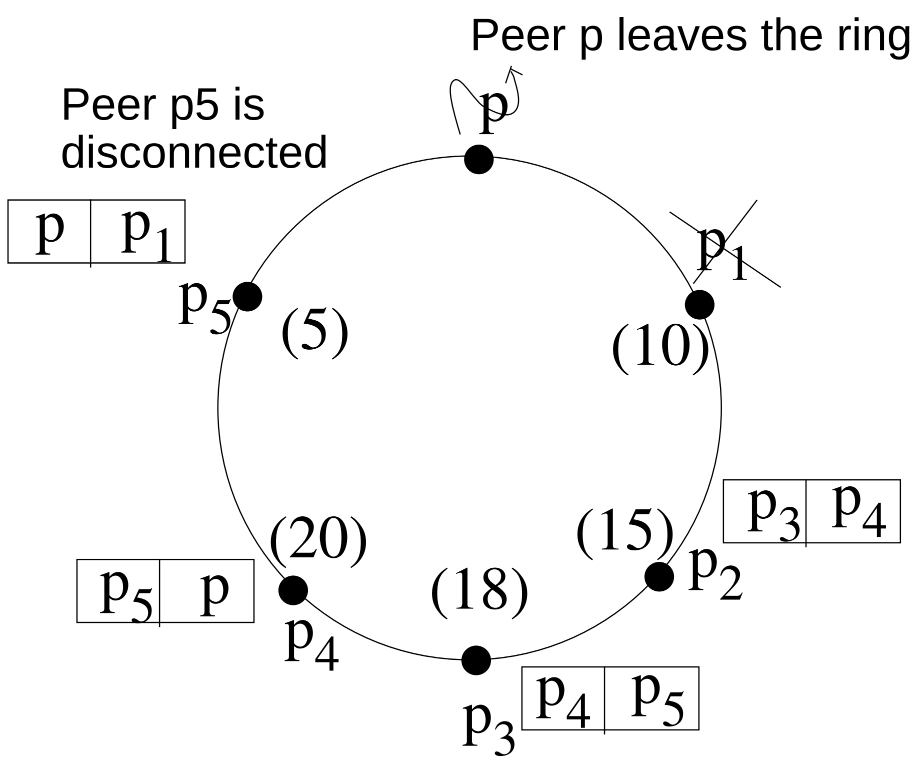


Figure 6.14: **Naive merge leads to decreased reliability**

We can prove the following theorem (the proof can be found in [LCGS05a]):

Theorem 2 (Consistent Successor Pointers) *If P_t is the set of peers in the* `JOINED` *state at time t, then Algorithms 1 and 2 ensure that P_t has consistent successor pointers at time t.*

In addition, we can prove the following theorem about the liveness of the `insertSuccessor` operation (i.e., that it will complete in a bounded time) given a certain peer failure rate, even in the presence of adversarial failures. We use the notation $p.state_t$ to denote the state of peer p at time t.

Theorem 3 (Insert Liveness) *If the stabilization procedure shown in Algo 2 runs at all non-failed peers at least once every T time units, and at most one peer fails every F time units, and $T < F$, then there exists a constant c bounded by $O(TF/(F-T))$ such that if the* `insertSuccessor(p')` *method is invoked on peer p at time t, and if both p and p' are non-failed at time $t+c$, then $p'.state_{t+c} =$* `JOINED`*.*

Handling Data Store Concurrency

Recall from the discussion in Section 6.4.2 that even if the ring is fully consistent, query results can be missed due to concurrency issues at the Data Store. Essentially, the problem is that the range of a peer can change while a query is in progress, causing the query to miss some results. How do we shield the higher layers from the concurrency details of the Data Store while still ensuring correct query results?

Our solution to this problem is as follows. We introduce a new API method called `scanRange` at the Data Store of each peer. This method has the following signature: `scanRange(lb, ub, handlerId, param)`, where (1) *lb* is the lower bound of the range to be scanned, (2) ub is the upper bound of the range to be scanned, (3) *handlerId* is the id of the handler to be invoked on every peer p such that $p.range$ intersects $[lb, ub]$ (i.e., p's range intersects the scan range), and (4) *param* is the parameter to be passed to the handlers. The `scanRange` method should be invoked on the Data Store of the peer p_1 such that $lb \in p_1.range$ (i.e., the first peer whose range intersects the scan range).

`scanRange` handles all the concurrency issues associated with the Data Store. Consequently, higher layers do not have to worry about changes to the Data Store while a scan is in progress. Further, since `scanRange` allows applications to register their own handlers, higher layers can customize the scan to their needs (we shall soon show how we can collect range query results by registering appropriate handlers).

Algorithm 3 shows the pseudocode for the `scanRange` method in a peer p. The method first acquires a read lock on the Data Store *range* (to prevent it from changing) and then checks to make sure that $lb \in p.range$, i.e., p is the first peer in the range to be scanned (lines 1-2). If the check fails, `scanRange` is aborted (lines 3-4). If the check succeeds, then the helper method `processHandler` is invoked. `processHandler` (Algorithm 4) first invokes the appropriate handler for the scan (lines 1-3), and then check to see whether the scan has to be propagated to p's successor (line 4). If so, it invokes the `processScan` method on p's successor.

Algorithm 5 shows the code that executes when p_{succ}.`processScan` is invoked by p.`processHandler`. `processScan` *asynchronously* invokes the `processHandler` method on p_{succ}, and returns. Consequently, p holds on to a lock on its range only until p_{succ} locks its range; once p_{succ} locks its range, p can release its lock, thereby allowing for more concurrency. Note that p can later split, merge, or redistribute, but this will not produce incorrect query results since the scan has already finished scanning the items in p.

We now illustrate the working of these algorithms using an example. Assume that `scanRange`$(10, 18, h_1, param_1)$ is invoked in p_2 in Figure 6.5. p_2 locks its range in `scanRange` (to prevent p_2's range from changing), invokes the handler corresponding to h_1 in `processHandler`, and then invokes `processScan` on p_3. p_3 locks its range in `processScan`, asynchronously invokes `processHandler` and returns. Since p_3.`processScan` returns, p_2 can now release its lock and participate in splits, merges, or redistributions. However, p_3 holds onto a lock on its range until p_3 handler is finished executing. Thus, the algorithms ensure that a peer's range do not change during a scan, but release locks as soon as the scan is propagated to the peer's successor (for maximum concurrency).

We can prove the following correctness theorem (we use the following notation: $r_1 overlaps r_2$ denotes that range r_1 overlaps with range r_2, $r_1 \supseteq r_2$ denotes that range r_1 fully contains range r_2).

Theorem 4 (scanRange Correctness) *Let us consider a p.scanRange(lb, ub, h_1, $param_1$) call that invokes handlers with id h_1 in peers $p_1, ..., p_n$ at times $t_1, ..., t_n$. Then:*

1. $\forall i(1 \leq i \leq n \Rightarrow p_i.range_{t_i} overlaps [lb, ub])$
2. $\forall i \forall j(1 \leq i, j \leq n \wedge i \neq j \Rightarrow \neg(p_i.range_{t_i} overlaps p_j.range_{t_j}))$
3. $\cup_{1 \leq i \leq n}(p_i.range_{t_i}) \supseteq [lb, ub]$.

