Providing Freshness for Cached Data in Unstructured Peer-to-Peer Systems

by

Simon Forsyth

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

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

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

Abstract

Replication is a popular technique for increasing data availability and improving performance in peer-to-peer systems. Maintaining freshness of replicated data is challenging due to the high cost of update management. While updates have been studied in structured networks, they have been neglected in unstructured networks. We therefore confront the problem of maintaining fresh replicas of data in unstructured peer-to-peer networks. We propose techniques that leverage path replication to support efficient lazy updates and provide freshness for cached data in these systems using only local knowledge. In addition, we show that locally available information may be used to provide additional guarantees of freshness at an acceptable cost to performance. Through performance simulations based on both synthetic and real-world workloads from big data environments, we demonstrate the effectiveness of our approach.

Acknowledgements

Khuzaima Daudjee, my supervisor. Raouf Boutaba and M. Tamer Özsu, the readers. IST, for making available machines with more than 80GB of RAM.

Dedication

This is for Maria.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many services on the Internet are founded on the idea that individuals can make small changes to data and have it available to others. For instance, Wikipedia allows users to modify parts of articles, people update their status on Facebook, and blogs and news articles provide areas for people to comment on content. The nature of these social media is distributed, for example, the creation of distributed, peer-to-peer (P2P) versions of web pages [8], content delivery [17], and social networks [3]. There is increasing attention given to such large-scale distributed data-driven content, often referred to as "Big Data" [43, 11].

Once data is distributed, it may not be stored in the location it is currently needed, requiring communication over the network for access. Caches reduce the amount of network traffic required in data-driven applications by replicating content closer to where it is accessed [16, 33], obviating the need to access remote data. While caching improves performance, unless the cached copies are synchronized when updated, stale data accesses inevitably occur. This compromises the usefulness and availability of data stored in wikis, blogs and other social media since users are generally uninterested in stale data and may require knowledge of the current state of the data in order to meaningfully update it.

Thus, maintaining cached data in the face of updates to provide freshness is an important problem in distributed systems. One option is to apply updates to replicated data in an eager (synchronous) fashion by ensuring that all caches have a copy of the update before applying it anywhere. While eager updates keep data fresh, they are prohibitively expensive [22], making it infeasible for maintaining data in large-scale distributed systems.

Proposals exist for quorums to limit the number of peers required for eager replication [13, 24]. Such schemes must know the number of replicas currently in the system, including those that are currently unreachable, which also makes it difficult to use in a cost-effective manner while introducing its own set of problems. We propose to manage and apply updates in a lazy (asynchronous) manner, in which updates are applied as they are received, then forwarded to other peers. Lazy techniques are feasible and cost-effective for distributing data to large numbers of copies [2]. However, lazy maintenance can result in data that is maintained at a slower rate, causing data freshness to lag behind in time. This problem is mitigated in structured P2P systems as the number and location of cached copies can be easily determined, simplifying transmission of updates. Ready access to knowledge about the state of cached copies does not exist in an unstructured P2P system, making techniques that require knowledge of the number or location of copies expensive.

1.2 Contributions

In this paper, we address the problem of maintaining cached data in unstructured P2P systems that are replicated on-demand, i.e. as a function of data requests that make up the query workload. As mentioned previously, examples of such replicated big data are blogs, wikis and the like used in the context of both social media and for e-science [43]. Results of queries are cached in the process of being returned to the peer that issued the query. These result replicas are maintained lazily, allowing the system to scale with respect to the number of peers, replicas and queries. Importantly, we show that data freshness can be provided at a reasonable cost, and that freshness can be traded for performance. This trade-off provides a spectrum of design choices for replica maintenance in unstructured distributed systems.

Specifically, we address the challenging issue of managing updates by leveraging path replication [29] to provide the following: (i) a memory of the path to control update propagation and increase search efficiency; (ii) a lazy update protocol to efficiently maintain cached data; (iii) guarantees of data freshness; (iv) tolerance of churn. We quantify freshness in terms of the number of updates and use this measure to study data freshness versus performance trade-offs in our system using both real and synthetic workloads. Additionally, we show that the trade-offs used to provide fresh data have better performance than the equivalent system without caching. Our protocols maintain cached data in a consistent manner such that updates are applied in the same order to all replicas in the system.

1.3 Outline

Chapter 2 presents related work, describing research that forms the base for the project as well as a discussion of alternative designs that were not adopted.

Chapter 3 describes the proposed system, which is based on a simple path replication system. Section 3.3 onwards describes how that system is augmented both to allow faster searches and the distribution of updates. In addition, improvements to deal with issues such as ensuring freshness despite lazy replication, churn, peer failure, and requests for older versions are described.

Chapter 4 provides measurements of the performance of the system. In addition to synthetic workloads, real workloads using traces from Wikipedia and Metafilter are used to show how the system behaves under two very different real scenarios. A qualitative description of the internal structure of the system provides context for understanding the results. Finally, chapter 5 concludes the paper.

Chapter 2

Related Work

While updates in unstructured networks are relatively unstudied, the networks themselves have been greatly studied, especially in the domain of search. This chapter therefore describes several unstructured networks, including both well-known and more recent networks. What few unstructured networks exist that support updates at all are given special attention.

2.1 Network Types

P2P networks fall into two broad classifications: *unstructured* networks in which there are few or no restrictions placed on peers, their connections, and the data they host, and *structured* networks, in which peers assume identifiers representing their place in the network and assigning responsibility for data based on their identifier. The problem of updates within a structured network is made simpler as a result of this assignment of responsibility. For instance, Han et al. examine updates in a social P2P network with the aim of minimizing the number of caches required while maintaining good performance [23]. Their algorithms exploit the assumption that the underlying network is based on the users' social structure, allowing them to reduce the number of caches by keeping copies local to peers.

Storing data in hierarchal trees of peers is common in structured systems. SCAN, Swarm and OceanStore all use trees to distribute updates among replicas [26, 37, 5]. Swarm also allows fine grained control of consistency by individual peers. While such approaches therefore appear similar to the directed graphs created in this work, the trees still are

arranged according to the identifiers assigned to the peers, allowing peers that are not part of the tree to easily find a leaf of the tree by contacting the neighbour closest to the tree based on a function of the neighbour's id. A tree in a structured system thus serves the purposes of reducing load on the root of the tree and avoiding latency and bandwidth issues when the peers are widely distributed, but does not make the data easier to find.

Therefore the bulk of discussion of related work concerns unstructured networks, especially those that are well known or address updates.

2.2 Overview of Approaches

Unstructured systems may vary significantly in their design. Several representative unstructured systems are therefore described, showing how network topology, search strategy, replica management and update management may be combined to form a single system.

2.2.1 Bubblestorm

Bubblestorm creates *bubbles* for search and replication [40]. These bubbles are generated by contacting a small fixed number of neighbours at random and sending the replica placement or search message to those neighbours. This process is repeated an arbitrary number of times. The result is similar to flooding (see Section 2.3.2), but because of the smaller fixed value, generates less load on the system. An update policy, with new versions existing as separate documents, is briefly described but not implemented or tested.

2.2.2 CoralCDN

CoralCDN is a P2P decentralized CDN intended to help websites deal with flash crowds, short bursts of traffic that greatly exceed typical traffic levels for the site, potentially exceeding the capacity of the server [17, 16]. Clients may choose to access data through CoralCDN rather than the origin server by appending ".nyud.net" to the host responsible for the content and requesting that URL. The Coral system then returns the requested data from its cache, fetching the data from the original server if needed. Data is replicated to multiple peers using *sloppy hashing*; the peers that receive replicas are those that have IDs with hashes similar to the hash of the key used to identify the data. Because the system is independent of the source of the content, it is unable to handle updates beyond expiring the cached copies and requesting them again after they expire.

2.2.3 FastTrack (KaZaA)

KaZaZ is a P2P filesharing system [28]. It consists of two levels of peers, ordinary peers and super peers. As a proprietary system, details about its exact operation are lacking, however it is known to contact super peers to find data and these super peers contain only an index indicating the peer or peers on which the data might be found. It therefore has a index on the super peers that is quickly accessible and copies all peers that choose to share the file (with a search creating a new copy that is made available). Because it is a filesharing system, replicas persist after a peer leaves and become available again when the peer rejoins the network. Updates are not supported.

2.2.4 Freenet

Freenet uses path replication along paths chosen based on a greedy search within a key space [8]. These walks are similar to random walks but have the property that two searches for the same data will perform the same walk if the network topology has not changed. Effectively, each peer uses the key of the data as an index into the list of neighbours. In addition, bloom filters are used to search the neighbours along the walk. Freenet inserts the initial document in the same way, establishing an initial path [6]. Earlier work states that updates are desirable but not implemented due to complications associated with ensuring all copies are updated [8]. This remains true for the latest version of Freenet in that versions are identified as separate documents with each document sharing a common name and a version affix [7]. To find the latest version of a document, a Freenet client issues multiple searches for the same base name with increasing version numbers, continuing until it does not find a document with a greater version number [18]. Because versions may be missing, the search continues for several values after the last confirmed version to ensure that the version is truly the most recent. This results in extra traffic and delays before presenting the result yet does not actually guarantee that the most recent version is found as it is possible that the last known version and the most recent version may be separated by an interval greater than that covered by the search. Apart from this basic support, changes to Freenet's protocol have focussed on security and privacy rather than updates [15].

The approach used by Freenet to handle new versions of data is also used by or suggested for other systems [40, 27].

2.2.5 Gia

Gia is a filesharing system intended to overcome the scalability problems with early versions of Gnutella [4]. It achieves scaling by having peers with greater capacity form more connections thereby placing load on peers in proportion to their network capacity. It avoids traffic issues associated with flooding by performing random walks that are aimed at the peers with the greatest number of connections. As a single random walk is unlikely to find a peer with the desired data quickly, an index of the data available on each peer is replicated to all peers within one hop to increase the presence of data items on the network. As the peers with the most connections also have the largest indexes, the random walk is thus greedy. Like most other filesharing systems, no provision is made for updates.

2.2.6 Gnutella

Gnutella is a simple filesharing system [9, 14]. It performs searches by flooding the network with a short time-to-live (TTL). Later versions improved scalability by adding super peers that contain indexes to speed search and reduce the amount of traffic by allowing a lower TTL since more results could be achieved from a smaller set of peers. Replicas are placed on the peers that initiate queries at the discretion of the user. Like other filesharing systems Gnutella does not support updates.

2.2.7 JXTA

The JXTA protocol combines both a structured system (DHT) and an unstructured system to handle churn. The expected location of a replica is looked up in a hash table and the peer corresponding to that location is contacted. If that peer does not reply or does not have the data, the neighbours of the expected peer according to the hash table are checked. If those peers are also unavailable or do not have the data, then a random walk is performed from a peer near where data was expected to be found. Replicas of the index are placed at the peer associated with the hash value and those up to k spaces away from it in the hash table. The value of k is not specified by the protocol. All entries in the DHT point to superpeers. Updates are not described as part of the protocol.

2.2.8 P-Grid (with extensions)

P-Grid closely resembles a distributed hash table in structure [1]. Peers maintain a partial index of the data present in the system, with references to one or more peers that contain the rest of the index. As the index is based on key prefixes, peers use depth-first search to navigate this structure. Indexes are updated when peers contact each other, increasing in specialization and decreasing in size over time. Adding a data item to the index is done by issuing queries to find a peer associated with that part of the index and informing that peer that the data is present. Because the prefix partitioning is random, a data item may not be known to all peers containing an index that should reference that data. To find all replicas, searches would need to be made from multiple initial peers. Datta et al. extend P-Grid to support updates through a gossip-based protocol [12], however they describe their extension as generic and applicable to systems other than P-Grid. They perform probabilistic flooding of known replicas when an update occurs. Peers that do not receive updates or rejoin the network pull updates to ensure that they are not missed. The authors assume that update conflicts do not exist and that updates are sparse enough that one will successfully complete before another update to the same data is initiated.

