PlanetP: Infrastructure Support for P2P Information Sharing

Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, Thu D. Nguyen
{mcuenca, peery, rmartin, tdnguyen}@cs.rutgers.edu

Technical Report DCS-TR-465
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

November 19, 2001

Abstract

Storage technology trends are providing massive storage in extremely small packages while declining computing costs are resulting in a rising number of devices per person. The confluence of these trends are presenting a new, critical challenge to storage and file system designers: how to enable users to effectively manage, use, and share huge amounts of data stored across a multitude of devices. In this paper, we present a novel middleware storage system, PlanetP, which is designed from first principles as a peer-to-peer (P2P), semantically indexed storage layer. PlanetP makes two novel design choices to meet the above challenge. First, PlanetP concentrates on content-based querying for information retrieval and assumes that the unit of storage is a snippet of XML, allowing it to index arbitrary data for search and retrieval, regardless of the applications used to create and manipulate the data. Second, PlanetP adopts a P2P approach, avoiding centralization of storage and indexing. This makes PlanetP particularly suitable for information sharing among ad hoc groups of users, each of which may have to manage data distributed across multiple devices. PlanetP is targeted for groups of up to 1000 users; results from studying communities of 100-200 peers running on a cluster of PCs indicates that PlanetP should scale well to the 1000-member threshold. Finally, we describe BreezeFS, a semantic file system that we have implemented to validate PlanetP's utility.

1 Introduction

Storage technology trends are providing massive storage in extremely small packages while declining computing costs are resulting in a rising number of devices per person. The confluence of these trends are presenting a new, critical challenge to storage and file system designers: how to enable users to effectively manage, use, and share huge amounts of data stored across a multitude of devices. The current model of hierarchical directory-based storage on a single machine is increasingly becoming inadequate to meet this challenge. Even today, a typical user is faced with the complex problem of managing gigabytes, and soon, terabytes, across many systems. For example, in the near future, a typical user will not only possess multiple PC's and laptops at work and home, but also will own a plethora of devices such as PDA's, cellphones, and digital cameras. Each of these is a complete computer capable of running a modern operating system and able to store gigabytes of information.

In this paper, we present a novel middleware storage system, PlanetP, which is designed from first principles as a peer-to-peer (P2P), semantically indexed storage layer. PlanetP makes two novel design choices as a storage layer. First, instead of blocks, PlanetP's unit of storage are snippets of eXensible Markup Language (XML) [1]. This choice makes it possible for PlanetP to index and support searches over the data without considering the applications used to create and manipulate the data.

The size of storage that will soon be available per person makes content-based search and retrieval a vital property new storage and file systems must support. The success of Internet search engines is strong evidence that content-based search and retrieval is an intuitive paradigm that users can leverage to manage and access large volumes of information. With just a few search terms, high quality, relevant documents are routinely returned to users out of billions of choices. PlanetP aims to provide a *personal search service*, built into the storage system, thereby enabling ad-hoc groups of users to store, share and manage data distributed across multiple devices.

[1] By snippet, we are really referring to an XML document. However, we use snippets to indicate that many XML documents may be very small.

The second novel aspect of PlanetP is that it is designed from the ground up as a P2P system. This means that PlanetP is designed to operate in a fluid environment, where peers may join and leave the system dynamically. Also, PlanetP explicitly recognizes that in such an environment, users may have many copies of the same data spread across multiple devices. For example, user now hoard information locally in devices such as laptops, PDA's, and cell phones in order to maintain access during periods of weak or non-existent connectivity. However, hoarding introduces new problems of consistency, and worse, of even finding the documents among the maze of directories. PlanetP provides explicit support for managing and synchronizing replicas to address these problems.

Two alternative design approaches are to rely on either centralized storage or centralized indexing and search. While centralization of storage is tempting from the system designer's perspective, in practice users will use storage wherever it is available; for example, it is inevitable that a laptop, cell phone, or PDA will be used to store data that is not in the central repository. Centralized indexing and search has proven to be a valuable approach in web search engines. However, this approach has many inefficiencies, including the replication of data in a central place as well as recrawl of data that may not have changed. Such inefficiencies are tolerable in commercial search engines because they are backed by companies with large computing, connectivity, and administrator budgets. These costs, especially administrators, cannot be ignored by ad hoc groups of users with limited resources.

One of the key elements of the success of the client-server model can be traced back to it's ability to allow small to medium sized groups to share information. PlanetP is designed to scale to this level as well, allowing a group of trusting users to manage a single storage volume across multiple machines, laptops, and devices. We target a regime of about 1000 devices, possibly spread across the Internet.

Targeting community sizes of around 1000 nodes instead of very large systems (such as those targeted by efforts like OceanStore[16], Chord[24], and PAST[8]) allows us to assume that each member can track the membership of the entire group. PlanetP takes advantage of this assumption to diffuse individuals' summaries of their content, in the form of Bloom filters [1], to all members of the group. This approach has two advantages. First, if a peer is off-line, other peers can still at least know whether it contains information that they are searching for. Second, searches place very little load on the community at large for the bulk of data that changes slowly; the searching peer query against his local store of Bloom filters and then contact peers with matching snippets directly.

The disadvantage of using a diffusion approach is that it may take some time to spread new information. Also, it requires that information be spread to everyone in the community. To address these concerns, peers in PlanetP also collaborate to implement an information brokerage service using consistent hashing [15]. This approach is similar in spirit to that taken in [24].

