Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility

Antony Rowstron[1] and Peter Druschel[2]

[1]Microsoft Research Ltd., St. George House, 1 Guildhall Street, Cambridge, CB2 3NH, UK.
[2]Rice University, 6100 Main Street, MS 132, Houston, TX 77005-1892, USA*

Abstract

This paper presents and evaluates PAST, a large-scale peer-to-peer persistent storage utility. PAST is based on a self-configuring, Internet based overlay network of storage nodes that cooperatively route file queries, store multiple replicas of files, and cache additional copies of popular files.

In the PAST system, storage nodes and files are each assigned uniformly distributed identifiers, and replicas of a file are stored at nodes whose identifier matches most closely the file's identifier. This statistical assignment of files to storage nodes approximately balances the number of files stored on each node. However, non-uniform storage node capacities and file sizes require more explicit storage load balancing to permit graceful behavior under high global storage utilization; and, non-uniform popularity of files requires caching to minimize fetch distance and to balance the query load.

We present and evaluate PAST, with an emphasis on its storage management and caching system. Extensive trace-driven experiments show that the system minimizes fetch distance, that it balances the query load for popular files, and that it displays graceful degradation of performance as the global storage utilization increases beyond 95%.

1 Introduction

Peer-to-peer Internet applications have recently been popularized through file sharing applications like Napster, Gnutella and FreeNet [1, 2, 2, 3]. While most of the attention has been focused on the copyright issues raised by these particular applications, peer-to-peer systems in general have many interesting technical aspects like scalability, completely decentralized control, self configuration and adaptation. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric.

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [4, 2, 5, 6, 7, 8]. We are developing PAST, an Internet based, peer-to-peer global storage utility, which aims to provide strong persistence, high availability, scalability, content integrity/privacy and, if desired, anonymity of clients and storage providers.

The PAST system is composed of a set of nodes, where each node is capable of storing files and of routing client requests to insert or retrieve a file, or to reclaim the storage associated with a file. The nodes form a self-organizing network. Inserted files are replicated across multiple nodes for availability. With high probability, the set of nodes over which a file is replicated is diverse in terms of geographic location, ownership, administrative entity, network connectivity, rule of law and so forth.

While PAST offers persistent storage services, its access semantics differ from that of a conventional filesystem. Files stored in PAST are associated with a *fileId* that is quasi-uniquely associated with the file's content, name and owner[1]. This implies that files stored in PAST are *immutable* since a modified version of a file cannot be written with the same fileId as its original. Files can be shared at the owner's discretion by distributing the fileId and, if necessary, a decryption key.

An efficient routing protocol, called *Pastry* and previously described in [9], ensures that client requests to *insert* or *reclaim* a file are routed to each node that stores the file. Client requests to *retrieve* a file are routed to the node that is "closest in the network"[2] to the client that

*Work done in part while visiting Microsoft Research, Cambridge, UK.

[1]The fileId is based on a secure hash of the file's content, name and owner. Therefore, it is extremely unlikely that files that differ in content, name, or owner have the same fileId.

[2]The notion of network proximity may be based on geographic lo-

issued the request, among the live nodes that store the requested file. The number of PAST nodes traversed, as well as the number of messages exchanged while routing a client request is at most logarithmic in the total number of PAST nodes in the system.

In this paper, we focus on the storage management and caching aspects of PAST. The rest of this paper is organized as follows. In Section 2, we present an overview of the PAST architecture and briefly describe Pastry, PAST's content location and routing scheme. Section 3 describes the storage management and Section 4 the mechanisms and policies for caching in PAST. Results of an experimental evaluation of PAST are presented in Section 5. Related work is discussed in Section 6 and we conclude in Section 7.

2 PAST Overview

This section provides an overview of the PAST architecture. PAST can be thought of a collection of nodes that form an overlay network in the Internet. Any host connected to the Internet can act as a PAST node by installing the PAST node software. Minimally, a PAST node acts as an access point for a user. Optionally, a PAST node may also contribute storage to PAST and participate in the routing of requests within the PAST network.

Users who wish to store files in PAST or contribute storage to PAST (as opposed to merely retrieve files) must be in possession of a smartcard issued by a certification authority. The smartcard performs user and node authentication and a small number of other security sensitive functions, such as generating PAST node identifiers (nodeIds) and file identifiers (fileIds).