Using the `scanRange` method, we can easily ensure correct results for range queries by registering the appropriate handler. Algorithm 6 shows the algorithm for evaluating range queries. lb and ub represent the lower and upper bounds of the

Algorithm 3 : $p.scanRange(lb, ub, handlerId, param)$

```
readLock range
if lb ∉ p.range then
    // Abort scanRange
    releaseLock range
else
    // p is the first peer in scan range
    p.processHandler(ub, handlerId, param)
end if
```

range to be scanned, and *pid* represents the id of the peer to which the final result is to be sent. As shown, the algorithm simply invokes the `scanRange` method lb, ub, and the id of the range query handler (shows in Algorithm 7). The parameter to the handler has two parts: the *pid* that the result should be sent to, and the result so far (initially ϕ). The range query handler (Algorithm 7) at a peer p works as follows. It first gets the items in p's Data Store that satisfy the query range and adds them to the result (lines 1-3). Then, if p is the last peer in the query range, it send the results to the peer *pid* (lines 6-7).

Using the above implementation of a range query, the inconsistency described in Section 6.4.2 cannot occur because p_2's range cannot change (and hence redistribution cannot happen) when the search is still active in p_2. We can prove the following correctness theorem:

Theorem 5 (Search Correctness) *For a query $Q = [lb, ub]$, if Algorithm 6 is initiated at peer p at time t such that $p.state_t =$ `JOINED` $\wedge\, lb \in p.range_t$, then the algorithm produces* correct *query results (as per the definition of correct query results in Section 6.4.1).*

Algorithm 4 : $p.processHandler(lb, ub, handlerId, param)$

// Invoke appropriate handler

Get $handler$ with id $handlerId$

newParam = $handler$.handle($ub, param$)

// Forward to successor if required

if $ub \notin p.range$ **then**

  $p_{succ} = p.ring.getSuccessor()$

  $p_{succ}.processScan(lb, ub, handlerId, newParam)$

end if

releaseLock $range$

Algorithm 5 : $p.processScan(lb, ub, handlerId, param)$

readLock $range$

Invoke $p.processHandler(lb, ub, handlerId, param)$ asynchronously

return

6.5 System and Item Availability

We now address system availability and item availability issues. Intuitively, ensuring system availability means that the availability of the index should not be reduced due to routine index maintenance operations (such as splits, merges, and redistributions). Similarly, ensuring item availability means that the availability of items should not be reduced due to index maintenance operations. Our discussion of these two issues is necessarily brief due to space constraints, and we only illustrate the main aspects and sketch our solutions. We refer the reader to the full technical report for details [LCGS05a].

Algorithm 6 : $p.rangeQuery(lb, ub, pid)$

// initiate a scanRange

p.scanRange($lb, ub, rangeQueryHandlerId, < pid, \phi >$)

Algorithm 7 : $p.rangeQueryHandler(lb, ub, < pid, resultSoFar >)$

// Accumulate results from p's Data Store

Find *items* in p's Data Store in range $[lb, ub]$

newResults = resultsSoFar + *items*

// If p is the last peer in the query range, send results

if $ub \in p.range$ **then**

    send newResults to peer *pid*

end if

return newResults

6.5.1 System Availability

An index is said to be *available* if its Fault Tolerant Ring is connected. The rationale for this definition is that an index can be operational (by scanning along the ring) so long as its peers are connected. The Chord Fault Tolerant Ring provides strong availability guarantees [SMK+01] when the only operations on the ring are peer insertions (splits) and failures. These availability guarantees also carry over to our variant of the Fault Tolerant Ring with the new implementation of `insertSuccessor` described earlier because it is a stronger version of the Chord's corresponding primitive (it satisfies all the properties required for the Chord proofs). Thus, the only index maintenance operation that can reduce the availability of the system is the merge operation in the Data Store, which translates to the `leaveRing` operation in the Fault Tolerant Ring. Note that the redistribute operation in the Data Store does not affect the ring.

We now show that a naive implementation of `leaveRing`, which is simply removing the merged peer from the ring, does in fact reduce system availability. We then sketch an alternative implementation for the `leaveRing` that provably does not reduce system reliability. Using this new implementation, the Data Store can perform a merge operation without knowing the details of the ring stabilization, while being guaranteed that system availability is not compromised.

Naive `leaveRing` Reduces System Availability: Consider the system in Figure 6.13 in which the length of the successor list of each peer is 2. Without a `leaveRing` primitive, this system can tolerate one failure per peer stabilization round without disconnecting the ring (since at most one of a peer's two successor pointers can become invalid before the stabilization round). We now show that in the presence of the naive `leaveRing`, a single failure can disconnect the ring. Thus, `leaveRing` reduces the availability of the system. The example is as follows. Assume that `leaveRing` is invoked on p, and p immediately leaves the ring. Now assume that p_1 fails (this is the single failure). The current state of the system is shown in Figure 6.14, and as we can see, the ring is disconnected since none of p_5's successor pointers point to peers in the ring.

Solution Sketch: The reason the naive implementation of `leaveRing` reduced availability is that pointers to the peer p leaving the ring become invalid. Hence, the successor lists of the peers pointing to p effectively decreases by one, thereby reducing availability. To avoid this problem, our solution is to increase the successor list lengths of all peers pointing to p by one. In this way, when p leaves, the availability of the system is not compromised. As in the `insertSuccessor` case, we piggyback the lengthening of the successor lists on the ring stabilization protocol. This is illustrated in the following example.