Finally, in order to export a well-known API, we are in the process of implementing a file system, Breeze, that allows users to build a hierarchical and semantic directory structure on top of PlanetP. In this paper, we describe the current status of the Breeze file system (BreezeFS) and how it leverages PlanetP to provide a content-addressable way to build directories; in BreezeFS, a directory not only represents a collection of files, it also represents a collection of files logically related by content. Sub-directories represent more refined queries over the content.

The remainder of the paper is organized as follows. Section 2 describes the design and implementation of PlanetP. Section 3 provides a preliminary evaluation of PlanetP's performance. Section 4 describes BreezeFS while Section 5 briefly examines BreezeFS performance. Section 6 discusses related work. Finally, Section 7 concludes the paper and discusses our planned future work.

2 PlanetP

As explained in Section 1, PlanetP is a peer-to-peer middleware storage layer. Like most distributed storage systems, queries for data are made transparent to their actual location. PlanetP is peer-to-peer in the sense that each node participates in the storage, search and retrieval functions.

PlanetP provides two distinct features for a storage layer. First, queries are made based on the content of the data, not a logical block number. Second, queries can be persistent. That is, a query can persist in the community long after it was initiated; a notify event is sent to the requesting peer when new data matching the query enters the system. The combination of these two features places PlanetP closer to publish-subscribe systems than a traditional storage layer. We describe how PlanetP differs from these systems in Section 6.

Given PlanetP's semantic, peer-to-peer model, the key questions are (1) what is the unit of storage, (2) how are the indexes built, (3) how are membership lists maintained , and (4) how queries are performed.

PlanetP's unit of storage is an XML snippet. We made this choice for two reasons: (a) the assumption of XML allows PlanetP to index and support searches over arbitrary data without considering the applications used to create and manipulate the data, and (b) in the future, we plan to use XML tags to provide additional semantics to PlanetP

queries. When a peer wishes to write a snippet to PlanetP, it uses an explicit *publish* operation. When publishing a snippet, the publisher can also provide a set of keys that should map to the snippet; this allows the publisher to associate keys with the snippet that may not appear in the snippet itself. We found this feature useful, for example BreezeFS makes extensive use of keywords not appearing in the document text. Currently, the only keys that PlanetP handles are text strings. In the future, we intend to extend PlanetP to allow typing of the keys using XML tags and plan to build automatic keyword extraction.

PlanetP uses two mechanisms to index the communal data store. First, for each peer, PlanetP summarize all the keys associated with snippets that the peer has published using a Bloom filter [1] and diffuses it throughout the community. Each peer can then query for content across the community by querying against the Bloom filters that it has collected. Diffusion is also used to maintain membership information across the peers; that is, each peer maintains a local list of all active members. In the absence of joins and departures, all members should eventually have the same local directory and set of Bloom filters.

Our global diffusion approach has the advantage of placing very little load on the community for searches against the bulk of slowly changing data [20, 7]. The disadvantage of using a diffusion approach is that new or rapidly changing information spreads slowly, as diffusion is necessarily spread out over time to minimize spikes in communication bandwidth. To address this problem, peers in PlanetP also implement an information brokerage service which uses consistent hashing [15] to publish and locate information. This second indexing service supports the timely location of new information, as well as the exchange of information between subsets of peers without involving the entire community.

When a query is posed to PlanetP, it performs two searches. First it uses its list of Bloom filters to compute the subset of peers that may have snippets that match the query, forwarding the query to this subset. Second, it contacts the appropriate information broker for each key in the query. Once all contacted peers and brokers have replied, PlanetP processes the replies and returns the matching XML snippets to the caller.

Persistent queries are kept by both the querying peer and the queried brokers. When new information is published to a broker that matches a persistent query, the broker forwards the information to the peer that posted the query. The querying peer also keeps the persistent query to check against new Bloom filters as they arrive via diffusion.

Figure 1 shows PlanetP's current architecture. XMLStore is an in-memory hash table that is used to store the published

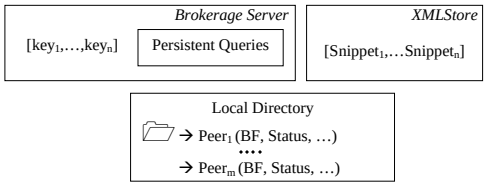

Figure 1. *PlanetP's current architecture.*

XML snippets. The Local Directory is a list of all members, their Bloom filters, and other per-member information (such as whether a member is currently on-line). The Brokerage Server implements the information brokerage service. When a PlanetP peer shuts down, PlanetP uses the local file system to store the content of the XMLStore. This is why we call PlanetP a "middleware" storage system.

In the remainder of this section, we discuss the diffusion algorithm used to spread information, how the information brokerage service works, and how queries are handled by PlanetP. At the end, we give the current programming interface exported by PlanetP.

2.1 Information Diffusion

PlanetP uses a combination of Harchol-Balter et al.'s name dropper algorithm [14] and Demers et al.'s rumor mongering algorithm [6] to maintain a consistent Local Directory at each member in the community. This algorithm works as follows. Each member x maintains a *gossiping* interval, call T_g. Every T_g, x randomly chooses a target y from its Local Directory and sends a summary of its local directory to y; we call this a gossip message. When y receives the gossip message, it determines whether x knows anything that it does not. If so, y contacts x to update its Local Directory. The default gossiping interval is currently set to 1 second to prevent an instantaneous communication spike whenever a new piece of information enters the system.