2.2.9 Quorum Based Updates

Some exploration has been done concerning adding quorums to existing systems as a method to support updates [13, 25]. These methods do not specify any particular infrastructure for search or replication, but instead build on top of any existing unstructured network. Quorums work on the following principle: contact multiple peers (a *quorum*) for read and write requests. A read request returns the most recent version from all peers that respond, while a write request writes the new version to all peers contacted. A minimum number of responses are required to form a quorum; if an insufficient number respond, the read or write fails. By ensuring that the number of peers read must overlap with the number of peers written, the latest copy of any piece of data may be found even if one or more peers are currently unavailable. In the case of reads, failure could be relaxed to a warning of stale data if desired. While contacting the quorums themselves can be made efficient, they do have one limitation noted in [13]: peers must be aware of all possible copies in the system, including those present on peers currently offline if those peers will rejoin the network and contribute their copy. Discovering this list of peers with copies can be expensive, implying that quorums work better when the number of replicas may be determined without needing to contact the entire network.

2.2.10 Unstructured Distributed Hash Tables

An unstructured distributed hash table (UDHT) replicates the distributed hash table API to unstructured networks [34]. A better name for it is unstructured distributed map. Structurally, it is similar to Gia (Section 2.2.5) with the addition of automatic placement of additional replicas through searches that terminate after a random number of hops. Also, as the goal was to emulate a hash table API, no statement is made about replication on search, though if it behaves similarly to structured DHTs, it is likely that no replicas are created as a result of a search. Updates are not supported.

2.3 Network Structure and Replica Placement

The most basic features of P2P systems containing data are how the network is connected, the location of resources in the network, and the method for locating those resources. Any taxonomy of these systems therefore begins with these essential features. In data-centric systems like those described in Section 2.2, the primary resource is the data and replicas of the data. Thus the location of resources may be defined by the placement of replicas.

All cells in the tables that follow may have the value *Unspecified*, which means that the subject is not covered for that system. As some systems described are not complete, this does not represent the absence of the feature from the system. For example, Lv. et al. provide results for multiple network topologies but do not specify a specific topology for use with their algorithm [29].

2.3.1 Topology

A simple description of how the graph is connected. Even though the network is unstructured, peers may preferentially connect to peers with desirable characteristics such as available bandwidth or low latency, allowing for several possible topologies.

- Proportional: Peers establish a number of connections proportional to their bandwidth/processing power. The result generally approximates a scale-free/power law network.
- Random: Peers are connected randomly. The peers usually try to pick neighbours so as to approximate a uniform distribution.

Table 2.1: Network Structure and Replica Placement

Related Work	Topology	Search	Initial Replica Placement	Search Replica Placement
Bubblestorm [40]	Proportional	Random Tree	Random Tree	None
FastTrack (KaZaA) [28]	Superpeer	Contact Superpeer / Proprietary	Index on Superpeer	Search Origin
Freenet [8]	Small World / Random	Greedy Search	"Request" Path	Request Path
Gia [4]	Proportional	Random Walk	Index on Neighbours	Search Origin
Gnutella 0.4 [9, 14]	Random	Flooding	None	Search Origin
Gnutella 0.6 [14]	Superpeer	Flood Superpeers	Index on Super Neighbours	Search Origin
JXTA [20]	Superpeer / Small World	DHT and Random Walk	Index in DHT	None
Lv et al. [29]	Unpsecified	Multiple Random Walk	Unspecified	Request Path / Visited Subset
P-Grid [1, 12]	Random	Greedy Search	Index on "Request" Path	Unspecified
Quorum Based Updates [13]	Unpsecified	Unpsecified	Unpsecified	Unpsecified
Rashkovits and Gal [35]	Client-Server	Not Applicable	None	Possibly At Cache Server
UDHT [34]	Proportional	Multiple Random Walk	Random Walk Subset / Index on Subset Neighbours	Unspecified (likely Search Origin)
Thesis	Random	Multiple Random Walk	None	Request Path

- Small World: Peers connect to "friends", this may be calculated in terms of interests or in terms of trusted contacts (Freenet). When the "friends" are defined in terms of the data stored, as in [23], the network is effectively structured.
- Superpeer: Peers are placed in two categories: ordinary peers and super peers. Generally ordinary peers connect only to super peers and super peers form a random network among themselves.

2.3.2 Search

Search is the method a peer uses to retrieve information it does not possess. Retrieval does not imply the creation of replicas as a result may be discarded after use.

- Flooding: If the item is not found on the current peer, send the request to every neighbour other than the one that sent the request. If a peer has already forwarded a specific request, it does not do so again.
- Random Walk: If the item is not found on the current peer, send the request to one randomly chosen peer.
- Multiple Random Walk: As random walk, but the initial request is sent to multiple peers. Subsequently contacted peers only send to one peer.
- Greedy Search: If the item is not found on the current peer, forward the request to a neighbour calculated to be most likely to contain the data. The calculation is based on local knowledge of the neighbours and the characteristics of the query. It is possible for the choice to fail, in which case the next best neighbour is chosen. Unlike most other search methods described, only exact matches are possible.
- DHT: Distributed Hash Table. A hash function calculates the peer that should have the data and that peer is directly contacted. As with Greedy Search, only exact matches are possible.
- Random Tree: As multiple random walk, but each peer contacted sends the request to multiple neighbours.
- Proprietary: While a search method exists, its exact form is unknown. In the case of the FastTrack system, the method used is further obscured by the presence of a reverse engineered client that is known to behave differently than the proprietary client when searching as an ordinary peer.

2.3.3 Initial Replica Placement

Some schemes replicate data items when they are created and when a peer containing them joins the network, before any requests have been made for that item. This is generally done to increase availability of the item.

- DHT: A Distributed Hash Table. A hash function calculates the peer that should have the data and that peer is directly contacted.
- Index: Instead of placing a copy of the data, store the address of the peer containing the data. The index may include keywords, hashes, file names, and other metadata.
- Neighbours: All neighbours of the peer receive a copy. Super Neighbours are neighbours that are also super peers.
- Random Tree: The peer places replicas on a subset of its neighbours, which repeat the process until an endpoint is reached.
- "Request" Path: Copies are placed on the peer or peers that would be contacted if the originator of the data were to search for it. This is distinguished from the Random Tree by being a deterministic set, and from the DHT in that the set of peers that receive replicas depends on the peer that originates the data and because intermediate peers receive the data, not just the peers representing the endpoint of the search.

2.3.4 Search Replica Placement

Many protocols increase the number of replicas when an item is requested. This section describes the policies used to choose the locations of additional replicas.

- Cache Server: The first peer contacted by the peer requesting the object stores a copy. This protocol is associated with systems that cache data at predictable locations and is typically found in structured systems using hash tables to find data.
- Request Path: Every peer in the discovered path from the requesting peer to the one that contained the result stores a copy. Lv et al. first demonstrated the case for using path replication in unstructured networks [29]. Yamato et al. modify it to reduce load on peers with high connectivity [45]. Path replication has also been used to cache indexes [30, 44], which is analogous to our path cache.
- Search Origin: The peer that started the search for an item stores a copy of it.

Table 2.2: Replica Management

Related Work	Maintenance	Persistence	Removal
Bubblestorm	Copy on Join	No	Flush
FastTrack (KaZaA)	None	Replicas	By Use
Freenet	None	Yes	LRU cache
Gia	None	Replicas	By User
Gnutella 0.4	None	Yes	By User
Gnutella 0.6	None	Replicas	By User
JXTA	None	No	TTL for Indexes
Lv et al.	Unspecified	Unspecified	Random
P-Grid	Split Index on Join	Yes	Unspecified
Quorum Based Updates	Unspecified	Unspecified	Unspecified
Rashkovits and Gal	None	Yes	Cost-based expiry
UDHT	Copy on Leave	No	Unspecified
Thesis	None	No	Cache

2.4 Replica Management

In addition to placement, dealing with the effect of churn and peer failure and handling removal are also common problems associated with data, even in the absence of updates.

2.4.1 Maintenance

Due to churn, the proportion of replicas and/or the number of replicas may change with time. Some systems take special action when peers join or leave the system to maintain the number or proportion of replicas in the system.

- Copy on Join: A joining peer contacts a random subset of peers and replicates all data for which those peers are masters.
- Copy on Leave: When a peer is detected by a neighbour to have left, the neighbour tries to replicate copies of all data that were on the leaving peer and itself to additional peers.
- Split Index on Join: A joining peer associates itself with the complement of the index contained in the first peer contacted. The joining then uses information available from the first peer contacted to construct the index.

- None: No special action is taken due to churn.

2.4.2 Persistence

When a peer leaves a network, it may retain information about the replicas in the system.

- No: No information about replicas is persisted.
- Replicas: Replicas present on the peer are persisted, but any additional information (e.g. indexes) is not.
- Yes: All information about replicas is persisted.

2.4.3 Removal

The policy for removing replicas in the absence of churn. The policy is usually based on the storage capacity of the peer or the freshness of the data.

- By User: The user controls which items remain in the cache. Items stay until the user removes them.
- Flush: All items are evicted from the cache at regular intervals. The peer then fetches new copies from the masters.
- LRU cache: The item that has been least recently used or requested is removed when there is not enough space for a new item.
- TTL: Each item includes a time-to-live value, which is an expiry time. After the TTL is reached, the item is removed.
- Random: When the cache is full, a randomly selected item is removed to make space for the new item.
- Cache: Items are evicted from a cache to make space for new items. The algorithm used to select the items to remove is unspecified.

Table 2.3: Update Management

Related Work	Source	Propagation	Freshness
Bubblestorm	Master	New Document Id	Unspecified
FastTrack (KaZaA)	Not Supported	Not Supported	Not Supported
Freenet	Key Holder	New Document Id	Version Probing
Gia	Not Supported	Not Supported	Not Supported
Gnutella 0.4	Not Supported	Not Supported	Not Supported
Gnutella 0.6	Not Supported	Not Supported	Not Supported
JXTA	Master	Change Original	TTL
Lv et al.	Not Supported	Not Supported	Not Supported
P-Grid	Any Peer	Probabilistic Flooding/Pull Requests	Pull Interval
Rashkovits and Gal	Master	Pull Requests	Cost Model
Quorum Based Updates	Any Peer	Push	Quorum
UDHT	Not Supported	Not Supported	Not Supported
Thesis	Master	Push	Multiple

2.5 Update Management

Update management is the process of initiating, applying and propagating updates. As updates are little studied in unstructured networks, many of the systems described do not support updates. These are marked as *Not Supported* in the table.

2.5.1 Source

The update source identifies the peers in the system with the ability to apply new updates. Peers not listed as a source can only receive and forward updates from other peers. However, they may possibly request a peer that is a valid source to perform the update on its behalf.

- Any: Any peer that holds a replica may update it.
- Key Holder: The ability to update an item is associated with an asymmetric encryption key. Peers that have the private key may update the data.

- Master: All updates originate with the master copy of the data.
- Not Supported: Updates cannot be made in this system.

2.5.2 Propagation

The method by which an update is distributed to other peers/replica holders from the peer that initiates the update.