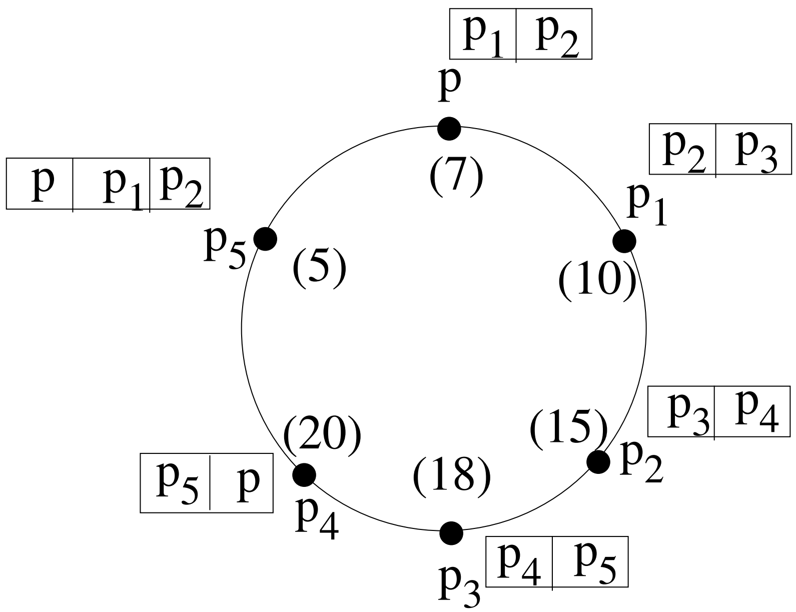


Figure 6.15: **Controlled leave of peer** p

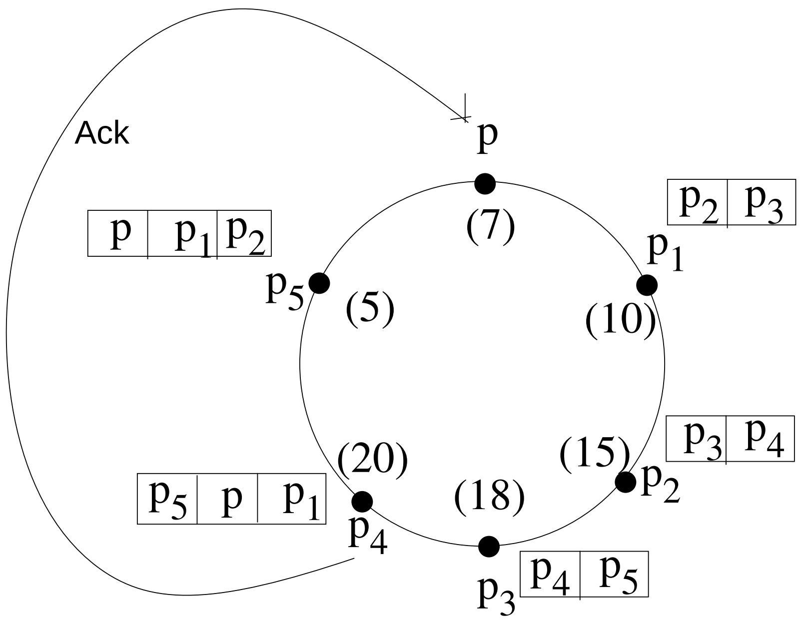


Figure 6.16: **Final Ack**:*Final ack received at peer p. Peer p is good to go*

Consider Figure 6.13 in which `leaveRing` is invoked on p. During the next ring stabilization, the predecessor of p, which is p_5, increases its successor list length by 1. The state of the system is shown in Figure 6.15. During the next ring stabilization, the predecessor of p_5, which is p_4, increases its successor list length by 1. Since p_4 is the last predecessor that knows about p, p_4 sends a message to p indicating that it is safe to leave the ring. The state of the system at this point is shown in Figure 6.16. It is easy to see that if p leaves the ring at this point, a single failure cannot disconnect the ring, as in was the case in the previous example. We can formally prove that the new algorithm for `leaveRing` does not reduce the availability of the system [LCGS05a].

6.5.2 Item Availability

We first formalize the notion of item availability in a P2P index. We represent the successful insertion of an item i at time t into an index I by $insert_{I,t}(i)$, and the deletion of an item i' at time t' as $delete_{I,t'}(i')$.

Definition (Item Availability): An index I is said to preserve *item availability* with respect to a set of insertions Ins and a set of deletions Del iff $\forall i \forall t(\exists t'(t' \leq t \wedge insert_{I,t'}(i) \in Ins \wedge \forall t''(t' \leq t'' \leq t \Rightarrow delete_{I,t''}(i) \notin Del))) \Rightarrow live_I(i,t)$.

In other words, for all time t when an item i has been inserted into the system and not deleted, i is a live item.

The CFS Replication Manager, implemented on top of the Chord Ring provides strong guarantees [DKK$^+$01] on item availability when the only operations on the ring are peer insertions and failures, and these carry over to our system too. Thus, the only operation that could compromise item availability is the `leaveRing` (merge) operation. We now show that using the original CFS Replication Manager in the presence of merges does in fact compromise item availability. We then describe a modification to the CFS Replication Manager and its interaction with the Data Store that ensures the original guarantees on item availability.

Scenario that Reduces Item Availability: Consider the system in Figure 6.7. Here, the top box associated with each peer represents the items replicated at that peer (recall that CFS replicates items along the ring). In this example, each item is replicated to one successor along the ring; hence, the system can tolerate one failure between replica refreshes. We now show how, in the presence of Data Store merges, a single failure can compromise item availability. Assume that peer p_1 wishes to merge with p_2 in Figure 6.7. p_1 thus does a `leaveRing` operation, and once it is successful, it transfers its Data Store items to p_2 and leaves the system. The state of the system at this time is shown in Figure 6.17. If p_5 fails at this time (this is the single failure), the items i such that $\mathcal{M}(i.skv) = 25$ is lost.

Solution Sketch: The reason item availability was compromised in the above example is because when p_1 left the system, the replicas it stored were lost, thereby reducing the number of replicas for certain items in the system. Our solution is to replicate the items stored in the merging peer p's and Replication Manager for one additional hop before p leaves the system. This is illustrated in Figure 6.18, where before p_1 merges with p_2, it increase the replicas for items in its Data Store and Replication Manager by one additional hop. Then, when p_1 finally merges with p_2 and leaves the system, the number of replicas is not reduced, thereby preserving item availability. We can again prove that the above scheme preserves item availability even in the presence of concurrent splits, merges, and redistributions [LCGS05a].

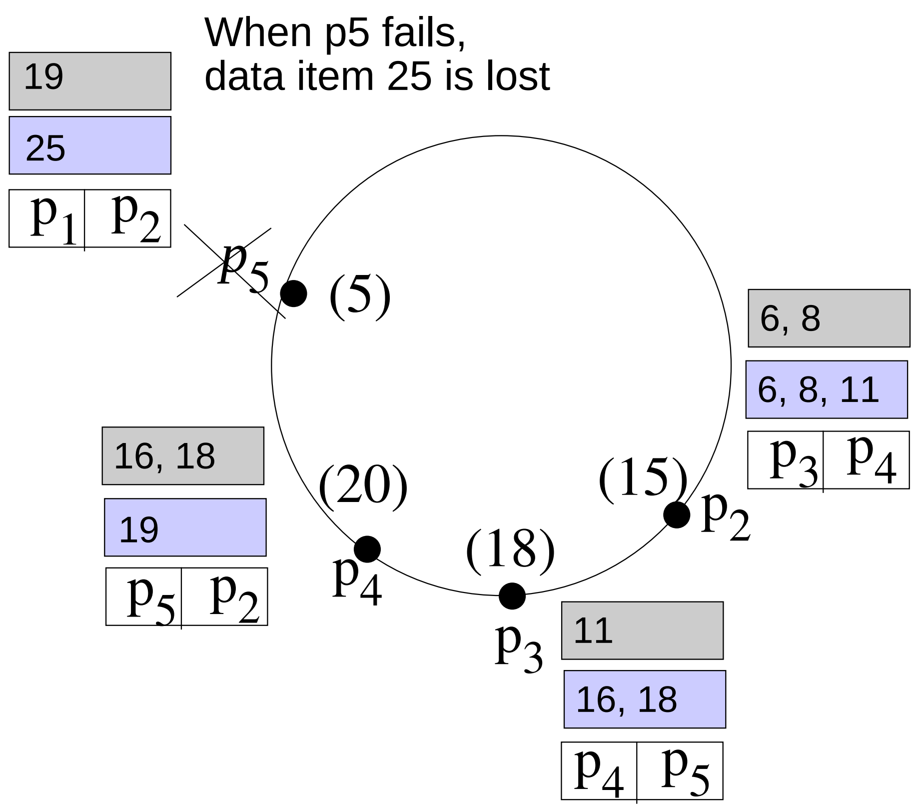


Figure 6.17: **Reduced Item Availability**: *Peer* p_5 *fails, causing loss of item* 25