The PAST system exports the following set of operations to its clients:

• `fileId = Insert(name, file, k)` stores a file at a user-specified number k of diverse nodes within the PAST network. The operation produces a 160-bit identifier (fileId) that can be used subsequently to identify the file. The fileId is computed as the secure (SHA-1) hashcode of the file's name, content, the client id and an integer-valued salt. This choice ensures (with very high probability) that files that differ in name, content, or owner have unique fileIds.

• `file = Lookup(fileId)` reliably retrieves a copy of the file identified by fileId if it exists in PAST and if one of the k nodes that store the file is reachable via the Internet. The file is normally retrieved from a live node near the PAST node issuing the lookup, among the nodes that store the file. Proximity is measured here in terms of a scalar, application defined metric, such as the number of IP network hops or network delay.

• `Reclaim(fileId)` reclaims the storage occupied by the k copies of the file identified by fileId. Once the operation completes, PAST no longer guarantees that a lookup operation will produce the file.

Each PAST node is assigned a 128-bit node identifier (nodeId). The nodeId indicates a node's position in a circular namespace, which ranges from 0 to $2^{128} - 1$. This nodeId is chosen randomly by the node operator's smartcard when a node joins the system. The smartcard uses a high-quality uniform random number generator and digitally signs the nodeId to ensure that only truly randomly chosen nodeIds are used in the system.

This process ensures that there is no correlation between the value of the nodeId and the node's geographic location, owner, organization, or network connectivity. It follows then that a set of nodes with adjacent nodeIds are highly likely to be diverse in all these aspects. Such a set is therefore an excellent candidate for storing the replicas of a file, as the nodes in the set are unlikely to conspire or be subject to correlated failures.

During an insert operation, PAST stores the file on the k PAST nodes whose nodeIds are numerically closest to the 128 most significant bits of the file's fileId. This invariant is maintained over the lifetime of a file, despite the arrival, failure and recovery of nodes. For the reasons outlined above, with high probability, the k replicas are stored on a diverse set of PAST nodes.

Another invariant is that both the set of existing nodeId values as well as the set of existing fileId values are uniformly distributed in their respective domains. The former follows from the random choice of nodeIds, the later follows from the use of a secure hash function to generate fileIds. Jointly, these properties ensure that the number of files stored by each PAST node is roughly balanced. This fact provides only an initial approximation to balancing the storage utilization among the PAST nodes. Since files differ in size and PAST nodes differ in the amount of storage they provide, additional, explicit means of load balancing are required. These methods are described in Section3.

The number k is chosen to meet the availability needs of a file, based on the expected failure rates of individual nodes. However, popular files may need to be maintained at many more nodes in order to meet and balance the query load for the file and to minimize latency and network traffic. PAST adapts to query load by caching additional copies of files in the unused portions of PAST node's local disks. Unlike the k primary replicas of a file, such cached copies may be discarded by a node at any time. Caching in PAST is discussed in Section 4.

PAST relies on an integrated request routing and content location scheme, called *Pastry*, to route requests to

cation, number of network hops, bandwidth, delay, or a combination of these and other factors.

the PAST nodes that store the associated file. Pastry is described and evaluated in an earlier paper [9]. To make this paper self-contained, we provide next a brief overview of Pastry.

2.1 Pastry

The fundamental capability that Pastry provides is to efficiently route messages among the nodes in the system. Specifically, given a fileId, Pastry routes the associated message towards the node whose nodeId is numerically closest to the 128 most significant bits (msb) of the fileId, among all live nodes. Given the invariant that a file is stored on the k nodes whose nodeIds are numerically closest to the 128 msbs of the fileId, it follows that a file can be located unless all k nodes have failed simultaneously (i.e., within a recovery period).

Pastry is highly efficient, scalable, fault resilient and self-configuring. Assuming a PAST network consisting of N nodes, Pastry can route to the numerically closest node to a given fileId in less than $\lceil log_{2^b} N \rceil$ steps on average (b is a configuration parameter with typical value 4). With concurrent node failures, eventual delivery is guaranteed unless $\lfloor l/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously (l is a configuration parameter with typical value 32).

The tables required in each PAST node have only $(2^b - 1) * \lceil log_{2^b} N \rceil + 2l$ entries, where each entry maps a nodeId to the associated node's IP address. Moreover, after a node failure or the arrival of a new node, the invariants in all affected routing tables can be restored by performing $O(log_{2^b} N)$ remote procedure calls (RPCs). In the following, we give a brief overview of the Pastry routing scheme.