- Change Original: For systems in which no replicas are stored on other peers, just indexes to it, updating the original copy completes the update.
- New Document Id: A new document is created with a new id. This document is distributed in the same manner as any other new document. The old version of the document remains in the system.
- Probabilistic Flooding: The information is propagated via flooding, but only a random fraction of the neighbours are contacted by each peer. Unlike the *push* method, a peer known to have a replica may not be contacted for an update.
- Pull Requests: Other peers are expected to pull the update from a system that has a newer copy.
- Push: A peer with an updated copy pushes it to the peers it knows are holding replicas.

Push and pull updates can be combined. In such cases, they have been labelled as gossip or epidemic protocols [32].

2.5.3 Freshness

The propagation method used by a system may not guarantee that every peer receives the latest version. Therefore peers often have additional methods to ensure that stale copies are not returned as the result of a query. Typical solutions rely on local knowledge or modify the query.

- Cost Model: The freshness of the copy is estimated and weighted against the cost of retrieving a possibly fresher copy from another peer or the master copy. The request specifies the relative priorities of freshness and latency.

- Pull Interval: Pull a fresh copy if an update has not been received after a locally specified duration. This approach establishes a time limitation on how stale an item can be.
- Quorum: A quorum is used to control read and write requests [13, 24]. A transaction will complete only if a calculated number of peers (depending on quorum method) it. This allows for eager replication and effectively perfect consistency while reducing the overhead for applying updates. While quorums are easy to track in a structured P2P network where data items might be identified by keys, they are much more difficult in an unstructured network – for instance, Henry et al. require that the entire network be contacted to generate an initial list of peers that could participate while Vecchio and Son provide no explicit algorithm for finding the list of interested peers. Because of the lack of stability in P2P networks, the quorums in the papers are chosen to only probabilistically overlap for conflicting operations.
- TTL: The copy expires after a specified interval, requiring the peer with the cached item to refetch it [39, 38]. Different from Pull Interval in that the time is set by the master instead of the client.
- Version Probing: Search for versions after last known version. If any are found, repeat. May cause issues if client has not searched for a long time as intermediate versions may no longer be found in network.

2.6 Summary

As table 2.5 shows, updates are not generally supported by unstructured networks. Two of the methods mentioned as supporting updates, Bubblestorm and Quorums, are incomplete: Bubblestorm has no method for identifying which result to return and Quorums do not provide a mechanism for finding peers that have copies. JXTA supports updates by not having replicas at all, and Rashkovits and Gal's method assumes centralized servers. Only Freenet and P-Grid truly support updates and both of them do it by restricting searches to key equality. Therefore an opportunity exists for a system that allows for both a general search strategy and updates.

Chapter 3

The Unstructured System

This chapter describes a simple unstructured system and a set of enhancements that combine to support updates using only local knowledge. These enhancements are capable of providing defined levels of freshness, from best-effort to absolute guarantees. Consideration is also given to maintaining updates in the presence of potentially adverse events like churn and peer failure.

3.1 Assumptions and System Model

The fundamental model for most information available via the Internet is that of content that is distributed to users. Content delivery networks (CDNs) work to make this information more available by providing caches so that users may acquire the data more quickly, especially in the cases where the original provider would be overloaded or is physically distant. In many cases however, the information distributed is not absolutely stable but changes over time. A CDN must therefore provide a mechanism for replacing old content with the new version.

This system therefore models a participatory CDN, in which the providers and consumers of data cooperate to share the load. As such a system, it must be able to provide up-to-date content to those who request it. This may be expressed as a high probability that the most recent version of the data will be retrieved. Therefore, updates must be supported and distributed efficiently. Efficient deletion is not required as cache turnover will eventually force deleted items out of a cache. Deletion of the original copies will eventually cause removal from all caches without any special effort.

As we expect providers will wish to retain control over their data, we assume that updates are applied at a single peer and that the order in which updates are applied is determined by this peer, identified as the *master* peer for that data item. A peer in the network may be the master for any number of data items, but only one peer is the master for a particular data item.

We assume that providers are interested in maintaining their data; consequently we assume that churn among master peers will be minimal as they will choose not to leave to ensure their data remain available. Conversely, peers which are not masters may undergo greater churn, possibly behaving in a manner similar to that observed in existing P2P networks, with most peers remaining for very short periods of time [36]. Finally, we assume peers do not exhibit malicious behaviour. They may crash, but all messages sent are trustworthy.

Critically, the network modelled is an unstructured network. Such a network allows peers to join and leave with little coordination as peers simply connect to those that are available rather than having to find a unique location in the structure and then assuming the responsibilities associated with that location. In exchange for simplicity in joining and leaving the network, data becomes harder to find and thus maintain. The network topology used as a base for the model is a regular random graph, using the SwapLinks algorithm described by Vishnumurthy and Francis to build the graph [42]. As no assumptions are made to exploit this topology, the algorithms used will work successfully on any network topology, though the time to complete operations may change.

3.2 Search and Replication

Searching and replication are inherently related operations. The choice of where an object is replicated affects the efficiency of the search strategy and some search strategies impose conditions on where caches are located. For search and replication, the system uses multiple random walks and path replication respectively. In an unstructured system, these choices provide a low network overhead, allowing scaling, while still providing reasonably fast search times [29]. Apart from the initial copy on the master peer, no copies of the data are placed on the network when the data is created or when the master peer joins the network.

3.2.1 Random Walks

Random walks provide a message efficient method to find items. A random walk is performed by choosing a single neighbouring peer at random to forward the query to and repeating until the desired item is found or the search is cancelled. Apart from the peer it just came from, a query may be forwarded to any neighbour. Performing multiple simultaneous random walks decreases the time required to find existing data while maintaining a limit on overhead due to network traffic. Regardless of the number of walks, the number of peers contacted, and thus the number of messages, is expected to be N/c where N is the number of peers and c is the number of peers with a copy of the data.

There are two simple methods to prevent random walks from continuing indefinitely. The first method is to associate a time-to-live counter with the random walk that decreases after each hop. When the counter reaches zero, the walk is terminated. The second option is to have peers check with the originator of the search to see if the walk should continue. The peer that originated the query may reply with a continue or cancel message to control the duration of the random walk. To minimize overhead, both in quantity of network traffic and time required to find the data, this check is not performed at every peer. Instead, it is performed at every k^{th} peer, where k is an arbitrarily chosen integer. While this second method adds additional traffic and time to find a result, it also allows the other random walks to terminate early once a result has been found. Further, it also avoids issues associated with selecting the correct value for the TTL [29]. As an additional benefit, a user can cancel a search and have its overhead quickly removed from the network. As a result, the system uses checking as a means of controlling the duration of the random walks.

3.2.2 Path Replication

Path replication is based on the idea that when a query finds a result, that query has been forwarded by one or more peers including the originator of the query. If the result is then forwarded back through the chain of peers to return to the originating peer then each peer in the chain can cache the data as well as forward it. The resulting set of new locations forms a path on the network overlay graph.

Path replication, along with some additional conditions, allows the placement of copies in quantities proportionate to the square root of the frequency of access (*square-root replication*). This ratio is proven to minimize the average number of messages across all queries for uneven popularity levels [10]. As peers have limited storage capacity, replicated items

are not stored permanently. To achieve square-root replication, the cache eviction policy must select data for removal without considering the query rate, therefore policies like least-frequently-used and least-recently-used cannot be used for cache eviction. In addition, since we want to set an absolute limit on the size of the cache, cache expiration policies based on a fixed clock time are not suitable. Two familiar examples of policies that do satisfy the requirement of independence from the query rate are random eviction, where items to be removed are chosen at random, and first-in-first-out, in which the copy first added to the cache is also the first to be removed.

3.3 Cache Structure

Path replication as described in Section 3.2.2 does not track changes to data. It places copies of data items as seen at the time they are requested on additional peers and will replicate additional copies from the caches on peers without regard to freshness. As a result, it is possible for stale copies to continue to proliferate in the system. While such behaviour is acceptable at the master, which always has the latest version, it is not a solution for maintaining cached copies. To address this deficiency, the cached copies of data are augmented with additional information about that data on a local level. This information will be later exploited both to efficiently provide freshness guarantees that a user can request and to push updates to peers with cached copies without maintaining global system state or using flooding, which is very expensive[29], to send updates.

3.3.1 Data Caching

A successful random walk provides two pieces of information about the query. The first is the data satisfying the query and the second is the path used to find that data. While path replication takes advantage of the path to create replicas, it does not store any information about the path. We therefore propose replicating information about the path in addition to the data satisfying the query (Algorithm 1). As a result, along with the data satisfying a query, each peer involved in path replication also stores the following metadata: (i) a unique identifier for the data, (ii) a version number assigned by the master, (iii) the estimated distance in number of hops to the master, (iv) the identity of the neighbour from which it received the data (its *parent*) and, (v) the identity of the neighbour to which it sent the data (its *child*) (Algorithm 1, lines 9-17). For example, after the random walk in Figure 3.1a, three cached copies and two new parent links are created (Figure 3.1c). Peers may have any number of children, limited only by the total number of neighbours. Because

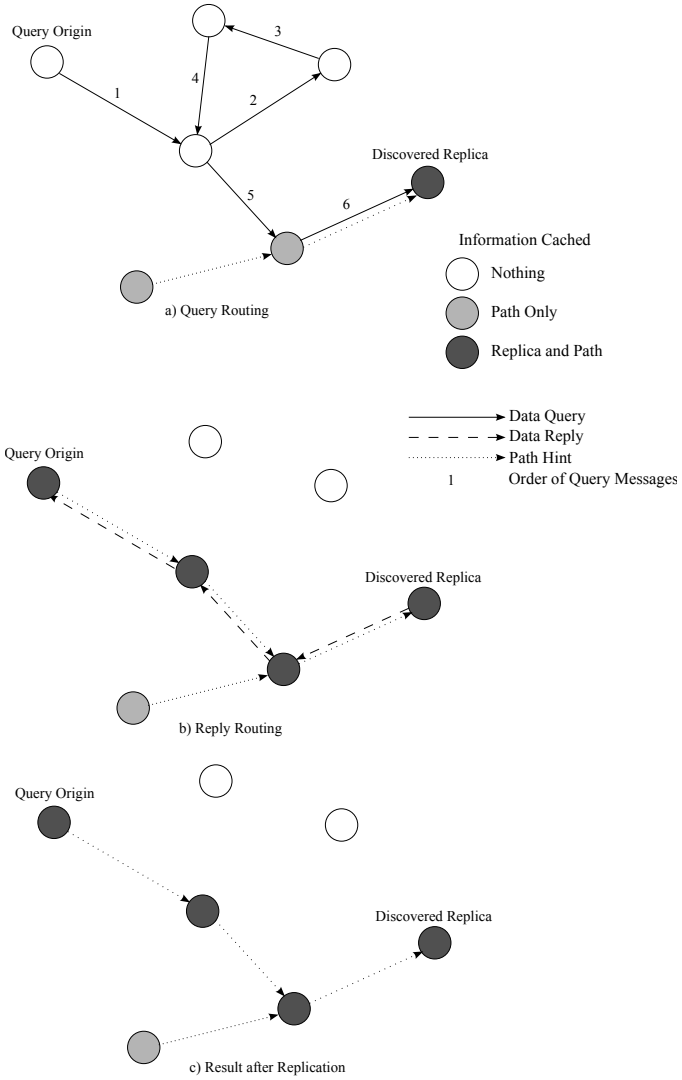

Figure 3.1: Random walks and path replication. In (a), a random walk traverses peers, including a cycle, until it discovers a replica. Due to the presence of a path cache, the last hop of the query is not chosen at random. The reply routes around cycles by having each peer track only the first peer by which it is contacted (b). The peer contacted by messages 1 and 4 in (a) thus only forwards the reply to the peer that sent message 1. In (c), the query is complete and all peers contacted by the reply now have cached copies and have added the source of the reply as a path hint and the destination as a child (not shown).

multiple random walks are used, it is possible that more than one walk will succeed. In that case, path replication will occur along all successful routes.