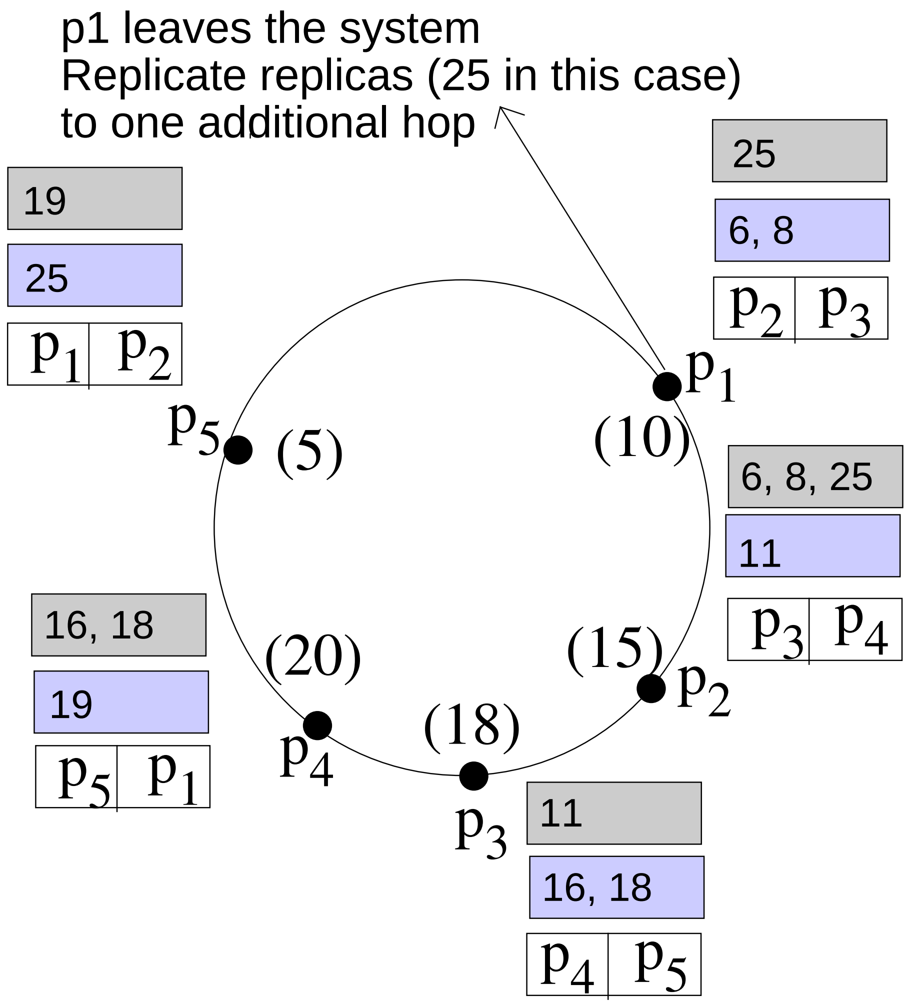


Figure 6.18: **Item Availability**: *Replicate item* 25 *one additional hop*

6.6 Experimental Evaluation

We had two main goals in our experimental evaluation: (1) to demonstrate the feasibility of our proposed query correctness and availability algorithms in a dynamic P2P system, and (2) to measure the overhead of our proposed techniques. Towards these goal, we implemented the P-Ring index, along with our proposed correctness and availability algorithms, in a real distributed environment with concurrently running peers. We used this implementation to measure the overhead of each of our proposed techniques as compared to the naive approach, which does not guarantee correctness or availability.

6.6.1 Experimental Setup

We implemented the P-Ring index as an instantiation of the indexing framework (Section 6.2.3). The code was written in C++ and all experiments were run on a cluster of workstations, each of which had 1GHz processor, 1GB of main memory and at least 15GB of disk space. All experiments were performed with 30 peers running concurrently on 10 machines (with 3 peers per machine). The machines were connected by a local area network.

We used the following default parameter values for our experiments. The length of the Chord Fault-Tolerant Ring successor list was 4 (which means that the ring can tolerate up to 3 failures without being disconnected if the ring is fully consistent). The ring stabilization period was 4 seconds. We set the storage factor of the P-Ring Data Store to be 5, which means that it can hold between 5 and 10 data items. The replication factor in the Replication Manager is 6, which means that each item is replicated 6 times. We vary these parameters too in some of the experiments.

We ran experiments in two modes of the system. The first mode was the *fail-free* mode, where there were no peers failures (although peers are still dynamically added and splits, merges, and redistributes occur in this state). The second was the *failure* mode, where we introduced peer failures by killing peers. For both modes, we added peers at a rate of one peer every 3 seconds, and data items were added at the rate of 2 items per second. We also vary the rate of peer failures in the failure mode.

6.6.2 Implemented Approaches

We implemented and evaluated all four of the techniques proposed in this chapter. Specifically, we evaluate (1) the *insertSuccessor* operation that guarantees ring consistency, (2) the *scanRange* operation that guarantees correct query results, (3) the *leaveRing* operation that guarantees system availability, and (4) the *replication to additional hop* operation that guarantees item availability. For *scanRange*, we implemented a synchronous version where the *processHandler* is invoked synchronously at each peer (see Algorithm 5).

One of our goals was to show that the proposed techniques actually work in a real distributed dynamic P2P system. The other goal was to compare each solution with a naive approach (that does not provide correctness or availability guarantees). Specifically, for the *insertSuccessor* operation, we compare it with the naive *insertSuccessor*, where the joining peer simply contacts its successor and becomes part of the ring. For the *scanRange* operation, we compare it with the naive range query method whereby the application explicitly scans the ring without using the *scanRange* primitive. For the *leaveRing* operation, we compare with the naive approach where the peer simply leaves the system without notifying other peers. Finally, for the *replication to additional hop* operation, we compare with the naive approach where additional replication is not done.

6.6.3 Experimental Results

We now present our experimental results. We first present results in the fail-free mode, and then present results in the failure mode.

Evaluating *insertSuccessor*

In this section we will present the results quantifying the overhead of our *insertSuccessor* when compared to the naive *insertSuccessor*. The performance metric used is the time to complete the operation; this time is averaged over all such operations in the system during the run of the experiment.

We vary two parameters that affect the performance of the operations. The first parameter is the length of the ring successor list. The longer the list, the farther *insertSuccessor* has to propagate information before it can complete. The second is the ring stabilization period. The longer the stabilization period, the slower information about leaving peers propagates due to stabilization.