To reduce gossiping when the system has reached a stable configuration, the gossip interval is adjusted dynamically. We use a simple algorithm from [6] which works quite well in practice: x maintains a count of the number of times it has contacted a peer that does not need any information from x's Local Directory. Whenever this count reaches 2, x increases its gossiping interval by 1 second. Whenever x receives a gossip message which contains more updated information than it has, then it resets its gossiping interval to the default. The constants we use were found experimentally to work well in our current test-bed, which is comprised of PCs connected by a 100 Mb/s Ethernet LAN. The values would likely need to be modified for WAN-connected communities.

Using a dynamic gossiping interval has two advantages. First, we do not need to define a termination condition. Given that the algorithm is probabilistic, there is always a small chance that any termination condition will not result in all peers having a consistent view of the system anyway. Second, when global consistency has been achieved, the bandwidth use is negligible after a short time.

Finally, a peer skews its random selection of a gossiping target more heavily toward peers that it has not contacted in a while. This increases the chances that the members views are different and so makes the gossiping useful.

When a member (re)joins the community—a join can happen when a brand new member joins or when a member that has been inactive comes back on-line—it simply starts gossiping to let others know that it is back on-line. To join, a new member must know how to contact at least one peer of the community that is currently on-line.

2.2 Information Brokerage

In addition to the use of Bloom filters to summarize information being shared with the community by each member, PlanetP also supports an information brokerage service for more flexible information sharing. Each member joining a PlanetP community can choose to support the brokerage service or not; this allows devices without sufficient computing, storage, or communication resources to avoid hosting this service.

This service works as follows. Information is published to the brokerage service as an XML snippet with a set of associated keys. The network of brokers use consistent hashing [4] to partition the key space among them. To implement this algorithm, each active member chooses a unique broker ID from a predetermined range (0 to $maxID$). Then, all members arrange themselves on a ring using their IDs. To map a key to a broker, we compute the hash H of the key. Then, we send the snippet and key to the broker whose ID makes it the least successor to $H \ mod \ maxID$ on the ring. For example, if H is 4, and a broker with ID 4 exists, then the key and snippet are sent to broker 4. On the other hand, if the broker with the smallest ID greater than 4 is 6, then we send the key and snippet to broker 6.

The complexity of implementing this service lies in handling the dynamic joining and leaving of members. A new member wishing to join a PlanetP community must first contact some active member of that community to obtain a copy of that member's Local Directory. It then randomly chooses an ID that does not conflict with the IDs of any known active member. As the copy of the Local Directory that it obtained is not guaranteed to be globally consistent, however, it still must check for conflicts. To do this, it contacts the peer that should be its immediate successor (as indicated by the Local Directory) and informs the successor of its presence and ID. If the successor has a predecessor with ID less than that of the new member, then the join is done. Otherwise, the new member has to contact the new successor and tries again.

Even after following this process, it is possible for members to join with the same broker ID. This can happen when many members join at once and there is wide inconsistencies among the Local Directories. When this occurs, the members simply compare their joining time (when duplication of identity is discovered through information diffusion). The member with the lowest joining time keeps the ID; the others must rerun the join algorithm to get new IDs. Any snippets and keys that were published at the loosing members are transferred to the winning member.

When leaving the community, the leaving peer should contact its successor in the ring and forward all data that have been published to it. If a member leaves without doing this cleanup, then the published data is lost. It is possible to avoid the loss of data by using several different hashing functions and publishing replicas to several different brokers. Currently, we do not support this replication; rather, we depend on the fact that data is summarized in the Bloom filters as well as published to the brokerage. If information is lost because a peer leaves without forwarding information, eventually, it can be found again when the Bloom filter is diffuse throughout the community.

In the best case, a join and departure only needs one message to contact the successor (a join also needs the original message to initialize its Local Directory). Later, this change to the ring structure will be piggybacked on our diffusion algorithm and thus spread to all active peers. On the other hand, if membership is in a state of extremely high flux, joining and leaving might require $O(n)$ messages, where n is the number of active members.

2.3 Searching

PlanetP currently supports a basic query language where a query string is interpreted as a conjunction of keys separated by white spaces (Keys comprised of several words can be specified using double quotes).

When presented with a query, PlanetP first searches the Bloom filters in its Local Directory to obtain a list of candidate peers that might have data matching the query. Briefly, a Bloom filter is an array of bits used to represent some set A. The filter is computed by obtaining n indexes for each member of A, typically via n different hashing functions, and setting the bit at each index to 1. Then, given a Bloom filter, we can ask, is some element x a member of A by com-

puting n indexes for x and checking whether those bits are 1. Bloom filters can give *false positives* but never *false negatives*. Further, membership tests are performed in constant time.

PlanetP also queries the appropriate brokers, which can return XML data snippets[2] or snippets containing URLs pointing to where matching data might be found. Returned XML snippets are kept in a list of matching snippets. Nodes pointed to by URLs are added to the candidate list. Once all brokers have been contacted, the querying peer forwards the query to all candidate nodes. XML snippets returned by candidate nodes are added to the list of matching snippets. When all candidates have replied, the list of matching snippets is returned to the caller.

Handling conjunctions is straightforward. When querying a candidate based on its Bloom filter, the entire query is sent and so matching snippets already satisfy the conjunction. For the brokers, the querying peer maintains a re-assembly buffer. By maintaining a hash for each snippet in this buffer, PlanetP can compare new matching snippets with the ones already present.

Persistent queries use the same language as regular queries, with the difference being that they can upcall the user asynchronously. When posting a persistent query, the user provides an object that will be invoked whenever a new matching snippet is found. Persistent queries can be matched by both the brokers or the Bloom filters; every time a new bloom filter is received, PlanetP tries to match all the local queries against it. Since we can't tell which keys have been added to the filter, PlanetP might upcall the user repeatedly with an already matched query. We believe that applications have better means to resolve this type of conflicts.