It is impossible to safely abort a random walk or the path replication initiated by that walk until a result has been returned to the peer that originated the query. This is because if a peer along the route fails before the replication has reached it, the peer that issued the query will never receive a result. Therefore a peer may participate in more than one path between the querying peer and the master peer. When this occurs, the peer is remembered by both of its parents, but it remembers only the parent with the shortest estimated distance to the master. As cycles in the graphs formed by following parent or child links cause unnecessary message overhead when messages are passed along them, cycles within each random walk are eliminated (e.g. the query cycle in Figure 3.1a is not followed in Figure 3.1b). Cycle elimination is performed by having each peer contacted during a random walk forward a successful result to only the first peer that contacted it during that walk (*query_source* in Algorithm 1). Each random walk records the list of first peers independently, even if they represent the same query from the same peer. In the absence of cache eviction, the edges connecting the parents to children thus form a directed acyclic graph that approximates a tree rooted at a single master copy of an item.

3.3.2 Path Caching

While the data cache allows updates to propagate by following child links as described in Section 3.4, it is inevitable that cache eviction will cause breaks in the graph. However, the metadata associated with each data item, especially the parent and child links, require less space to store than all but the smallest data items. Since this means that a peer may store the metadata for many more data items than full replicas in the same amount of space, the system additionally includes a *path cache* containing only the metadata described in section 3.3.1. We refer to the cache containing the data items as the *data cache* to differentiate the caches. When data is purged from the data cache, all the metadata is copied to the path cache instead of being removed. This allows the graph structure to be maintained for a greater period of time, helping to keep results fresh. It also means that when a break in the graph does occur, it may isolate only copies of the metadata rather than a full copy of the data, further preventing stale results.

In addition, the path cache is used to improve the performance of random walks. When a random walk visits a peer containing information in its path cache that is relevant to the query, the walk next visits the parent peer stored in the cache instead of a random peer (as in the message labelled 6 in Figure 3.1a). Such hints help a walk reach the relevant

Algorithm 1 Caching during Path Replication

```
1: cache ← find_in_data_cache(reply.id)
2: if cache is empty then
3:    no_data ← true
4:    cache ← find_in_path_cache(reply.id)
5:    if cache is empty then
6:       cache ← new_cache()
7:    end if
8: end if
9: if no_data or cache.version < reply.version then
10:   cache.data ← reply.data
11:   cache.version ← reply.version
12: end if
13: if cache.distance > reply.distance then
14:   cache.distance ← reply.distance
15:   cache.parent ← reply.source
16: end if
17: cache.child.add(query_source)
18: reply.distance ← reply.distance + 1
19: send(query_source, reply)
```

master but do not necessarily guarantee success since a peer identified in the path cache may have removed the path information from its cache, left the network, or crashed. In such cases, the walk returns to random selection at the next hop to continue the search.

3.4 Updates

As stated earlier, each data item is associated with a master peer that maintains a copy of the data item and manages all updates for that item. It determines the order of updates and records a strictly increasing version number on successive updates. As the master copy, it has no parent for a data item, but otherwise tracks exactly the same information as is present at every replica. Therefore, while other peers may initiate an update, it is always applied at the master copy and the updated version propagated to other peers.

New versions are pushed lazily along the edges of the directed graph produced by following the child links in the caches (the *child graph*). Peers that contain a copy of the data in their data cache update their copy and forward the update message to their children, while peers that have a copy in the path cache forward but do not store the data item (Algorithm 2, lines 17-23). As the directed graph may contain more than one path to the same peer, peers check the version of the update before applying or forwarding it. If the version in the update message is the same or older than the present version, the update message is discarded (Algorithm 2, line 11). Therefore, these version numbers also ensure that all updates are applied in the same order as that set by the master, ensuring a globally consistent order for updates. Since it is possible that an update may be missed by a peer, each update must contain a complete copy of the data to ensure that any missed updates do not affect the final result.

The master acts as a source vertex for the complete child graph associated with a data item. Therefore, unless the graph has been cut due to a cache eviction from a non-sink[1] peer or a peer has failed, an update is guaranteed to reach every copy of the item. Each peer contacts only the subset of its neighbours known to have had a copy of the data previously. This ensures that peers that have never been part of a search for a data item will not be contacted for an update.

A lack of an update to a copy could occur because the update message did not have time to propagate to the copy returned or because the cached copy is not on a complete path to the master due to cache eviction or peer failure. Because the path cache does not

[1]Recall that a non-sink vertex in a directed graph is one that has at least one edge that leaves that vertex. This means that the peer has at least one child neighbour associated with the data item.

Algorithm 2 Update at a peer

```
1: cache ← find_in_data_cache(update.id)
2: if cache is empty then
3:    cache ← find_in_path_cache(update.id)
4:    if cache is empty then
5:       return
6:    end if
7:    includes_data ← false
8: else
9:    includes_data ← true
10: end if
11: if cache.version < update.version then
12:    if cache.parent <> update.parent then
13:       cache.child.add(cache.parent)
14:       cache.parent ← update.parent
15:       cache.distance ← update.distance
16:    end if
17:    if includes_data then
18:       cache.data ← update.data
19:    end if
20:    cache.version ← update.version
21:    for all child in cache.children do
22:       send_update_message(child, update)
23:    end for
24: end if
```

rely on random eviction and is relatively large, such events are expected to require peer failure to occur.

There is currently no method for a cached copy to request a newer version if it no longer receives update messages along a path from the master. However, a path may be reformed by queries that pass through one of its parents on the partial path that remains, which would allow it to receive the next update message. Usefully, the common policies that do allow data to persist indefinitely, like least-recently-used, also violate the restrictions required for square-root replication, while those that do not allow data to persist indefinitely conform to the replication restrictions. For the one exception to the above, random eviction, the probability P that an item will be removed from the data cache after k queries for items not in the cache is: $P = \sum_{i=1}^{k} \frac{1}{N} \left(\frac{N-1}{N}\right)^{i-1}$ where N is the number of items in the data cache. This is the sum of the probabilities that an item will be removed at time step i, given that it has not been removed at any previous time step. Therefore the probability that the item will be removed from the cache increases with the number of queries for other items and will be removed from the data cache with a probability of at least 0.5 after $N/2 + 1$ such queries have been made and after sufficient queries will be removed at any probability ¡ 1. It may therefore be considered to effectively not retain any data permanently. The lack of data permanency means that all stale data will eventually be replaced in the system. The replacement data itself may not be fresh, however, unless further steps are taken to avoid propagating stale data or deal with disconnected pieces of the graph. As a result, techniques for both approaches are presented in sections 3.5 and 3.6.

3.4.1 Update Cost

The structure of the directed graph of child links for data determines the cost of an update. This structure is dependant on the queries made for data as described above. For all graphs built when only one search succeeds, the structure of the graph is a tree (or a forest if the graph has become disconnected). In such cases, from the definition of a tree, the number of messages sent to propagate an update is $n - 1$ where n is the number of peers in the connected portion of the graph that includes the master. When only one random walk is issued per search, the above statement is a complete characterization of the cost of an update.

As described in section 3.2, multiple random walks are used to decrease the time required to find data. In this case, there are up to k paths to a peer from the master, where k is the number of random walks issued, and so the time is bounded by $k * (n - 1)$ messages.

This bound may be improved in two ways. First, k may be replaced by the average number of successful searches. This number is low for unpopular data and high for popular data as increasing the number of copies in the system increases the probability that multiple walks will find a copy around the same time. Second, it is impossible to have more routes than are physically present in the underlying network. Therefore, k has a maximum value equal to the maximum number of neighbours each peer has. When the data is popular and k takes this maximum value, which happens when the number of random walks is significantly higher than the number of neighbours, the result is equivalent to flooding. To maintain the total number of random walks while avoiding flooding, random walks may be staggered in time, issuing a small number of walks initially, then issuing more if results are not returned quickly. Since popular data is expected to return results more quickly due the the larger number of cached copies, this method will usually begin fewer walks with more popular data, reducing the number of unneeded paths and avoiding flooding. For reasons stated later in section 3.6.2, this initial number of walks should be greater than one even though it leads to some redundancy in very popular networks.

3.5 Guaranteeing Freshness

Data is considered fresh on a peer when there are no versions of that data in the system that have been updated more recently. We formally define this standard notion of freshness as follows:

Definition 1 (Freshness)**.** Data returned by a query is *fresh* if and only if at the time the query discovers version v_k of the data, the version at the master is v_k, where k is an integer denoting the kth version.

The first result found and returned to the peer that issued a query represents the shortest search time to find a result. However, this result may not be fresh due to the time required to propagate update messages and the possibility that the child graph has become disconnected. While the total (serialization) order for all updates to a data item is determined by its master, the system as is provides only best effort guarantees for data freshness to queries that do not contact the master. Thus, we provide freshness guarantees by proposing to use two algorithms that augment the basic system. These freshness guarantees can be specified by a peer in its query.

3.5.1 Master Forcing

First, the parent links at each peer can be used to find the fresh result at the master. This is equivalent to ignoring the cached results and using the path cache to provide hints for the random search. This approach guarantees that the query will see fresh data when it reaches the master peer. However, as discussed earlier, cache eviction can cause breaks in the chain to the master, forcing a second random walk to start where the first left off. As the probability of a random walk being the same as a previously discovered path decreases exponentially with length, the presence of such walks means that we may not always benefit fully from the path cache. However, even in the worst case, it should provide a connected region surrounding the master that will reduce the length of the walk. The trade-off here is that while fresh data is obtained, this increases the load at the master peer thereby increasing the possibility of creating a bottleneck for popular queries.

3.5.2 Session Guarantee

After a data item has been evicted from the data cache at a peer, it is possible for a subsequent request for that data from that peer to retrieve a copy that is older than the previously held copy. However, because the path cache maintains knowledge of the version of the data, it is possible to detect this event occurring in some cases. Further, even when the information about the data is purged from the path cache, it is possible for an application built on top of the system to remember which version of the data was retrieved. Thus, the second guarantee that we provide is that a peer can request a version of data that is guaranteed to be no older than the last known version accessed by the same peer.

Definition 2 (Session Guarantee)**.** Data meets a *session guarantee* for a peer when if that peer knows it has seen version v_i at time t_x, then for all times $t_n > t_x$, that peer sees version $v_k, k \geq i$ of that data.

When a request for a session guarantee is made and a cached copy of the requested data item is found, the version of the cached data is compared to the minimum version in the request. If the version requirement is not met, then the peer forwards the request to its parent as though it did not have a copy of the data. We further extend this guarantee to intermediate peers during a search. If an intermediate peer can use its path cache to improve the search, it also updates the minimum version if it is aware of newer version of the data. In this manner, a peer guarantees minimum versions of data not only for itself, but also for any peer that contacts it looking for data. Because search would normally stop

when the first replica is found, this guarantee has an effect only when the data is found in the path cache first.

In addition to the guarantees available above, we can also exploit the presence of multiple random walks in the system to provide a greater probability of freshness without a guarantee. As discussed earlier, it is impossible to terminate any random walk until a result has been returned to the peer that issued a query. Thus there may be other random walks that have found different versions of the same data, some of which may be more recent than the first version returned. It is therefore possible to wait for additional results to be returned before electing a final result. However, since the time until the last result returned may be double the time for the first result to return without any guarantee as to the quality of that data, we do not explore this option in this paper other than to note that a short wait for additional results may occasionally provide more recent data.