For the purpose of routing, nodeIds and fileIds are thought of as a sequence of digits with base 2^b. A node's routing table is organized into levels with $2^b - 1$ entries each. The $2^b - 1$ entries at level n of the routing table each refer to a node whose nodeId shares the present node's nodeId in the first n digits, but whose $n+1$th digit has one of the $2^b - 1$ possible values other than the $n+1$th digit in the present node's id. Note that an entry in the routing table points to one of potentially many nodes whose nodeId have the appropriate prefix. Among such nodes, the one closest to the present node (according to the proximity metric) is chosen in practice.

In addition to the routing table, each node maintains pointers to the set of l nodes whose nodeIds are numerically closest to the present node's nodeId, irrespective of prefix. (More precisely, the set contains $l/2$ nodes with larger and $l/2$ with smaller nodeIds). This set is called the *leaf set*. Figure 1 depicts the state of a hypothetical PAST node with the nodeId 10233102 (base 4), in a system that uses 16 bit nodeIds and a value of $b = 2$.

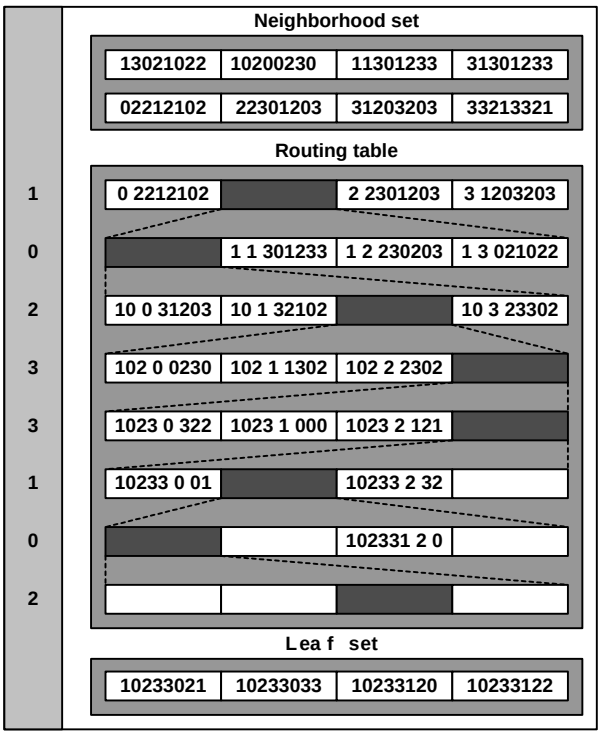

Figure 1: State of a hypothetical Pastry node with nodeId 10233102, $b = 2$. All numbers are in base 4. The top row of the routing table represents level zero. The neighborhood set is not used in routing, but is needed during node addition/recovery.

In each routing step, a node normally forwards the message to a node whose nodeId shares with the fileId a prefix that is at least one digit (or b bits) longer than the prefix that the fileId shares with the present node's id. If no such node exists, the message is forwarded to a node whose nodeId shares a prefix with the fileId as long as the current node, but is numerically closer to the fileId than the present node's id. It follows from the definition of the leaf set that such a node exists in the leaf set unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously.

Locality In the following, we turn our attention to the properties of the Pastry routing scheme with respect to the network proximity metric. Pastry can normally route messages to any node in $\lceil log_{2^b} N \rceil$ steps. Another question is what distance (in terms of the proximity metric) a message is traveling. Recall that the entries in the node routing tables are chosen to refer to the nearest node with the appropriate nodeId prefix. As a result, in each step a message is routed to the nearest node with a longer prefix match (by one digit). While this local decision process clearly can't achieve globally shortest routes, simulations have shown the the average distance travelled by a message is only 40% higher than the distance of the source

and destination in the underlying network [9].

Moreover, since Pastry always takes the locally shortest step towards a node that shares a longer prefix with the fileId, messages have a tendency to first reach a node, among the k nodes that store the requested file, that is near the client (according to the proximity metric). One experiment shows that among 5 replicated copies, Pastry is able to find the nearest copy in 76% of all lookups and it finds one of the two nearest copied in 92% of all lookups [9].

Node addition and failure A key design issue in Pastry is how to efficiently and dynamically maintain the node state, i.e., the routing table, leaf set and neighborhood sets, in the presence of node failures, node recoveries, and new node arrivals. The protocol is described and evaluated in full detail in [9].

Briefly, an arriving node with the newly chosen nodeId X can initialize its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message to the existing node Z with nodeId numerically closest to X. X then obtains the leaf set from Z, the neighborhood set from A, and the ith row of the routing table from the ith node encountered along the route from A to Z. One can show that using this information, X can correctly initialize it state and notify interested nodes that need to know of its arrival, thereby restoring all of Pastry's invariants.