2.4 Interface

PlanetP currently supports the following interface:

GoOnline(peers) `peers` is a set of members of the community, some of which should currently be online. A call to this function tells PlanetP to contact the given peers and join the community. Note that PlanetP currently assumes each peer can only be a member in one community. We will extend this framework to allow a user to join multiple communities at once in the near future.

GoOffline() Gracefully leave the community.

[2]Peers can store data directly in the brokerage service, which allows important content to be shared even when the owning peer is off-line. Of course, we limit the size of the XML data that can be published this way to prevent overload of the brokers.

Publish(important_keys, all_keys, snippet, timeout) Share `snippet` with the community. `all_keys` contains the list of all keys that should map to `snippet`; that is, any query looking for a key contained in `all_keys` should find `snippet`. `important_keys` contains a subset of the keys in `all_keys` that should be used to publish `snippet` to the brokerage service. If `timeout` is nonzero, then the brokers will drop the snippet after time `timeout`.

Query(query_string) Find all published XML snippets that matches the query `query_string`.

Persistent_Query(query_string, call_back) Find all published XML snippets that matches the query `query_string`. Additionally, make the query persistent; invoke the `call_back` every time there is a new snippet that matches `query_string`.

This is not intended to be the final API for PlanetP. In many ways, this API was shaped by our design and implementation of BreezeFS. We expect this API to change significantly over time.

3 PlanetP Performance

In this section, we assess PlanetP's performance by running a PlanetP community on two clusters of PCs. Note that PlanetP's most significant contribution is its ability to ease the management and sharing of data distributed across hundreds (or even thousands) of devices; absolute performance is a secondary concern (although PlanetP must be efficient enough for users to want to use it). PlanetP's effectiveness in providing this functionality can only be evaluated with use, however. Here, we assess PlanetP's performance to evaluate its ability to scale and to show that users can expect reasonable performance.

3.1 Experimental Environment

We run a PlanetP community on two clusters of PCs, one with 6 quad 500 Mhz Pentium III Xeon machines and one with 8 800 MHz Pentium III machines. The Xeon machines have 1 GB of memory each while the others have 512 MB each. All machines are interconnected by a 100 Mb/s switch Ethernet LAN. All machines are running Red Hat Linux 7.1, kernel 2.2.

PlanetP is written entirely in Java and currently stands at around 7000 lines of code. Our experiments were performed on the BlackDown Java JVM, version 1.3.0. The resource requirements of the Java JVM and its instability effectively limited us to about 10 peers per single-processor

Operation	Cost (ms)
Bloom filter insertion	44.8 + (0.012 * no. keys)
Bloom filter search	0.4 + (0.011 * no. keys
Bloom filter compress	51.2 + (0.002 * no. keys in filter)
Bloom filter decompress	36.5 + (0.002* no. keys in filter)
XMLStore insertion	245.5 + (0.043 * no. keys)
XMLStore search	0.6 + (0.0001 * no. keys)

Table 1. *Costs of PlanetP's basic operations. Each cost is presented as a fixed overhead plus a marginal per key overhead; for example, the cost for inserting n keys into a Bloom filter is $44.8ms + 0.012n$.*

machine and 20 per quad-process machine. This gave us a maximum community of about 200 peers.

To coordinate the experiments, we implemented a central coordinator. The coordinator runs on a separate machine but does spawn a slave daemon on each machine running PlanetP. Each slave process is responsible for running and communicating with the PlanetP peers on that machine. All the coordinator processes were listening on a multicast socket that was used for communicating commands from the master coordinator.

3.2 Micro Benchmarks

We start by measuring the costs of PlanetP's basic operations, the manipulation of Bloom filters and managing the XMLStore. Table 1 lists these operations and the measured costs. These measurements were performed on one of the 800 MHz PIII machines. We observe that while using Java allowed us to shortened the development time of PlanetP—for example, we were able to use the Java Collections Framework to implement most of the more complex data structures—it extracts a cost in performance. Most of the basic operations have fixed overheads of several to tens of milliseconds, and a marginal per key cost of several to tens of microseconds.

Currently, we are using constant size Bloom filters because we are investigating whether it is possible to combine several filters when diffusing information to reduce bandwidth consumption. One drawback of using fixed-size Bloom filters is that we must choose a size large enough to summarize the largest XMLStore without introducing too many false positives. To reduce bandwidth consumption, PlanetP compresses the Bloom filters when diffusing them. The compression scheme is a run-length compression that uses Golomb codes [12] to encode runs. Figure 2 shows that this compression scheme is quite effective at compressing Bloom filters.

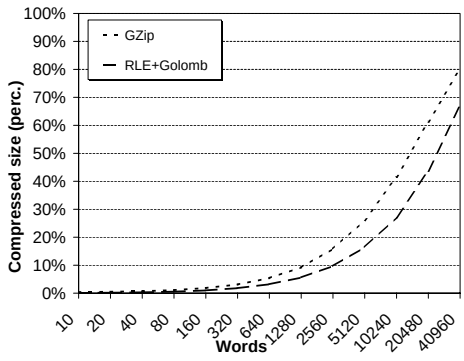

Figure 2. *Normalized size of compressed filters vs. different number of keys in the filter. We are compressing a fixed-size filter of 50,000 bytes, which should give approximately a 5% error rate for summarizing 50,000 words. We show that the Golomb-based run-length encoding is quite effective when compared to gzip, on average uses half of its size.*

3.3 The Cost to Join

We now assess the expense of having new members join an established community. To do this, we start a community of 100 peers and wait until their views of membership is consistent. Then, n peers will attempt to join the community; we measure the time required until all members have a consistent view of the community again as well as the required bandwidth during this time. For this experiment, each peer was set to share 1000 keys with the rest of the community through their Bloom filters. All new members will join the information brokerage service but there are no keys published to this service.