Figure 6.19 shows the effect of varying the ring successor list length. There are several aspects to note about this figure. First, the time for our *insertSuccessor* increases linearly with the successor list length, while the time for the naive *insertSucessor* remains constant. This is to be expected because the naive *insertSuccessor* only contacts the successor, while our *insertSuccessor* propagates information to as many predecessors as the length of the successor list. Second, perhaps surprisingly, the rate of increase of the time for our *insertSuccessor* operation is very

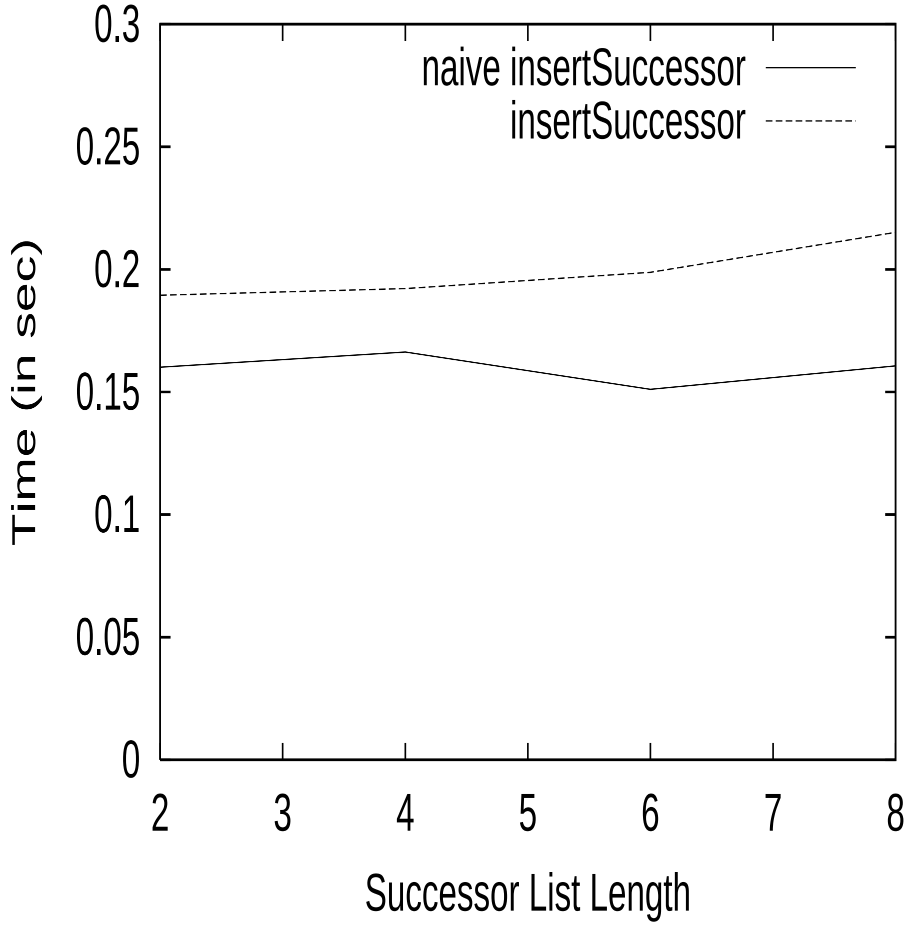


Figure 6.19: **Overhead of insertSuccessor**:*Plot showing effect of varying successor list length*

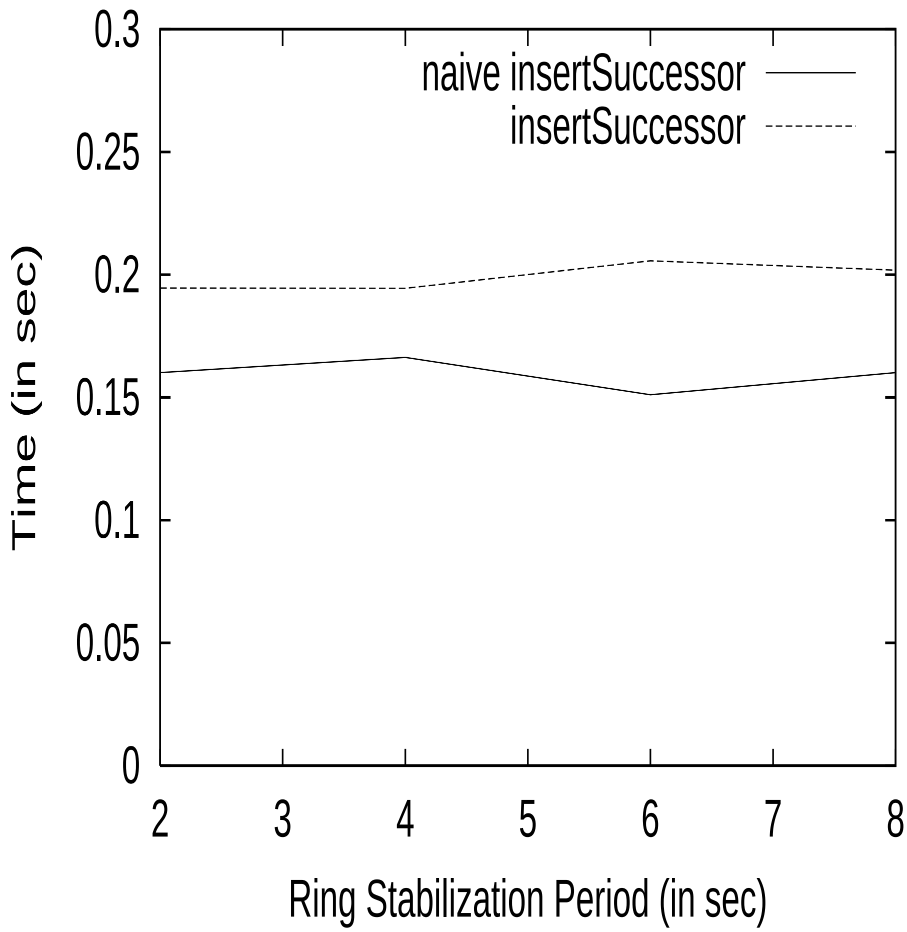


Figure 6.20: **Overhead of insertSuccessor**:*Plot showing the effect of Ring Stabilization period*

small; this can be attributed to the optimization discussed in Section 6.4.3, where we proactively contact predecessors instead of only relying on the stabilization. Finally, an encouraging result is that the cost of our *insertSuccessor* is of the same ball park as that of the naive *insertSuccessor*; this means that users do not pay too high a price for consistency.

Figure 6.20 shows the result of varying the ring stabilization frequency. The results are similar to varying the successor list length. Varying the ring stabilization period also has less of an effect on our *insertSuccessor* because of our optimization of proactively contacting predecessors.

Evaluating ***scanRange***

In this section, we investigate the overhead of using *scanRange* when compared to the naive approach of the application scanning the range by itself. Since the number of messages needed to complete the operation is the same for both approaches, we used the elapsed time to complete the range search as the relevant performance metric. We varied the size of the range to investigate its effect on performance, and averaged the elapsed time over all the searches requiring the same number of hops along the ring. Each peer generates searchs for ranges of different sizes, and we measured the time needed to process the range search, once the first peer with items in the search range was found. This allows us to isolate the effects of scanning along the ring.