3.6 Graph Maintenance

As described so far, the system meets all of the requirements set out by the model. It creates caches to distribute load and applies updates to the caches. Further, it provides mechanisms to increase the probability or guarantee that a retrieved copy is fresh. However, the system as described does not address issues such as churn, nor does it try to maintain a connected graph after the caches fill and items start to be evicted. The system is therefore augmented to maintain the value of the caches after they become full as well as mitigating the effects of churn and peer failure.

3.6.1 Cache Eviction

To ensure that updated versions are successfully propagated to all copies in the data cache, the child graph must remain connected. To this end, we explore several eviction policies for the caches. These are *random*, *first-in-first-out* (FIFO), *least-frequently-used* (LFU), *least-recently-used* (LRU), *sink-first*, and *root-first*. Note that the LFU, LRU and sink-first policies break the square root replication requirement described in Section 3.2.2. They are included to test the possibility that despite the increase in search overhead they may allow more searches to return fresh results.

The *sink-first* policy is motivated by the idea that copies at the end of a path are potentially the least valuable. Update messages reach them last, and there are the greatest number of hops between them and the master, increasing the chance for the path to become

disconnected. Moreover, sinks represent peers that can be removed without disconnecting the graph, suggesting that removing them should help keep the graph connected. For this policy, the number of child peers associated with a data item are considered when choosing an item to remove. If there are no children associated with the data item at a peer, then removing the path information associated with that data item cannot break any paths to a copy in a data cache and therefore one of these items is removed at random. It is possible that there are no sinks in the path cache. In this case, we revert to the least-frequently-used policy to select an item to evict on the grounds that it is more important to maintain connections and ensure freshness for frequently accessed and/or updated data.

Algorithm 3 Sink-first cache eviction

```
1: select set of items for which length(children) = 0
2: if set is non-empty then
3:     return random item from set
4: else
5:     return least-frequently-used item
6: end if
```

The *root-first* policy takes a different approach to the problem of disconnected graphs. Instead of trying to keep the graph from disconnecting, it instead tries to destroy the disconnected subgraph more quickly when a disconnection occurs. That is, when a peer discovers that it represents the root for a subgraph and is not the master, it assumes that it is disconnected and therefore should remove itself as it will become stale. This assumption is not always correct since children can become parents if they discover that updates are coming from an alternate route (Section 3.6.2), but it is always true if the subgraph is not connected.

Algorithm 4 Root-first cache eviction

```
1: select set of items for which parent = nil
2: if set is non-empty then
3:     return random item from set
4: else
5:     return item chosen using first-in-first-out policy
6: end if
```

3.6.2 Churn and Fault-Tolerance

Network churn and peer crashes also affect the connectivity of the child graphs for data. Each time a peer leaves the network, the graphs for data items it contained may be broken, preventing updates from reaching all copies in data caches.

We note that the multiple paths generated by the random walks provide some ability to withstand this problem. Specifically, when an update message is received from a peer that is not the parent of the peer in question, it indicates that a different route has become faster, suggesting that there is a break in the previous path. Therefore, the peer that receives such an update can set its parent to the new peer and update its distance. It then demotes the old parent to a child and forwards the message as normal (Algorithm 2). This provides some capability to route around peers that have left the network due to churn or faults.

For further protection, we reverse the SwapLinks procedure [42] on peer exit instead of dropping all neighbours. The peer that leaves takes pairs of neighbours and sends a message to each neighbour stating that it is leaving the graph and to replace its connection to the leaving peer with that of the other neighbour in the pair. A list of the data items for which the new neighbour remains a child of the existing parent is also sent to maintain that aspect of connectivity. The contacted peers then update the parent and child graphs for all data items they currently store information about. This procedure preserves some of the connections in child graphs when a peer leaves the system without crashing.

3.7 Multiversioning

There are scenarios in which the most recent version of a document is not always the desired version. The most obvious of these are revision control systems and the source code contained within them, as well as multi-versioned backup systems that allow users to recover files that have been overwritten. While source code generally requires all of the files be stored locally and backups are not appropriate to a system designed around temporary storage, both of them are based around the idea of reverting to a previous version. This idea is also expressed in systems more amenable to our model, such as Wikipedia, which allows users to view any version of a document, with the latest version being most requested.

Since multiple versions may be requested by users, it makes sense to be able to store and request older versions of the document as well. Such a feature is a simple extension, requiring only additional space be devoted to the caches to store multiple versions. Now,

Algorithm 5 Caching during Path Replication (lines 9-12)

if *no_data* **or** $cache.version < reply.version$ **then**
  $cache.data.add(reply.version, reply.data)$
  $cache.version \leftarrow reply.version$ {$cache.version$ is now a key to the most recent data}
end if

when a new version of a data item arrives at a peer as a result of a search or update message, it does not replace any older ones, but is instead added to a pool of versions. Algorithm 5 provides context for the one-line change required for replication after a search, with the same change made to line 18 of Algorithm 2 When a request arrives at the peer, the most recent version in the pool is returned.

Requests may now also specify a specific version that the user is interested in. When such a request arrives, the data cache now considers it to be found only when that exact version is present and returns that version rather than the most recent. If it is not, then the query continues and the information from the data cache is used as though it were contained in the path cache; that is, as a guide to a peer further up the path that may contain the data. While no specific policy for deciding which or how many versions should be kept is proposed, all versions are removed when an item is moved from the data cache to the path cache.

3.8 System Summary

The complete system runs on an unstructured network using multiple random walks to find data and path replication to cache copies. The caches retain information about the path which is used to propagate updates. When data is evicted from the data cache, the path information is moved to a separate path cache to preserve the routing information so that updates may continue to be routed to peers that retain the data in their cache. By taking advantage of the version numbers associated with the data and the path information, guarantees of freshness beyond a best effort policy are provided. Multiple random walks and transmission of local path information on peer exit ameliorate the effects of churn and peer failure.

Chapter 4

Performance Evaluation

To examine the effectiveness of the system, a series of experiments were performed to evaluate its ability to provide fresh data. While measuring freshness is the primary goal, some performance data was also collected to evaluate the costs of providing the freshness guarantees. In addition, the structure of the graphs formed by the paths is examined to provide context and an explanation for the results.

4.1 Experimental Setting

Experiments were performed on the PlanetSim simulator to evaluate the capability of the algorithms. These experiments used synthetic workloads and two trace-based workloads, one from Wikipedia [41] and one from Metafilter [31]. The synthetic workload modelled uniform and Zipfian distributions of queries and explored the effect of changing parameters such as the update frequency. The Wikipedia trace represents a read-mostly workload with a very large number of data items, while the Metafilter trace describes a flash-crowd-like workload where only a few data items are popular at one time. Despite the large scale of the Wikipedia trace, it behaves similarly to the synthetic workload.

4.1.1 PlanetSim

PlanetSim[19, 21] is a cycle-based simulator written in Java. It models large numbers of peers in a series of discrete time steps, with no implicit time associated with each step. During each step, all messages queued for sending in the previous step are delivered and

new messages are queued at each peer. Additional events, such as the creation of new peers or forcing the failure of a large number of peers, may be applied between each cycle. The design enforces compartmentalization, with separate transport, network, and application layers, each of which communicate through a well defined interface.

4.1.2 Configuration

The following baseline was used for all experiments. Variations from these values are described with the results for that experiment. The network graph contains 10 000 peers, each connected to 32 other peers. To represent the difference between master and non-master peers, 20% of the peers are designated as master peers and all data items are placed on them using a uniform distribution.

For all peers the data cache size is 25 items and the path cache size is 125 items. While the cache sizes are pessimistic, we note that machines do not gain capacity as the number of data items in the network grows, so any cache size will eventually represent a very small proportion of the total amount of data no matter how large it is. We therefore choose a small cache size relative to the number of data items to model the network when there is a significant amount of data relative to the capacity of each peer's ability cache. In addition, our experiments demonstrate that even a small path cache still has enough room to maintain path information for the most frequently accessed data on every peer. Increasing the size of the path cache retains more information relating to frequently accessed data, thereby having the obvious effect of improving the freshness results. The first-in-first-out cache eviction policy is used for both caches.

To issue queries, a random peer in the network is designated uniformly at random to issue the request. The data requested by the query is chosen according to the distribution associated with the workload. The peer issues 16 random walks (per the suggested range in [29]) to look for the item and reports success when the first of these walks returns with a result. No walk is terminated until after the first returns a result; especially, no TTL value has been set as a second cutoff method. While this would be impractical in a real system as overhead would gradually increase as searches for non-existent items accumulate, it permits an examination of the worst case behaviour of a search.

Updates are performed only at the master copy of the item. The master pushes the updated versions to children who have identified themselves through queries as described in the model.

4.1.3 Measurement

As the simulator is cycle-based, complete messages are sent in a single cycle unless they are deliberately delayed. Therefore, we measure time overhead to find a result in the number of times a message is forwarded before it reaches its destination (*hops*). While this does not approximate networks in which the cost to send messages to different peers varies significantly, it does place focus on a scale invariant feature of the network and so remains relevant whether the underlying network is a LAN or the Internet since it ignores the relative speeds of the machine and the network.

The number of queries that have not completed is monitored and formal measurement begins once the number stabilizes. At this point the number of queries completed at each cycle is approximately equal to the number of queries issued at each cycle. In addition, both the data and path caches are full and are distributed according to the query load.

The first random walk to return a result is used to measure freshness, even if another random walk associated with the search returned a more recent version.

All experiments were run five times and the results averaged over all runs. As there was little variation in the results, the 95% confidence interval for all freshness results is less than 0.2%.

4.1.4 Workloads

Synthetic

The number of data items is 10 000. With this number of data items, the standard data cache size described in Section 4.1.2 can store 0.25% of their total number. Data items for queries are selected using a Zipf distribution with an exponent equal to 1 and uniformly (to create a normal distribution). Within the Zipf distribution, the most popular item represents about one percent of the total number of queries. For brevity, we refer to the frequency of data access according to each distribution as *popularity*. The data to be updated is chosen uniformly at random.

Queries are issued at a rate of 100 queries per cycle (1% of the total number of peers per cycle) while updates are applied at a ratio equal to 20% of the number of queries issued. For many applications in which users share updates, such a ratio is pessimistic, as far more people consume updates than produce them. It is therefore likely that most real scenarios will, like those from our traces, exhibit better results. As the simulator's only limit on

network traffic during a cycle is the number of messages received, changing the query rate affects only the number of cycles required to gather data.

The observed variance in the fraction of fresh queries is less than one percent, and the median number of hops for all Zipf distributions was constant over all runs.

Wikipedia

The first source of real data for traces is Wikipedia. Traces from Wikipedia access logs between 18 Sep 2007 20:10:48 GMT and 21 Sep 2007 13:11:44 GMT were used as a source for the workload [41]. From the raw logs we extracted all reads and updates to articles from the English version. Searches, special pages like Random that are not part of the content, and images were discarded. Because the update rate is low (about 1 update per 1000 page requests), the workload was then modified to increase the effective update rate without biasing the data. To do this, all 55 488 updates but only the first million reads recorded in the sample were kept. The reads were then issued at a constant rate of 100 per time step. To preserve the features in the updates, the intervals between them in seconds were preserved and rescaled to the number of time steps required to issue all the read requests such that the first update was issued with the first read and the last update at the same time step as the last read request. This means that the number of reads per time step was not constant and that the mean is 5.5 updates per time step. The number of articles (data items) referenced by the million reads in the trace is 409 744. The most popular item in the trace represents 0.2% of the total number of queries.