Figure 3 plots the time to reach consistency vs. the number of joining peers for our diffusion algorithm as well as the original name dropper algorithm. Figure 4 shows the total bandwidth used per peer during this process. Finally, Figure 5 plots the average per-node bandwidth against the number of joining members.

Our results show that even when the community *doubles in size*, members reaches stability in several hundred seconds. While this number will likely increase for a WAN-connected community, a time to reach global consistency of around tens of minutes seem quite reasonable when your community size changes by 100%. The growth shown in Figures 3 and 4 is consistent with the expected running time of the underlying name dropper algorithm. However, note that our optimizations has significantly reduced the time as well as the bandwidth required to reach global consistency.

Finally, observe that the average bandwidth required during

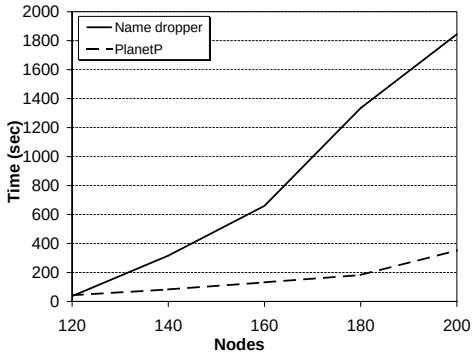

Figure 3. *Average time required to reach a stable state when joining different amounts of nodes on a stable network of 100 nodes.*

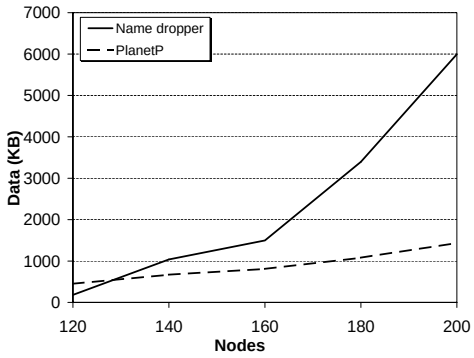

Figure 4. *Average data sent by each node when joining different amounts of nodes on a stable network of 100 nodes.*

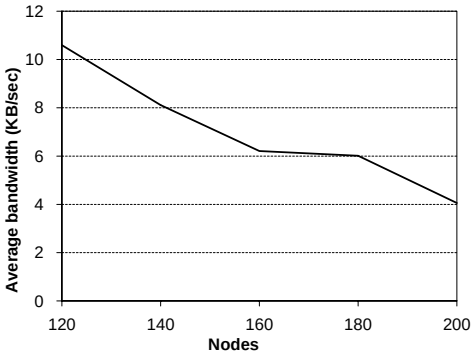

Figure 5. *Average bandwidth used per node per second when adding different amounts of nodes to an stable group of 100 nodes.*

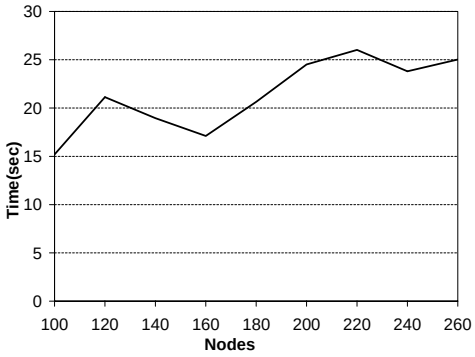

Figure 6. *Average time needed to propagate an updated Bloom filter throughout the community.*

this period of high diffusion activity is relatively constant vs. the number of joining members — in fact, it is decreasing. This is because the gossiping interval is effective at spacing out the diffusion. We are trading the time where views may be inconsistent for lower average bandwidth usage. The success of this trade-off bodes well for PlanetP's scalability to the regime that we are targeting, on order of 1000 peers.

3.4 Diffusion

In this section, we examine the time required to diffuse a Bloom filter in a stable community. This is important because it gives an estimate of how long peers must be in contact in order to update information that are hoarded at each peer for off-line operation.

For this study, we started communities of sizes ranging from 100 to 260. Once the community is stable, the coordinator injects a new piece of data on a random peer and measures the time it takes for that peer to diffuse its new Bloom filter throughout the entire community.

Figure 6 plots the measured times against community size. The slow growth in time is quite promising for PlanetP's scalability.

3.5 Information Brokerage Service

Finally we wanted to test the performance of the brokerage service in the presence of joins and departures. In this study, we start a community of 160 peers sharing 1000 keys in the brokerage service and wait until it achieves stability. Then, the coordinator randomly selects nodes and tells them to leave the community and return in 180 seconds; when a peer leaves, it forward all brokerage information correctly. When

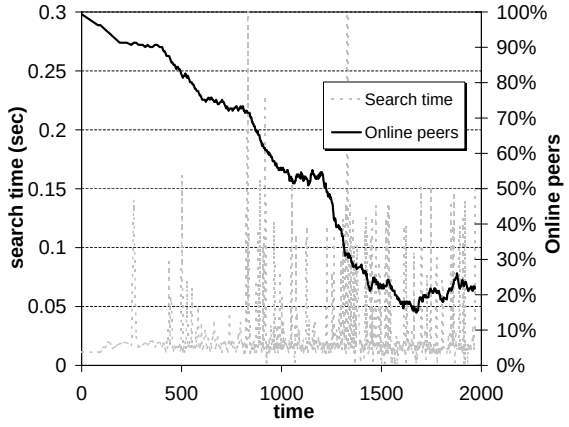

Figure 7. *Search time need by the brokers network to find a key. In this study a varying percentage of the nodes are leaving and joining the network.*