Figure 6.21 shows the performance results. As shown, there is practically no overhead to using *scanRange* as compared with the application level search; again, this indicates that the price of consistency is low. To our surprise, the time needed to complete the range search, for either approach, does not increase significantly

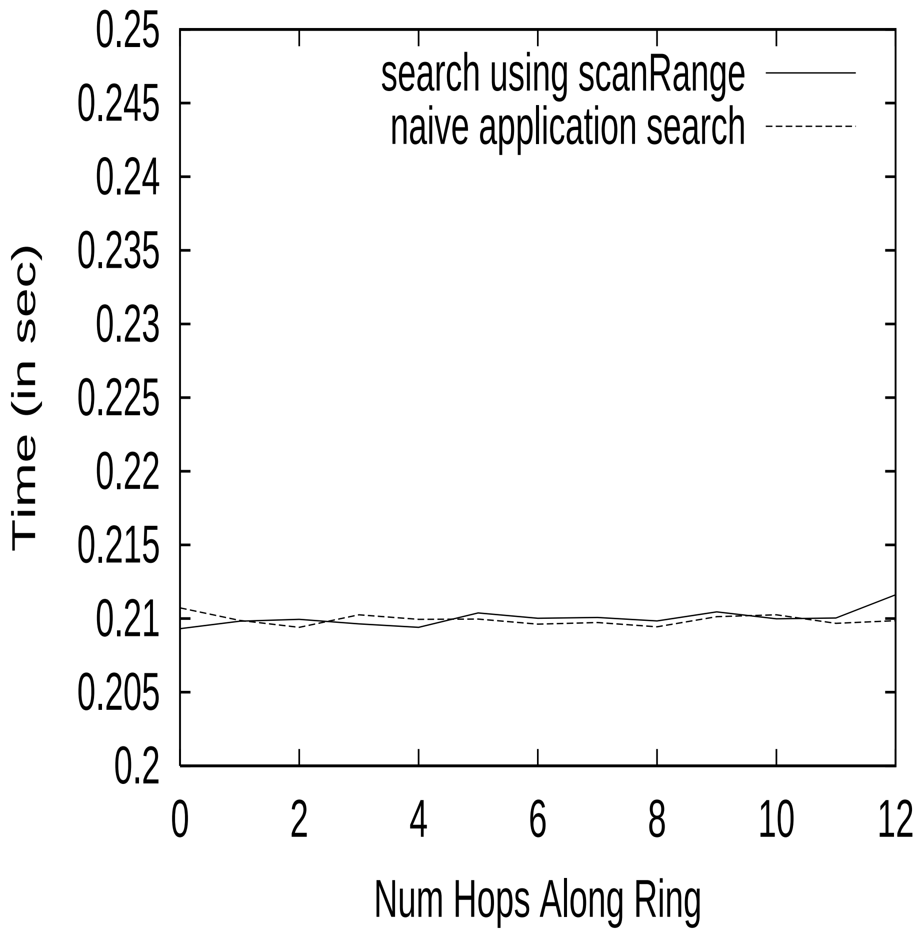


Figure 6.21: **Overhead of scanRange**

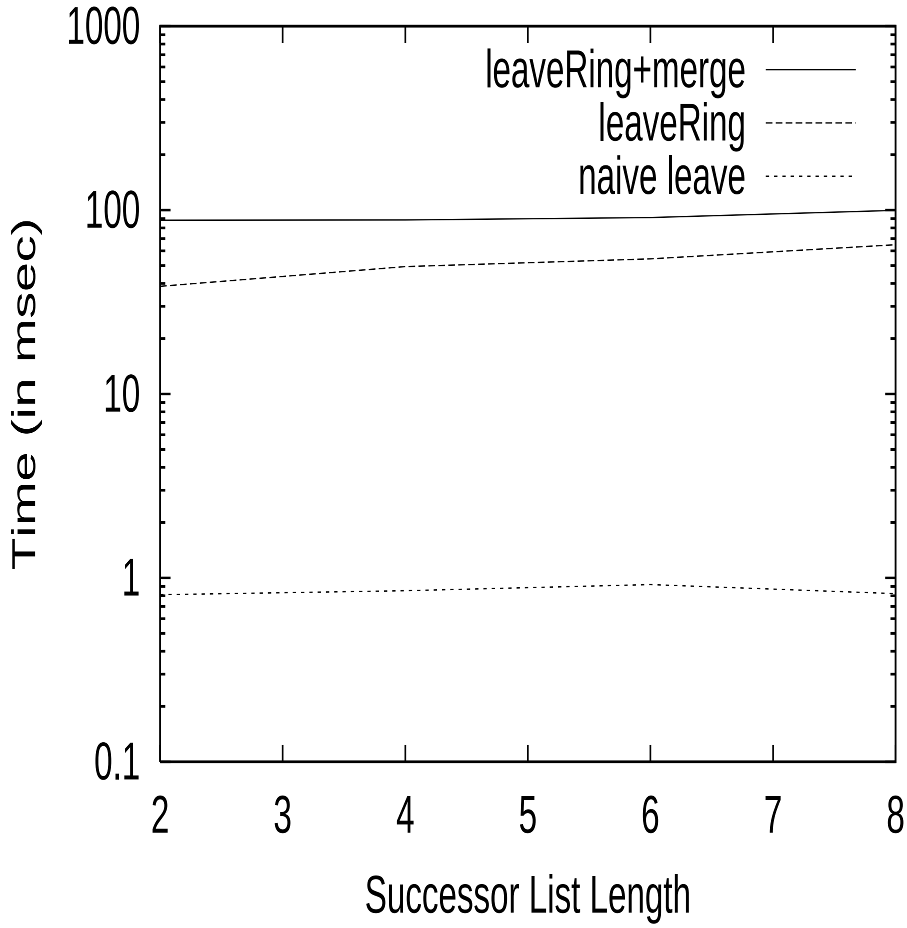


Figure 6.22: **Overhead of leaveRing**

with the increased number of hops. On further investigation, we determined that this was due to our experiments running on a cluster in the local area network. In a wide area network, we expect the time to complete a range search to increase significantly with the number of hops.

Evaluating *leaveRing* and *Replicate to additional hop*

In this section, we investigate the overhead of the proposed *leaveRing* and *replicate to additional hop* operations as compared to the naive approach of simply leaving the ring without contacting any peer. For this experiment, we start with a system of 30 peers and delete items from the system that cause peers to merge and leave the ring.

We measure the time elapsed for three operations: (1) the *leaveRing* operation in the ring, and (2) the merge operation in the Data Store (which includes the time for *replicate to additional hop*), and (3) the naive *leaveRing*. Figure 6.22 shows the variation of the three times with successor list length. Note the log scale on y-axis. We observe that the *leaveRing* and merge operations take approximately 100 msec, and do not constitute a big overhead. The naive version takes only 1 msec since it simply leaves the system.