The trace was run on 10 000 peers with data cache sizes of 1000 items (0.25% of the total number of data items, matching the proportion in the synthetic workload). The path cache was quintuple the size of the data cache and session guarantees (Section 3.5.2) were enabled.

Metafilter

We also ran experiments based on a workload trace from Metafilter [31]. Metafilter is a community blog, where any user may add a story with links and the other users discuss it. The available data spans the entire history of the site so we used data for 2012. User comments represent updates to the data, but a trace of the reads was not available.

Based on the read queries for cached data in CoralCache for Slashdot, a similar site [16], a similar flash crowd with a quick onset and a trail of requests after was modelled. To achieve this model, a time step in the simulator was used to represent the passing of

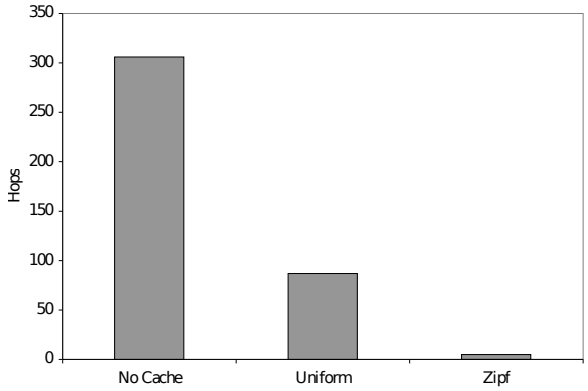

Figure 4.1: Number of hops required to find first replica

one second of real time. The clock was started at the time of the first update in 2012, and updates were applied in the second they occurred. As the updates already model the desired flash crowd and they provide reason for a user to make a second request, a random number of reads were issued after the update. The number of reads to issue for a single update was normally distributed with a mean of 75 and a standard deviation of 25. These reads were applied at random intervals until the number generated by that update completed. As a result, the mean number of reads per item was proportional to the number of updates, with a normal distribution and distributed in time in the same form as the updates.

4.2 Results

4.2.1 Query Response Time

In every run of the experiment, the median number of hops to find the first replica is 87 using a uniform distribution and 5 using a Zipf distribution for the popularity of the data items (Figure 4.1). This shows that caching is more effective when some data items are more popular than others. For comparison, running the same system without caching requires a median of 306 hops to find the only replica. In addition, there were far more incomplete queries without caching: the system ran out of memory due to an excess of incomplete searches. Therefore, it is possible that the true median is much higher as fully one third of the searches issued were not complete when the simulation terminated.

Interestingly, the number of hops may be used to estimate how popular the item is

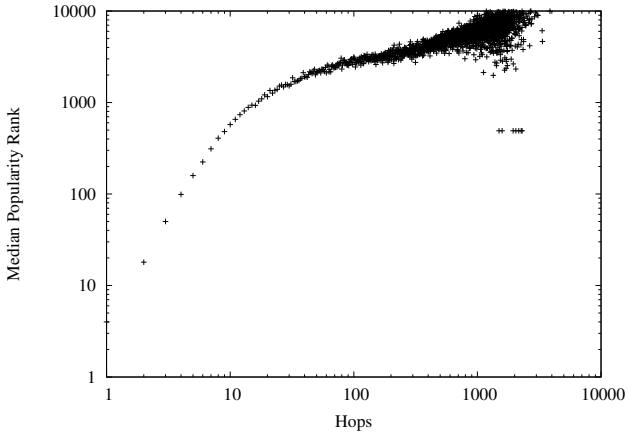

Figure 4.2: Median popularity for hops required to find data (Zipf). The most popular data item has rank 1.

(Figure 4.2). As the number of hops increases, the item found is more likely to be less popular.

4.2.2 Freshness

Both the Zipf distribution and the uniform distribution of data access frequency return similar results for freshness at a 20% update ratio. Under the Zipf distribution, 85.5% of queries returned fresh results while the uniform distribution returned fresh data for 88.3% of queries (Figure 4.3). Because a uniform distribution is less realistic and because there is no variation in popularity of data, we do not further discuss it[1]. Fresh data is returned more often as the update ratio decreases, with 97.7% of queries returning fresh data at a 5% update ratio for the Zipf distribution.

Query results that are more than one version older than the current fresh version are rare. Figure 4.4 shows that 97.8% of the results are no more than one version older than a fresh copy.

Figure 4.5 shows the fraction of fresh results as a function of the data items ranked by access frequency. Due to the Zipf distribution of the access frequency for data items, we use a logarithmic scale on the horizontal axis, causing results to be plotted according

[1]The behaviour of queries for the uniform distribution was identical to that of the least popular data items under the Zipf distribution in all cases.

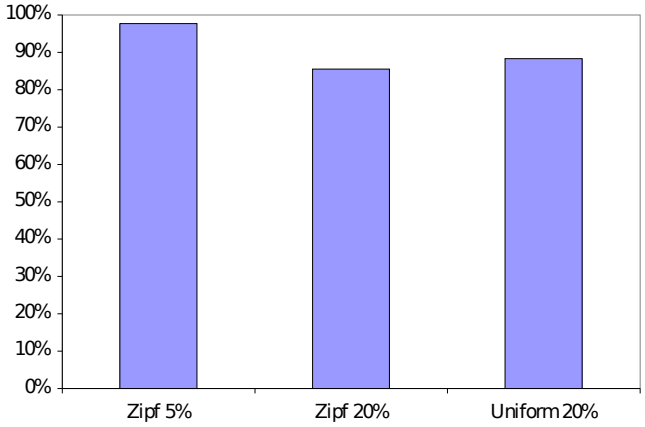

Figure 4.3: Number of queries returning fresh data

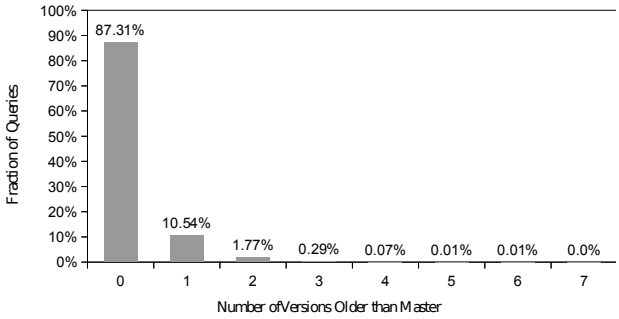

Figure 4.4: Age of Versions Returned; 0 Denotes Fresh Data

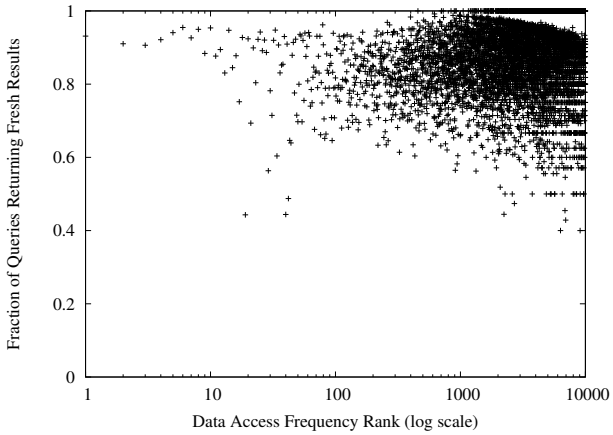

Figure 4.5: Fraction of fresh results at 20% updates (Zipf)

to the frequency at which they occur. Thus, the fraction of total queries for a particular data access frequency rank is equivalent to the horizontal space occupied. The fraction of fresh results depends on the data item access frequency, with three bands of effectiveness. The first band, which consists of roughly the 10 most popular data items and one quarter the total number of queries issued, has excellent results with 92.7% of queries returning fresh data. Stale results here are almost entirely a result of the time required to propagate updates. The second band consists of approximately the next 90 most popular items and about one quarter the total number of queries with 82.8% of these queries returning fresh data. The third band, containing the remaining items and one half the total queries, represents the least popular data items and shows some variation in the results. However, a very high proportion (89.9%) of queries in this band returned fresh data.

Freshness is sensitive to the time required to forward and apply updates. Importantly, increasing the delay at each peer, while reducing the absolute freshness values presented, does not change the fundamental shape of any of the freshness curves. Conversely, when we reduced the propagation delay to the minimum possible, freshness for the most popular data items was over 99%.

The presence of the path cache is essential to achieving these results. When it is removed, the median number of hops remains the same, however, the number of queries returning fresh fresh data drops to 74.4% with almost all of the loss occurring among the most popular data (Figure 4.6). Specifically, the most popular data is now fresh 56.4% of the time, the middle band is fresh 71.5% of the time and the least popular data provide fresh results for 87.6% of the associated queries (Figure 4.7).

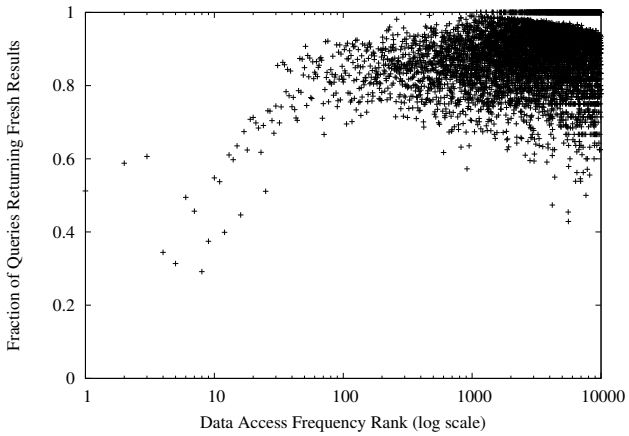

Figure 4.6: Fraction of fresh results without a path cache (Zipf)

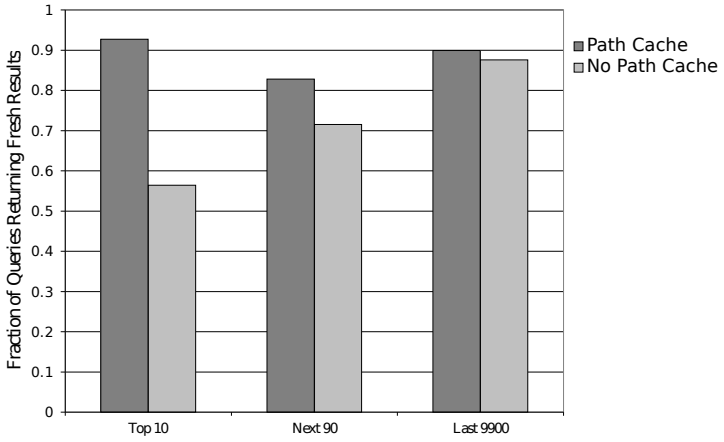

Figure 4.7: Effect of Path Cache on Freshness: The top 10 data items represent approximately 25% of all queries, the next 90 also represent about 25%, while the remaining 9900 represent the remainder.

4.2.3 Guaranteeing Freshness

We implemented the two freshness guarantees from Section 3.5. The first method terminates every search at the master copy, guaranteeing fresh results (Definition 1). As this strategy does not reduce the load on the master, it should be used only when freshness is essential to operation. To measure the increased cost to find the master, we continued every search after a cached copy was found until reaching the master but did not perform path replication on the continued portion of the search. This strategy allows us to measure the cost when a small fraction of the queries make this demand. The continued queries needed a median of 204 hops to find the master copy, improving upon a random walk without using a cache. In addition, one third of the queries took 10 hops or fewer to find the master using the path cache, whereas less than one percent of the queries took 10 or fewer hops without any cache. Further, the performance benefit offered by the caches increases as the cache becomes bigger. Quadrupling the size of the path cache to 5% of the number of data items decreases the median number of hops to find the master to 58. The enlargement of the path cache did not significantly affect the fraction of queries returning fresh data nor the median time to find a cached copy for a query run without the guarantee.