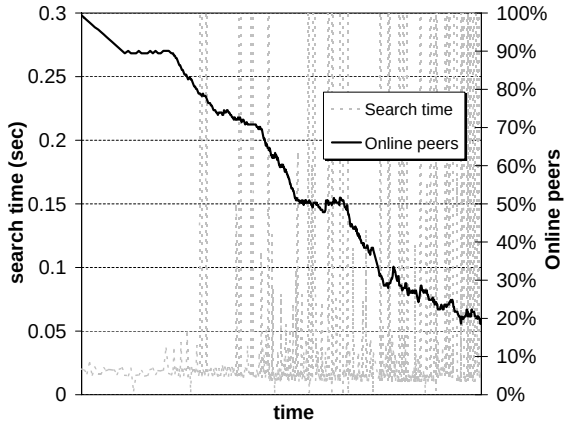

Figure 8. *Search time need by the brokers network to find a key. In this study a varying percentage of the nodes are leaving and joining the network. To overcome temporal inconsistencies every node tries 3 times to get the key before giving up*

a peer rejoins, it starts with an empty Local Directory.

Once the test bed is ready, the coordinator asks random peers that are active to query for a random key from the original 1000 and measures the time required for a reply to come. Figure 7 plots the result. In this Figure, the *Online peers* curve gives the number of peers active vs. time. The *Search time* curve gives the search time for the sequence of queries. As the percentage of active peers decrease, this means that a larger number is joining and leaving the community at a given instant in time. Thus, at the extreme right of the curve, about 80% of the members are leaving and joining while only 20% are active at any one time.

Clearly, as flux increases past about 25% of the community joining and leaving, search time starts to increase significantly. On the other hand, only 22% of the queries over the entire time of the experiment did not find the matching data because of temporal inconsistencies. The bulk of these failures occurred when 50% or more of the community is joining/leaving.

Figure 8 shows the same experiment except that the querying peer retries a failed query three times. While the search time still increases significantly when flux in membership becomes too great, the fraction of failed queries drops to 5%. The bulk of these failures occurred when 80% of the community is joining/leaving.

This experiment shows that the brokerage service is quite robust in the face of dynamic changes to the community's membership. Under normal circumstances, one would not expect over 50% of the members in the community to be actively joining or leaving at once.

4 BreezeFS

We have implemented BreezeFS, a middleware semantic file system, on top of PlanetP to validate its utility. BreezeFS provides similar functionality to the semantic file system defined by Gifford et al. [11]. BreezeFS's novelty lies in the fact that it provides for content querying across a dynamic community of users, where each user's data is potentially distributed among a multitude of devices, without requiring centralized indexing. We call BreezeFS a middleware system because it does not manage storage directly. Files are stored using the local file system of each device; files to be shared with the community are *published* to BreezeFS, which then uses PlanetP to make it possible for the entire community to search for these files based on its content.

Like the semantic file system, a directory is created in BreezeFS whenever the user poses a query. BreezeFS creates links to files that match the query in the resulting directory. If a file is created or modified such that it matches

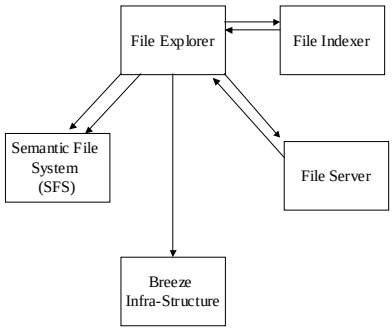

Figure 9. *Overview of File Explorer.*

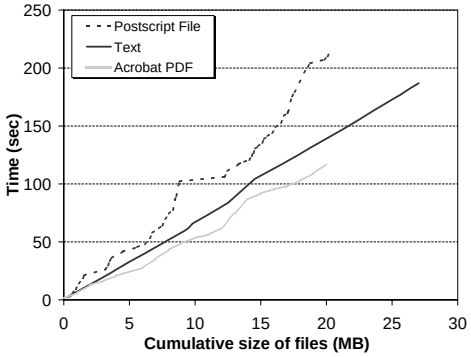

Figure 10. *Comparison of Time Required to index vs. Total cumulative size of files.*

some query, BreezeFS will update the directory to have a link to this file. Currently, query-based directories in BreezeFS are read-only. That is, only BreezeFS is allowed to change the content of the directory, users must use the *publish* operation to add new files. We are currently working to remove this limitation [3].

Building a query-based sub-directory is equivalent to refining the query of the containing directory. For example, if the user has created a directory *Query:Breeze* and then creates a sub-directory *Query:file system*, the content of the sub-directory would include all files that match the query *Breeze and file system*.

Figure 9 shows BreezeFS's basic architecture, which is comprised of four components: the Explorer, File Indexer, File Server, Semantic File System. When the user publishes a file, BreezeFS first passes it through the File Indexer, which takes the textual contents of the file and separates them into a set of keys. Currently, the File Indexer can index text, PDF, and postscript files; as noted in [11], the functionality of the File Indexer can be arbitrarily extended by users.

Once the published file has been indexed, BreezeFS gives the local pathname of the file to the File Server, which is basically a very simple web server. When given a file, the File Server returns a URL. When a GET request arrives for this URL, the File Server returns the content of the published file.