To handle node failures, neighboring nodes in the nodeId space (which are aware of each other by virtue of being in each other's leaf set) periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed. All members of the failed node's leaf set are then notified and they update their leaf sets to restore the invariant. Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial. A recovering node contacts the nodes in its last known leaf set, obtains their current leafs sets, updates its own leaf set and and then notifies the members of its new leaf set of its presence. Routing table entries that refer to failed nodes are repaired lazily; the details are not relevant to the subject of this paper [9].

2.2 PAST operations

Next, we briefly describe how PAST implements the insert, lookup and reclaim operations.

In response to an *insert request*, the client's PAST computes the SHA-1 hashcode of the file's content. The node then chooses an integer valued salt and computes a fileId as the SHA-1 hashcode of the content hashcode, the file's textual name, the client's id (provided by the client's smartcard) and the salt. It then passes the resulting fileId, the content hashcode and the desired replication factor k to the client's smartcard. The smartcard debits the required storage against the client's quota, and issues a *write certificate*, which contains the fileId, content hashcode and replication factor.

The write certificate and the associated file are then routed via Pastry to the node A whose nodeId is closest to the fileId. That node then verifies the write certificate using its own smart card, recomputes the content hashcode and compares it with the content hashcode in the write certificate. If everything checks out, then the node accepts responsibility for a replica of the file and forwards the insert request to the other $k-1$ nodes with nodeIds numerically closest to the fileId.

Once all k nodes have accepted a replica, an acknowledgment is passed back to the client, to which each of the k closest nodes attach a *store receipt*. The client verifies the store receipts to confirm that the requested number of copies have been created. If anything goes wrong at any point during the insertion process, such as an illegitimate write certificate, corrupted content, or a failure to locate sufficient storage to store the k copies, an appropriate error indication is returned to the client.

In response to a *lookup request*, the client node routes an appropriate request message towards the node with nodeId closest to the requested fileId. As soon as the request message reaches a node that stores the file, that node responds with the file and does not route the request message further. Due to the locality properties of the Pastry routing scheme and the fact that file replicas are stored on k nodes with adjacent nodeIds, a lookup is very likely to find a replica that is near the client (according to the proximity metric).

A *reclaim request* proceeds analogous to an insert request. The nodes storing the file each issue and return a *reclaim receipt*, which the client then presents to its smartcard for a credit against its storage quota.

2.3 Security

While the details of security in PAST are beyond the scope of this paper, we provide here a brief overview. The smartcards associated with each node collectively form a PAST system's trusted computing base. Clients, node operators and the PAST software running in a node are not trusted.

The smartcards guarantee the integrity of nodeId and fileId assignments, thus preventing an attacker from controlling all nodes in a portion of the nodeId space, or overwhelming the system with file insertions in a portion of the fileId space. Store certificates prevent a malicious node from causing the system to create less than k diverse replicas of a file without the client noticing it. Write and reclaim certificates can help enforce client storage quotas and can prevent storage exhaustion.

The Pastry routing scheme can be randomized, thus

preventing a malicious node from repeatedly intercepting a request message, causing persistent failure of the requested operation. File integrity can be verified by clients and storage nodes since the fileId is based on a secure hash of the file's content. If desired, a client can ensure file privacy by encrypting the content.

In the following sections, we describe the storage management and the caching in PAST. The primary goal of storage management is to ensure the availability of files while balancing the storage load as the system approaches its maximal storage utilization. The goal of caching is to minimize client access latencies, to maximize the query throughput and to balance the query load in the system.

3 Storage management

PAST's storage management aims at allowing high global storage utilization and graceful degradation as the system approaches its maximal utilization. The aggregate size of file replicas stored in PAST should be able to grow to a large fraction of the aggregate storage capacity of all PAST nodes before a large fraction of insert requests are rejected or suffer from decreased performance.

In line with the overall decentralized architecture of PAST, an important design goal for the storage management is to rely only on local coordination among nodes with nearby nodeIds, to fully integrate storage management with file insertion, and to incur only modest performance overheads related to storage management.

The responsibilities of the storage management are to (1) balance the remaining free storage space among nodes in the PAST network as the system-wide storage utilization is approaching 100%; and, (2) to maintain the invariant that copies of each file are maintained by the k nodes with nodeIds closest to the fileId.