The second method provided a session guarantee (Definition 2) that a copy older than one already seen would not be returned to the peer issuing the query. For this experiment, we delimited a session by the amount of time information about the session remains in the path cache. We did not keep any data in the path cache for a period of time longer than when it would normally be evicted by the cache eviction policy. Therefore, popular items were requested several times in a single session, while the least popular data items were requested only once within a session. Since the least popular data have few to no cached copies, they are least affected by this policy as there is a high probability that any request would be satisfied by the most recent version at the master.

In addition to providing the stated guarantee, we observe the following effects. First, in addition to the guarantee that the copy is at least as up-to-date as the last accessed version, there is also an increase in the probability that the data returned is fresh. Where the total number of fresh results is 85.5% without the guarantee, it is 91.6% with the guarantee. Second, the disproportionate reduction in the proportion of fresh results seen from Figure 4.5 for moderately popular items is reduced in Figure 4.8. The median number of hops before the query completes is 4, which implies that there is no additional time cost to add session guarantees to the system in all cases.

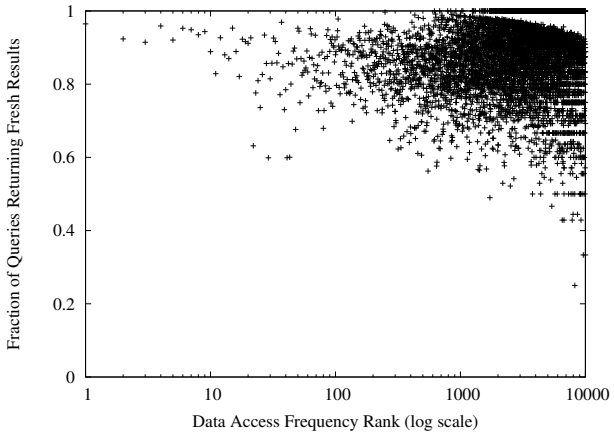

Figure 4.8: Fraction of fresh results with session guarantees

4.2.4 Wikipedia

As unmodified data are uninteresting, only the 11 617 pages modified by the 55 488 updates were considered when evaluating freshness. The number of fresh results returned is extremely high, matching the 5% update scenario with 97.77% of requests returning fresh data. The low popularity of even the most popular item had a significant effect on the time to find results. The median number of hops was 242.

As Figure 4.9 shows, the same three bands for freshness based on popularity remain visible. The well defined lines in the least popular band are due to low numbers of reads; the mean number of reads per data item is 2.5. From right to left, the lines represent one, two, three and four stale reads for a data item.

4.2.5 Metafilter

91.7% of the queries returned fresh results. As the Metafilter trace simulates a flash crowd, caching was very effective, with the median number of hops equal to zero, indicating that over 50% of queries are satisfied by a locally cached copy. This shows that the caches adapt quickly to changes in the popularity of data. As nearly all items become the most popular item for a short period of time, we do not include a graph showing results by popularity.

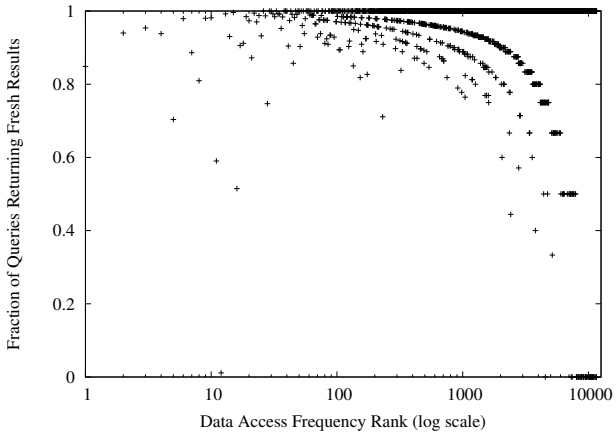

Figure 4.9: Fraction of fresh results for modified items in Wikipedia trace (10 000 peers, 1000 item data cache)

4.2.6 Churn

To examine the effects of churn on the system, we ran additional synthetic experiments in which we introduced churn at the beginning of the experiment and waited for the system to stabilize as with other experiments. We introduce churn only to non-master peers in accordance with our model of expected user behaviour. The non-master peers experience churn by distributing their lifespan according to a Weibull distribution with $k = 0.4$ based on the observations of real P2P networks by Stutzbach and Rejaie [36]. New peers are introduced to replace leaving peers, and these peers do not leave for a minimum of 150 cycles to both give them time to fully join the system and to ensure that they have time for one or more queries to complete before leaving. We set λ so that an average of 7 peers leave the system each cycle, for a total of 7% of the peers leaving the system every hundred cycles.

Our experiments show that the system is not only resistant to churn but improves freshness when churn occurs. The fraction of queries that return fresh data is 89.2%. We note that the second, less popular band of data observed in Section 4.2.2 experiences a noticeable improvement from churn while the third band is unaffected.

When we examine the graphs of the paths formed in each of these bands, we identify the following causes for their behaviour. For the items with greatest popularity, the breaks sometimes cause delays in propagation, causing queries to reach stale results slightly more often. For the items of moderate popularity, the breaks in the paths are routed around and

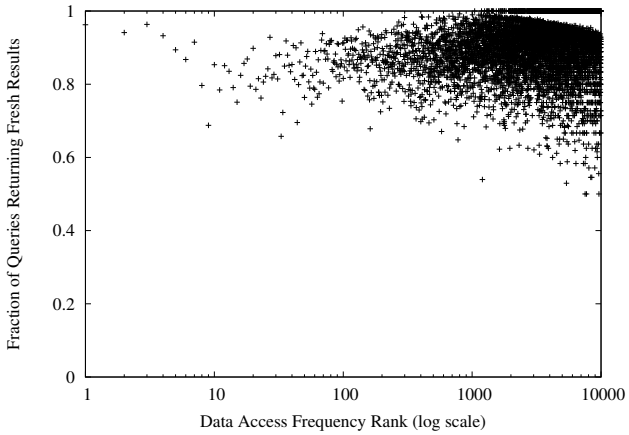

Figure 4.10: Fraction of fresh results with churn

new queries tend to reduce the maximum path length, producing a decrease in the amount of time required to propagate an update and a corresponding increase in the number of fresh results. For the least popular items, the lack of cached copies causes there to be no paths to break, so churn does not affect the results.

As we noted that adding session guarantees to the system increased freshness, especially in the area most affected by churn, we repeated the churn experiment with the session guarantee added (Figure 4.11). With that guarantee added, 93.3% of searches retrieved fresh results. Thus, in addition to the benefit of having freshness on a per session basis, session guarantees mitigate the effect of churn on the system.

4.2.7 Cache Eviction Policy

Experiments with different cache eviction policies were performed with larger caches. The data cache was set to 50 items and the path cache was enlarged to 250 to magnify the differences between cache algorithms.

The *sink-first* policy, which preferentially removes sinks from the graph in the path cache as proposed in Section 3.6.1, is clearly sub-optimal. It performs more poorly than any other method, even though its fallback algorithm, LFU, had the best performance for the same cache. The scheme successfully kept graphs locally connected, however, when a break did occur, due to churn or a lack of sinks in a particular peer's cache, the newly disconnected graph remained disconnected, creating islands of stale data.

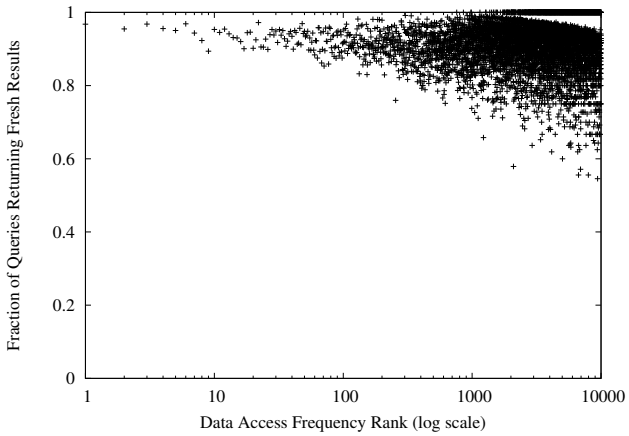

Figure 4.11: Fraction of fresh results with churn and session guarantees

Table 4.1: Freshness for Cache Eviction Algorithms

		Data Cache		
		Random	Root First	FIFO
Path Cache	LFU	82.54%	90.00%	90.06%
	Random	76.07%	82.12%	84.63%
	Leaf First	70.45%	78.27%	82.69%
	FIFO	80.58%	87.24%	88.88%
	LRU	80.33%	87.52%	89.43%

While the random policy allows for square root replication, it appears to be an inferior policy for maintaining freshness of that same data. FIFO offers the same quality of replication as a random policy while greatly increasing freshness. While an LFU policy offers further increase, its benefits appear modest and are offset by an increase in read costs for less popular data. The difference in freshness between the LRU and FIFO policies is insignificant, though this may be an indication that the cache sizes remain too small to make a distinction between the policies even after the increase in size for this experiment and so not indicate that they are equivalent in performance.

Preferentially removing the root when it has no connection slightly decreases freshness when applied to the data cache over plain FIFO. This means that removing data that was added earlier is more valuable than removing data that cannot be updated. It seems likely that such a policy may perform better for data that is very frequently updated, especially if other data in the system is updated less frequently.

4.2.8 Multiversioning

To test multiversioning, 2% of queries were reissued after they were answered to search for the version immediately prior to that contained in the reply. This process was applied to both the synthetic and Wikipedia data sets. These additional searches had little effect on the freshness of other queries as 85.1% of those queries returned fresh results for the synthetic data and 99.0% for the Wikipedia data. The median number of hops to find data one version old is the same as that needed to find the initial result for the synthetic workload and better, with 22 hops, for the Wikipedia data. These results imply that the peer that initially satisfied the request is usually also able to satisfy the request for older data.

4.2.9 Model Verification

In addition to evaluating freshness, we performed checks to verify that the implementation of path replication and random walks is correct. Using snapshots of the system, we measured the number of replicas for selected popularity levels to measure the levels of replication across popularity. The replication levels were approximately proportional to the square-root of query frequency, as expected.

As a second check, we measured the number of hops required to find data with respect to the data's popularity. Figure 4.12 shows the frequency of the total number of hops required to both find an existing replica of the data and to place new replicas along the

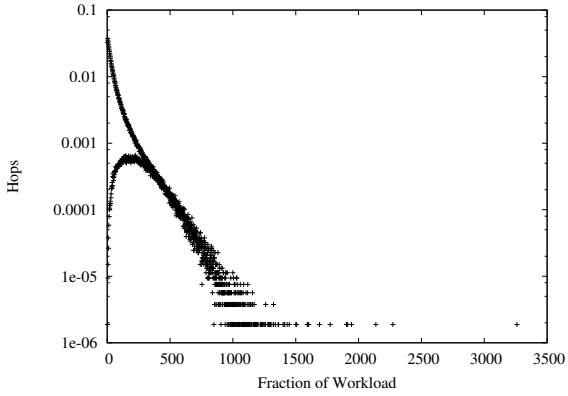

Figure 4.12: Number of hops required to complete a query (Uniform)

path to the peer that issued the query for the case of a uniform distribution of queries. While the exponential distribution is clearly visible, there is a second, lower, set of data points underneath the expected set. The upper set contains frequencies when the number of hops is even, the lower set when the number of hops is odd. This split between odd and even hops occurs because of cycle elimination on the return path; if there are no cycles then the number of links traversed looking for the item is equal to the number of links traversed placing replicas on the return path. If, however, a cycle or cycles whose total length is odd forms during the search, the total number of hops is reduced by an odd value as the cycles are skipped when results are returned, resulting in odd counts. As cycles at short distances are rare, the number of times small odd hop counts occur is correspondingly low. Without maintaining a list of previously visited peers in the message, it is impossible to avoid all possible cycles. Thus the distribution of the number of hops has the same form as that observed by Lv et al.[29], though the initial values are somewhat lower because of the smaller cache size and different number of data items in the system.