Evaluation in Failure Mode

We have so far studied the overhead of our proposed techniques in a system without failures. We will now see how our system behaves in a system with failures. In particular, we will measure the variation of the average time taken for a insertSuccessor operation with the failure rate of peers. The system setting is as follows: We insert one peer every three seconds into the system, and we insert two items every second. We use the default successor list length (4) and default ring stabilization period (4 sec).

Figure 6.23 shows the variation of average time taken for a insertSuccessor operation with the peer failure rate. We observe that even in the case when the failure rate is as high as 1 in every 10 seconds, the time taken for insertSuccessor is not prohibitive (about 1.2 seconds compared to 0.2 seconds in a stable system).

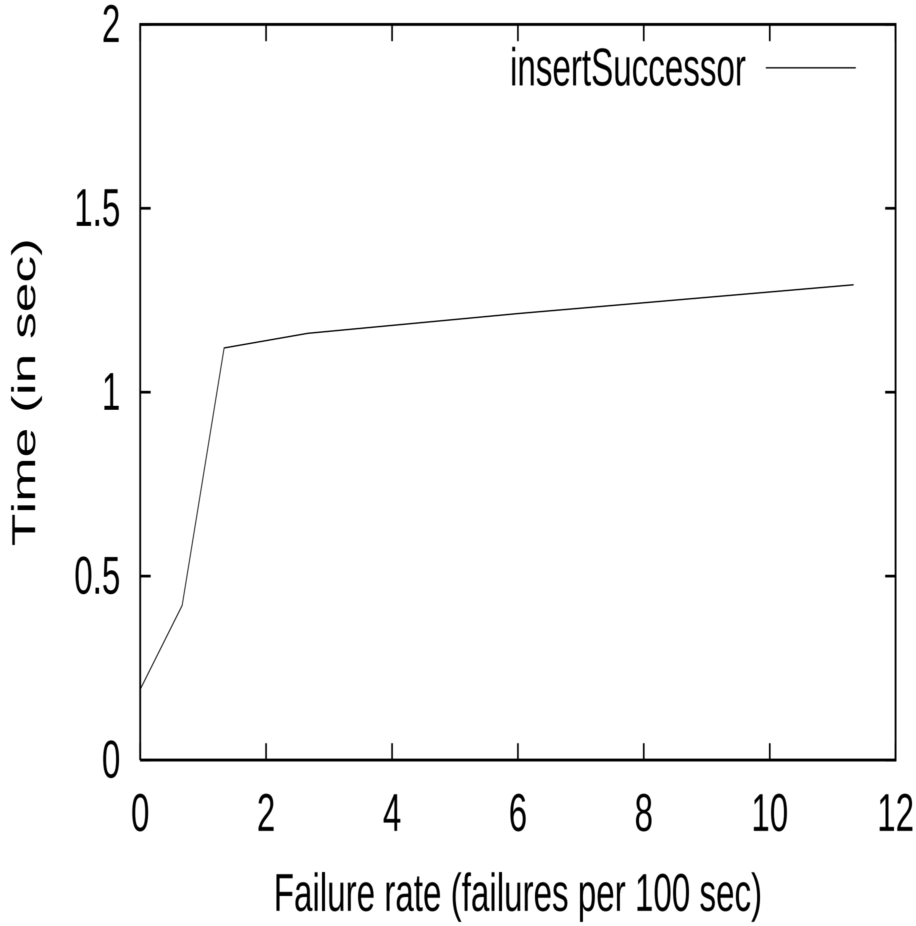


Figure 6.23: ***insertSuccessor* in failure mode**

6.7 Related Work

There has been a flurry of recent activity on developing indices for structured P2P systems. Some of these indices [RFH+01, SMK+01, RD01b] can efficiently support equality queries, while others can support both equality and range queries (e.g., [Abe01, AS03, BRS04, CLM+04a, DGA03, GBGM04, GAE03a, GLS+04b]. This paper addresses query correctness and availability issues for such indices, which have not been previously addressed for range queries. Besides structured P2P indices, there are unstructured P2P indices such as [webc, CGM02]. Unstructured indices are very robust to failures, but do not provide guarantees on query correctness and item availability. Since one of our main goals was to study correctness and availability issues, we focus on structured P2P indices.

There is a rich body of work on developing distributed index structures for databases (e.g., [JC92b, JK93, KW94, LNS94, Lom96]. However, most of these techniques maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability problems when dealing with thousands of peers. Some index structures, however, do maintain replicas lazily (e.g., [JK93, KW94, Lom96]). However, these schemes are not designed to work in the presence of peer failures, dynamic item replication and reorganization, which makes them inadequate in a P2P setting. In contrast, our techniques are designed to handle peer failures while still providing correctness and availability guarantees.

Besides indexing, there is also some recent work on other data management issues in P2P systems such as complex queries [GWJD03, HHL+03, WSNZ03, VPT03, TH04, TXKN03]. An interesting direction for future work is to extend our techniques for query correctness and system availability to work for complex queries such as joins and aggregations.

6.8 Conclusion

We have introduced the first set of techniques that provably guarantee query correctness and system and item availability for range index structures in P2P systems. Our techniques provide provable guarantees, and they allow applications to abstract away all possible concurrency and availability issues. We have implemented our techniques in a real distributed P2P system, and quantified their performance.

As a next step, we would like to run our system on Planetlab, and we would like to perform a thorough performance comparison between different P2P range index structures requiring query correctness and availability guarantees.

BIBLIOGRAPHY

[Abe01] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In *Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, 2001.

[AHKV03] Micah Adler, Eran Halperin, Richard M Karp, and Vijay V Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *STOC*, 2003.

[AS03] James Aspnes and Gauri Shah. Skip graphs. In *SODA*, 2003.

[Bai75] N.T.J. Bailey. *Epidemic Theory of Infectious Diseases and its Applications*. Hafner Press, 1975.

[BCM03] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In *IPTPS*, 2003.

[BFLZ03] Daniel S. Bernstein, Zhengzhu Feng, Brian Neil Levine, and Shlomo Zilberstein. Adaptive peer selection. In *IPTPS*, 2003.

[BGGM04] Mayank Bawa, Aristides Gionis, Hector GarciaMolina, and Rajeev Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.

[BHO+99] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. In *ACM Transactions of Computer Systems*, 1999.

[BRS04] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.

[BSS02] Andr Brinkmann, Kay Salzwedel, and Christian Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *SPAA*, 2002.

[BSV03] R. Bhagwan, S. Savage, and G.M. Voelker. Understanding availability. In *Proc. 2^{nd} International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 135–140, 2003.

[CCN+06] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Anthony Rowstron. Virtual ring routing: Network routing inspired by dhts. In *SIGCOMM*, 2006.

[CDN+96] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proc. 1996 Usenix Technical Conference*, San Diego, CA, jan 1996.

[CGM02] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer networks. In *ICDCS*, 2002.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WWW*, 2004.

[CLM+04a] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: An index structure for peer-to-peer systems. In *Cornell Technical Report*, 2004.

[CLM+04b] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. A storage and indexing framework for p2p systems. In *WWW Poster*, 2004.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert T. Morris. Serving dns using a peer-to-peer lookup service. In *IPTPS*, 2002.