Finally, BreezeFS embeds the URL in a XML snippet and publishes to PlanetP along with the extracted keys. BreezeFS always uses the *Persistent_Query* call so that PlanetP will upcall into BreezeFS when changes in the files across the community should cause changes in some local query-based directory. Also, recall that PlanetP allows two sets of keys to be published with a snippet; one that will be published with the information brokerage network and one that will be published through the slower diffusion of Bloom filters. BreezeFS uses the top 10% of keys that have the highest counts as the former set. All keys are published in the latter set. In this way, a member of the community can find a file in a very short time after publication if he is searching for one of the key that appears most often in the file. For other keys, the newly published file can only be found when the new Bloom filter has been diffused throughout the system. Although BreezeFS's scheme for choosing the "important" keys is very simple, for structured documents it may be possible to use more complex methods like the ones used in WWW search engines. For example Google [2] uses a combination of font size, capitalization and position to rank the importance of a word with respect to the page where it resides. We leave this as future work.

BreezeFS automatically updates directories for addition via PlanetP's persistent queries. Updates for removal—that is, when a file is deleted by its owner or modified to no longer map to the directory's query—is more difficult. Whenever the user opens a directory, BreezeFS checks the last time that the directory was updated. If this time is greater than a fixed threshold, BreezeFS reruns the entire query to get rid of stale files.

Given PlanetP, we were able to implement BreezeFS in less than two weeks, with much of the first week given to designing BreezeFS graphical interface.

5 BreezeFS Performance

To evaluate the feasibility of using BreezeFS in everyday computing for sharing of information, we wanted to get an idea of how large the set of keys for each user might get and the cost of indexing the corresponding files.

[3] In fact, we are working to extend BreezeFS's capabilities to be similar to HAC [13].

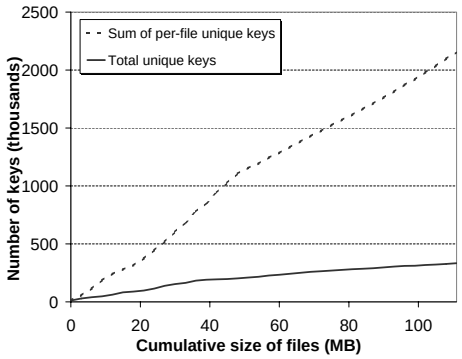

Figure 11. *Number of keys found when indexing different amounts of text files.*

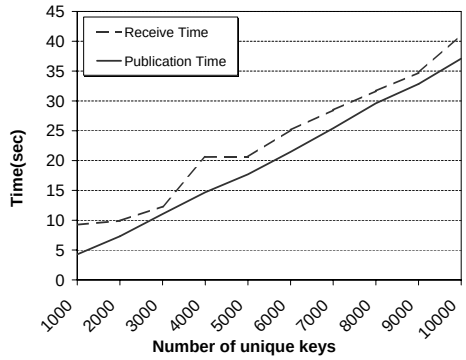

Figure 12. *Time it takes for publishing information.*

To test this, we used three set of files; one consisting of postscript files, another of acrobat pdf files, and one of text files. The postscript files and pdf files were obtained from http://citeseer.com and the text files were obtained from the Guttenberg Project (http://promo.net/pg/).

Figure 10 plots the indexing time against the cumulative size of sets of files being indexed when running on an 800 MHz PIII PC. Postscript and PDF files are converted to text files first using ps2ascii and pdftotext. This graph shows that indexing can be done approximately at the rate of 0.15 MB/sec. This implies that if we are willing to use 10% of our computing power to let BreezeFS index files that have been updated (or created) in the background, we can index 0.015 MB/sec (over 1 GB/day).

Figure 11 shows the number of keys generated by parsing approximately 118 MB of text files. The *Per File* line gives the total number of keys that would be published with PlanetP on a per file basis. The *Total* line indicates the number of unique keys across all of the files. This provides an approximation of the upper bound on how many keys PlanetP must deal with per node, which affect the size of the Bloom filters.

It should be noted that the files used for this experiment contained digital books in at least four different languages; English, German, Spanish, and French. These files also are considered the worst case for publishing since they are comprised completely of text content; there is no space lost to formatting or graphics.

By using at least four languages and approximately 120 MBytes of data, the number of overall unique keys does not exceed 350,000. This implies a maximum size of several hundred KB for the Bloom filters. In the current implementation, we are using 50 KB Bloom filters. While not huge, these sizes may strain the capacities (memory and bandwidth) of devices with limited resources. We are currently exploring methods to reduce the overheads of working with Bloom filters.

Finally, Figure 12 shows the costs of publishing a set of keys to PlanetP's brokerage service. In this experiment, we ran a community of 20 peers and measured the time required to publish files with varying number of matching keys. The time shown for BreezeFS is the time required for a peer with a persistent query posted for the last key that is published to the brokerage service to be notified about the new file. If we keep the set of keys to be published to the brokerage service small (say on the order of several hundred keys), then publication is quite fast; about 400 ms for 100 keys. Also, we believe that this number can be heavily optimized. Currently, keys are published to brokers one at a time without any concurrency; hence the linear increase in publication time.

6 Related Work

Infrastructural support for P2P system has recently become a popular research topic (e.g., [22, 26, 19, 24]). Much of this work has focused on scaling to very large storage systems (e.g., network of 10^6 peers), and thus, has concentrated on scalable mechanisms for id to object mapping. For example Pastry [22] and Tapestry [26] provide similar features to PlanetP's information brokerage service, although their service is based on the Plaxton et al.'s approach [18]. PlanetP is different from these systems in that (a) we focus on much smaller communities, which leads to significantly different design decisions, and (b) we focus on content-based information retrieval, as opposed to support for more traditional file system services.