These results confirm that our search and replication algorithms are correct.

4.3 Graph Structure

To quantify path information, we periodically record a snapshot of the caches on each peer for designated data items. From this cache information, we collect information about the paths within the system for individual data items, including quantity and path lengths, as well as a count of all the versions of data currently present in the system. The snapshots

fell into three categories corresponding to the three observed bands in the distributions of freshness.

In the first category, path lengths are short and the graph is usually connected. The medium popularity items exhibit great variation in path length and graph connectivity, sometimes having effectively no paths and sometimes exhibiting reasonable connectedness. As the popularity of data items in this category decreases, the frequency of disconnected graphs increases though the number of cached copies does not decrease significantly. The lowest popularity items usually have no copies other than the master.

Over time, the paths for the most popular queries tend to decrease in length while maintaining a constant number of copies. This suggests that they are approaching a form resembling a minimum spanning tree of the covered peers, though with additional redundant connections. Such redundant connections increase the total number of messages sent to apply an update, increasing the system load. However, they also provide fault tolerance as they may be used to route around problems as seen in the churn results, decreasing the number of stale copies. Paths do not become shorter for data associated with less popular queries.

Combining these observations suggests the following interpretation of the graph structure for the path networks in each of the popularity categories. Those in the most popular section form large connected graphs with minimal distance from the master to all sinks. Updates complete quickly and reach every replica reliably. They are also costly as they flood the network, but since the majority of peers have a copy, sending messages to the majority of peers is required. Those in the second band still reliably place replicas, but a large proportion of the paths to the master break, causing the graph of path information to form multiple disconnected pieces. Updates complete slowly as there can be long paths without an efficient route to the master and the disconnections prevent some replicas from receiving updates. The disconnected nature of the graph also prevents update messages from flooding the network, as only those peers that remain connected to the master have updates pushed to them. Those in the third band have so few replicas that searches are more likely to find the master as they are to find any copy, thus preserving a high level of freshness. Update messages are efficient, as they rarely follow more than a single previous random walk.

The average number of children remembered by peers containing path information slowly increases with time. This is because long lived replicas, most notably the master, never remove peers from the list of children. Specifically, after a peer adds a new data item to the cache, it acquires new children only when it is the target of a query. When this happens regularly, it will tend to return to the data cache from the path cache, preserving

the knowledge of interested peers. If it does not receive requests, it will eventually be evicted from the path cache and all information is lost. In other words, the peers with the fewest number of children are also the ones most likely to be in the path cache and the ones most likely to be evicted. This result is not specific to any popularity level.

4.3.1 Disconnected Graphs

The second band of popularity observed in Section 4.2.2 for data items illustrates the problems that occur when graphs become disconnected. The session guarantee's success in reducing these problems shows that it is possible to use local knowledge to deal with some cases. In addition, experiments before implementing the reverse link swapping procedure on peer exit showed a far greater reduction in the number of fresh results under churn.

Identifying when a graph has become disconnected and actively reconnecting, will break the square-root replication shown to provide ideal read performance [10] if it causes new replicas to be placed proportionately to the frequency of disconnection, which occurs most frequently among less popular items.

Conversely, becoming disconnected may improve the speed of updates, improving freshness, when a new search finds a shorter path than the previous search via a nearby peer that found a shorter path. Further, since new paths are only tried in association with requests, the system does not spend resources on improving results that are unlikely to be requested. Therefore, establishing ideal graph dynamics using only local knowledge is an interesting problem that exhibits opportunities for improvement.

Chapter 5

Conclusion and Future Work

We addressed the challenging problem of managing updates in an unstructured peer-to-peer system using a lazy update protocol that propagates each update along the paths originally used to return data to the peers that issued queries. The resulting technique effectively controls update propagation, providing fresh data for most queries. Its assurance that updates cannot be applied out of order and consistency in providing data that is close in version to the current make it suitable for systems with high absolute update rates.

While the system supports arbitrary queries, we have not measured their cost. We note that the path cache becomes less effective as a search tool when multiple data items match the query and if multiple items return along the same walk then the pressure on the associated caches is increased. Dealing with these issues is left to future work.

We also demonstrated methods to trade performance for freshness. We provided two freshness guarantees, following the cached path information to the master and session guarantees. Searching for the master guarantees freshness and can be done faster than a random walk that does not take advantage of the path information. Session guarantees ensure that versions of data older than the last accessed version are not seen. This session guarantee also improves, at a small cost, the proportion of queries returning fresh data. We therefore recommend providing the session guarantee in all cases.

We demonstrated that the cache eviction policies used for the data and path caches have a significant effect on the freshness of data within the system by directly affecting the form of the graph. As such, further research in cache eviction strategies appears likely to prove fruitful.

The use of random walks and path swapping on peer exit are effective at dealing with unanticipated breaks in the path graph. Churn has little effect on the results for the most

and least frequently accessed data items and has a beneficial effect on the system overall. This result demonstrates a valuable characteristic for unstructured networks, whose simplicity in joining and leaving encourage algorithms that do well in unstable environments.

Overall, we have shown that our system provides freshness under a variety of system loads, with the large number of data items in the Wikipedia trace demonstrating scalability, and the Metafilter trace demonstrating adaptability to changing conditions.

References

[1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. *Cooperative Information Systems*, pages 179–194, 2001.

[2] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *SIGMOD Conference*, pages 97–108, 1999.

[3] Sonja Buchegger and Anwitaman Datta. A case for P2P infrastructure for social networks - opportunities and challenges. In *Proceedings of WONS 2009, The Sixth International Conference on Wireless On-demand Network Systems and Services*, Snowbird, Utah, USA, February 2-4, 2009.

[4] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM, 2003.

[5] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. Scan: A dynamic, scalable, and efficient content distribution network. In *In Proceedings of the International Conference on Pervasive Computing*, 2002.

[6] I. Clarke, O. Sandberg, M. Toseland, and V. Verendel. Private communication through a network of trusted connections: The dark freenet.

[7] Ian Clarke, Oskar Sandberg, Matthew Toseland, and Vilhelm Verendel. Private communication through a network of trusted connections: The dark freenet. *Network*, 2010.

[8] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *International*

workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[9] Clip2. The gnutella protocol specification v0.4 (document revision 1.2). http://www.clip2.com/GnutellaProtocol04.pdf, 2001.

[10] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 177–190, New York, NY, USA, 2002. ACM.

[11] Data, data everywhere. *The Economist, Special Report: Managing Information*, February 2010.

[12] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 76–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] D. Del Vecchio and S.H. Son. Flexible update management in peer-to-peer database systems. In *Database Engineering and Application Symposium, 2005. IDEAS 2005. 9th International*, pages 435 – 444, july 2005.

[14] S. Ertel. Unstructured p2p networks by example: Gnutella 0.4, gnutella 0.6.

[15] Nathan S. Evans, Chris Gauthierdickey, and Christian Grothoff. Routing in the dark: Pitch black. In *In Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC). IEEE Computer Society*, 2007.

[16] Michael J. Freedman. Experiences with coralcdn: a five-year operational view. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[17] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[18] Freenet Project. Updateable subspace key, 2010.

[19] Pedro García, Carles Pairot, Rubén Mondéjar, Jordi Pujol, Helio Tejedor, and Robert Rallo. Planetsim: A new overlay network simulation framework. In Thomas Gschwind and Cecilia Mascolo, editors, *Software Engineering and Middleware*, volume 3437 of *Lecture Notes in Computer Science*, pages 123–136. Springer Berlin / Heidelberg, 2005.

[20] Sandra Garcia Esparza. Jxta-sim: A simulator for evaluating the jxta lookup algorithm. Master's thesis, University of Dublin, 2009.

[21] Pedro García Lopèz, Carles Pairot Gavaldà, Rubén Mondéjar Andreu, Jordi Pujol Ahulló, Marc Sanchez Artigas, and Helio Tejedor Navarro. Planetsim project, 2009. http://projects-deim.urv.cat/trac/planetsim/.

[22] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[23] L. Han, M. Punceva, B. Nath, S.M. Muthukrishnan, and L. Iftode. Socialcdn: Caching techniques for distributed social networks. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing, September 2012*, 2012.

[24] Kevin Henry, Colleen Swanson, Qi Xie, and Khuzaima Daudjee. Efficient hierarchical quorums in unstructured peer-to-peer networks. In *OTM Conferences (1)*, pages 183–200, 2009.

[25] Kevin Henry, Colleen Swanson, Qi Xie, and Khuzaima Daudjee. Efficient hierarchical quorums in unstructured peer-to-peer networks. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, pages 183–200, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35:190–201, November 2000.

[27] Christof Leng, Wesley W. Terpstra, Bettina Kemme, Wilhelm Stannat, and Alejandro P. Buchmann. Maintaining replicas in unstructured p2p systems. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 19:1–19:12, New York, NY, USA, 2008. ACM.

[28] Jian Liang, Rakesh Kumar, and Keith W. Ross. The fasttrack overlay: a measurement study. *Comput. Netw.*, 50(6):842–858, April 2006.

[29] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[30] E.P. Markatos. Tracing a large-scale peer to peer system: an hour in the life of gnutella. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 65–65. IEEE, 2002.

[31] Metafilter. Metafilter infodump, 2012.

[32] Mujtaba Khambatti Mujtaba. Push-pull gossiping for information sharing in peer-to-peer communities. In *In Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pages 1393 1399, Las Vegas*, pages 1393–1399. CSREA Press, 2003.

[33] Oversi. Overcache p2p caching and delivery platform, 2010.

[34] K.P.N. Puttaswamy and B.Y. Zhao. A case for unstructured distributed hash tables. In *IEEE Global Internet Symposium, 2007*, pages 7–12. IEEE, 2007.

[35] Rami Rashkovits and Avigdor Gal. A cooperative model for wide area content delivery applications. In *OTM Conferences (1)'05*, pages 402–419, 2005.

[36] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[37] Sai Susarla and John Carter. Flexible consistency for wide area peer replication. *Distributed Computing Systems, International Conference on*, 0:199–208, 2005.

[38] Xueyan Tang, Huicheng Chi, and S.T. Chanson. Optimal replica placement under ttl-based consistency. *Parallel and Distributed Systems, IEEE Transactions on*, 18(3):351 –363, march 2007.

[39] Xueyan Tang, Jianliang Xu, and Wang-Chien Lee. Analysis of ttl-based consistency in unstructured peer-to-peer networks. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1683 –1694, dec. 2008.

[40] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 49–60, New York, NY, USA, 2007. ACM.

[41] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[42] V. Vishnumurthy and P. Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1 –12, april 2006.

[43] M. Waldrop. Wikiomics. *Nature, Special Issue on Big Data*, 455:22–25, September 2009.

[44] C. Wang, L. Xiao, Y. Liu, and P. Zheng. Distributed caching and adaptive search in multilayer p2p networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 219–226. IEEE, 2004.

[45] H. Yamamoto, D. Maruta, and Y. Oie. Replication methods for load balancing on distributed storages in p2p networks. In *Applications and the Internet, 2005. Proceedings. The 2005 Symposium on*, pages 264 – 271, jan.-4 feb. 2005.