[CRWZ05] Xin Chen, Shansi Ren, Haining Wang, and Xiaodong Zhang. Scope: Scalable consistency maintenance in structured p2p systems. In *INFOCOM*, 2005.

[Dav] B.D. Davison. Web caching and content delivery resources.

[DGA03] A. Daskos, S. Ghandeharizadeh, and X. An. Peper: A distributed range addressing space for p2p systems. In *DBISP2P*, 2003.

[DGH+87] A. Demers, D.H. Greene, J. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proc. 6^{th} ACM Symp. Principles of Distributed Computing (PODC)*, 1987.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[DLN02] D. Karger D. Liben-Nowell, H. Balakrishnan. Observations on the dynamic evolution of peer-to-peer networks. In *Proc. 1^{st} International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag, 2002.

[GAE03a] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[GAE03b] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, California, United States, January 2003.

[GBGM04] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[GBL+03] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[GLS+04a] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[GLS+04b] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

[GRB00] Indranil Gupta, Robbert Van Renesse, and Kenneth P. Birman. A probabilistically correct leader election protocol for large groups. In *International Symposium on Distributed Computing*, 2000.

[GWJD03] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[HHB+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, September 2003.

[HHL+03] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[ICH00] Brandon Wiley Ian Clarke, Oskar Sandberg and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *In Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[IPT02] *Proc. 1st Intnl. Wshop. Peer-to-Peer Systems (IPTPS)*. LNCS 2429, Springer-Verlag, 2002.

[IRD02] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[JC92a] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *International Parallel Processing Symposium*, pages 319–325, March 1992.

[JC92b] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *IPPS*, 1992.

[JK93] Theodore Johnson and Padmashree Krishna. Lazy updates for distributed search structures. In *Proceedings of the ACM SIGMOD Conference on Management of Data, May, 1993, Washington D.C.*, pages 337–346, 1993.

[JOV05] H. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

[KK03a] M. F. Kaashoek and D. R. Karger. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'2003)*, 2003.

[KK03b] M. F. Kaashoek and D. R. Karger. Koorder: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'2003)*, 2003.

[KKD01] D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. In *Proc. 33rd ACM Symp. Theory of Computing (STOC)*, 2001.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[Kos00] D. Kossmann. The state of the art in distributed query processing. In *ACM Computing Surveys*, Sep 2000.

[KR04] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *IPTPS*, 2004.

[KW94] Brigitte Krll and Peter Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, 1994*, pages 265 – 276. ACM Press, 1994.

[LCGS05a] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *Cornell Technical Report*, March 2005.

[LCGS05b] Prakash Linga, Adina Crainiceanu, Johannes Gehrke, and Jayavel Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *SIGMOD*, 2005.

[LNBK02] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proc. 1st International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag, 2002.

[LNS93] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Lh* - linear hashing for distributed files. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 327–336. ACM Press, 1993.

[LNS94] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 342–353. Morgan Kaufmann, 1994.

[Lom96] David B. Lomet. Replicated indexes for distributed data. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida*, pages 108–119. IEEE Computer Society, 1996.

[LS00] Witold Litwin and Thomas Schwarz. Lh*$_{\mathrm{RS}}$: A high-availability scalable distributed data structure using reed solomon codes. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29-2, pages 237–248. ACM, 2000.

[LS05] Jonathan Ledlie and Margo Seltzer. Distributed, secure load balancing with skew, heterogeneity, and churn. In *INFOCOM*, 2005.

[Man04] Gurmeet Singh Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *PODC*, 2004.

[MNR02] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of Distributed Computing (PODS'02)*, 2002.

[Moh02] C. Mohan. Caching technologies for web applications. In *Talk at Cornell University, Ithaca, NY*, 2002.

[NW03] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuousdiscrete approach. In *SPAA*, 2003.

[PH97] D. Provey and J. Harrison. A distributed internet cache. In *Proc. 20^{th} Australian Computer Science Conference*, Sydney, Australia, feb 1997.

[PS02] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *Proc. 1^{st} International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag, 2002.

[RD01a] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California*, August 2001.

[RLSK03] A. Rao, K. Lakshminarayanan, S. Surana, and R. Karp. Load balancing in structured p2p systems. In *IPTPS*, 2003.

[RS04a] Venugopalan Ramasubramanian and Emin Gun Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM*, 2004.

[RS04b] Venugopalan Ramasubramanian and Emin Gn Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI*, 2004.

[Sar04] Carlo Sartiani. On the correctness of query results in xml p2p databases. In *P2P*, 2004.

[SGAA04] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[SGG02] S. Saroiu, P.K. Gummadi, and S.D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[SMW+03] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. Odissea: A peer-to-peer architecture for scalable web search and information retrieval, 2003.

[STZ04] Subhash Suri, Csaba D. Toth, and Yunhong Zhouy. Uncoordinated load balancing and congestion games in p2p systems. In *IPTPS*, 2004.

[TH04] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD*, 2004.

[TJ02] Marvin Theimer and Michael B. Jones. Overlook: Scalable name service on an overlay network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002.

[TXKN03] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high performance peer-to-peer content and resource sharing systems. In *CIDR*, 2003.

[URLa] Modeling topology of large internetworks.

[URLb] Jbi home page.

[URLc] Fireflies of selangor river, malaysia.

[URLd] Internet traffic archive.

[VPT03] David Maier Vassilis Papadimos and Kristin Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[vRMH98] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. IFIP Middleware*, 1998.

[Wan99] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 29(5):36–46, oct 1999.

[WC97] Z. Wang and J. Crowcroft. Cachemesh: a distributed cache system for the world wide web. In *Proc. Web Cache Workshop*, 1997.

[weba] BitTorrent website. http://www.bittorrent.com/.

[webb] Exceem website. http://www.exceem.com/.

[webc] Gnutella website. http://www.gnutella.com/.

[webd] Kazaa website. http://www.kazaa.com/us/index.htm.

[webe] OpenNap website. http://opennap.sourceforge.net/.

[webf] Overnet/eDonkey website. http://edonkey2000.com/.

[webg] Suprnova website. http://www.suprnova.org/.

[Wes] D. Wessel. Squid internet object cache. `http://squid.nlanr.net`.

[WNO+02] X. Y. Wang, W. S. Ng, B. C. Ooi, K. L. Tan, and A. Y. Zhou. Buddyweb: a p2p-based collaborative web caching system. In *Proc. International Workshop on Peer-to-Peer Computing*, 2002.

[WSNZ03] Lee Tan Wee Siong Ng, Beng Chin Ooi and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE 2003*, 2003.

[ZDKS03] F. Dabek B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proc. 2^{nd} International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[ZKJ01] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: an infrastructure for fault-taulerant wide-area location and routing. In *Technical Report UCS/CSD-01-1141, U.C.Berkeley*, 2001.

[ZvRM02] Lidong Zhou, Robbert van Renesse, and Michael March. Implementing ipv6 as a peer-to-peer overlay network. In *Symposium on Reliable Distributed Systems (SRDS 2002)*, 2002.