The most similar approach to ours is Chord [24]. Both PlanetP and Chord use consistent hashing [15] to map keys to brokers. Chord, however, also focuses on scaling to very

large communities. Thus, they concentrate on how to find information when members do not have a global view of the system. Thus, PlanetP is different than Chord in the same ways that it differs from Pastry and Tapestry.

Finally, CAN [19] is another id to object mapping algorithm designed for very large P2P systems. We could have used CAN but consistent hashing was more compatible with our targeted environment, since we can leverage the fact that each member has a view of the entire system to optimize performance.

Because of all the added functionality, PlanetP resembles previous work done on tuple spaces [10, 25, 17] and publisher-subscriber models [21, 9]. The former introduced the idea of space and time decoupling by allowing publishers to post tuples without knowing who the receiver is or when it will pick up the tuples. The latter added the concept of flow decoupling, which means that a node does not need to poll for updates; rather, it will be notified asynchronously when an event occurs. PlanetP is the first P2P infrastructure that has tried to leverage functionality from both of these bodies of work. Previous work in those areas had relied on assumptions like broadcast/multicast [23] or server based schemes [17] to communicate among members. More recently system like Herald [3] have started to study self organizing solutions in environments similar to ours. They have proposed building a publish/subscribe system by using replicated servers on P2P networks.

PlanetP heavily relies on previous work done on the area of consistent hashing [15]. We have implemented a version similar to [4] in which we don't assume that malicious users will try to unbalance the key distribution by picking non random keys. On this first implementation we have used MD5 to hash keys so we can claim that is cryptographicaly hard for an adversary to choose keys that will overload a particular node. Similarly to what was done in Chord we have extended the original consistent hashing algorithm to be able to support concurrent node joins and leaves.

Several file systems, or file oriented applications, have been built using various peer to peer infrastructures as their framework, including [16, 8, 22, 5]. BreezeFS is different than these systems in that we were interested in providing a semantic file system, as opposed to traditional hierarchical directory-based systems.

With respect to the design of file systems, PlanetP is closest to the Semantic File System [11] and the HAC File System [13]. In fact, these two systems have provided many of the functionalities that we are trying to imbue BreezeFS with. Our main contribution is a design and implementation that will support sharing (and hoarding) across ad hoc and dynamic communities.

7 Conclusions

A significant challenge looms on the horizon for designers of storage and file systems: how to enable users to effectively manage, use, and share huge amounts of data stored across a multitude of devices. We have described PlanetP, a novel semantically indexed, P2P storage system, and BreezeFS, a semantic file system built on top of PlanetP. PlanetP makes two novel design choices to meet the above challenge. First, PlanetP concentrates on content-based querying for information retrieval and assumes that the unit of storage is a snippet of XML, allowing it to index arbitrary data for search and retrieval, regardless of the applications used to create and manipulate the data. Second, PlanetP adopts a P2P approach, avoiding centralization of storage and indexing. This makes PlanetP particularly suitable for information sharing among ad hoc groups of users, each of which may have to manage data distributed across multiple devices.

We have studied PlanetP's performance on a cluster of machines allowing community sizes of up to 200 peers. Results indicate that PlanetP will likely scale well up to our targeted size of around 1000 peers. Aside from scalability issues, we believe that the ability to organize and locate information based on content is the most important aspect of PlanetP and BreezeFS. This belief is guided by the fact that many of us routinely use web search engines to locate information. In fact, we often don't even bother to use bookmarks, which are organized with the familiar hierarchical structure. Rather, we rely on search engines to consistently return relevant documents out of the billions of choices when given a few search terms. PlanetP and BreezeFS aim to carry this paradigm to ad hoc groups of users to ease the management and sharing of data distributed across multiple devices.

8 Future Work

We intend to pursue at least four directions in continuing work on PlanetP and BreezeFS. First, we will explore how to go beyond PlanetP's requirement for exact key matching in queries; two possibilities include the use of phonetic string matching techniques and stemming to be able to match queries with approximate words. Second, we will explore adding semantics to the query language, including an awareness of XML tags, and extending the query language significantly. Third, we will explore extending BreezeFS to include functionalities provided by systems such as HAC. Finally, we need to study security implications of PlanetP and BreezeFS.

References

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

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[3] L. Cabrera, M. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[4] L. D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Department of EECS, MIT, 1998.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[7] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 35–46, May 2001.

[9] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. Technical Report DSC ID:2000104, EPFL, 2001.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O. Jr. Semantic File Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[12] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 1966.

[13] B. Gopal and U. Manber. Integrating Content-Based Access Mechanisms with Hierarchical File System. In *In the Proceedings of the 3th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–278, 1999.

[14] M. Harchol-Balter, F. T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Symposium on Principles of Distributed Computing*, pages 229–237, 1999.

[15] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, November 2000.

[17] T. Lehman, S. McLaughry, and P. Wyckoff. Tspaces: The next wave. Hawaii Intl. Conf. on System Sciences (HICSS-32), January 1999.

[18] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[20] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In Proceedings of the 2000 USENIX Annual Technical Conference, June 2000.

[21] D. S. Rosenblum and A. L. Wolf. A design framework for internet-scale event observation and notification. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 344–360. Springer–Verlag, 1997.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[23] A. Rowstron and A. Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar 96*, volume 1123, pages 511–513. Springer-Verlag, Berlin, 1996.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[25] Sun Microsystems. Javaspaces specification. http://java.sun.com/products/javaspaces/, 1998.

[26] Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.