Goals (1) and (2) appear to be conflicting, since requiring that a file is stored on k nodes closest to its fileId leaves no room for any explicit load balancing. PAST resolves this conflict in two ways.

First, PAST allows a node that is *not* one of the k numerically closest nodes to D to alternatively store the file, if it is in the leaf set of one those k nodes. This process is called *replica diversion* and its purpose is to accommodate differences in the storage capacity and utilization of nodes within a leaf set. Replica diversion must be done with care, to ensure that the file availability is not degraded.

Second, *file diversion* is performed when A's entire leaf set is reaching capacity. Its purpose is to achieve more global load balancing across large portions of the nodeId space. A file is diverted to a different part of the nodeId space by choosing a different salt in the generation of its fileId.

In the rest of this section, we discuss causes of storage imbalance, state assumptions about per-node storage and then present the algorithms for replica and file diversion. Finally, we describe how the storage invariant is maintained in the presence of new node addition, node failure and recovery. An experimental evaluation of PAST's storage management follows in Section 5.

3.1 Causes of storage load imbalance

Recall that each PAST node Pastry maintains a leaf set, which contains the l nodes with nodeIds numerically closest to the given node ($l/2$ with larger and $l/2$ with smaller nodeIds). Normally, the replicas of a file are stored on the k nodes that are numerically closest to the fileId (k can be no larger than $(l/2) + 1$).

Consider the case where not all of the k closest nodes can accommodate a replica due to insufficient storage, but k nodes exist within the leaf sets of the k nodes that can accommodate the file. Such an imbalance in the available storage among the $l + k$ nodes in the intersection of the k leaf sets can arise for several reasons:

- Due to statistical variation in the assignment of nodeIds and fileIds, the number of files assigned to each node may differ.
- The size distribution of inserted files has high variance and may be heavy tailed.
- The storage capacity of individual PAST nodes differs.

Replica diversion aims at balancing the remaining free storage space among the nodes in each leaf set. In addition, as the global storage utilization of a PAST system increases, file diversion may also become necessary to balance the storage load among different portions of the nodeId space.

3.2 Per-node storage

We assume that the storage capacities of individual PAST nodes differ by no more than two orders of magnitude at a given time. The following discussion provides some justification for this assumption.

PAST nodes are likely to use the most cost effective hardware available at the time of their installation. At the time of this writing, this might be a PC with a small number of 60GB disk drives. Given that the size of the most cost effective disk size can be expected to double in no less than one year, a node with a typical, low-cost configuration at the time of its installation should remain viable for many years, i.e., its capacity should not drop below two orders of magnitude of the largest newly installed node.

Our assumption does not prevent the construction of sites that provide large-scale storage. Such a site would

be configured as a cluster of logically separate PAST nodes with separate nodeIds. Whether the associated hardware is centralized (large multiprocessor node with RAID storage subsystem or cluster of PCs, each with a small number attached disks) is irrelevant, although considerations of cost and fault resilience normally favor the latter. Even though the multiple nodes of a site are not administratively independent and may have correlated failures, the use of such sites does not significantly affect the average diversity of the nodes selected to store replicas of a given file, as long as the number of nodes in a site is very small compared to the total number of nodes in a PAST system.

PAST controls the distribution of per-node storage capacities by comparing the advertised storage capacity of a newly joining node with the average storage capacity of nodes in its leaf set. If the node is too large, it is asked to split and join under multiple nodeIds. If a node is too small, it is rejected. A node is free to advertise only a fraction of its actual disk space for use by PAST. The advertised capacity is used as a basis for the admission decision.

3.3 Replica diversion

The purpose of replica diversion is to balance the remaining free storage space among the nodes in a leaf set. Replica diversion is accomplished as follows.

When an insert request message first reaches a node with a nodeId among the k numerically closest to the fileId, the node checks to see if it can accommodate a copy of the file in its local disk. If so, it stores the file, issues a store receipt, and forwards the message to the other $k-1$ nodes with nodeIds closest to the fileId. (Since these nodes must exists in the node's leaf set, the message can be forwarded directly). Each of these nodes in turn attempts to store a replica of the file and returns a store receipt.

If a node A cannot accommodate a copy locally, it considers replica diversion. For this purpose, A chooses a node B in its leaf set that is not among the k closest and does not already hold a diverted replica of the file. A asks B to store a copy on its behalf, then enters an entry for the file in its table with a pointer to B, and issues a store receipt as usual. We say that A has diverted a copy of the file to node B.

Care must be taken to ensure that a diverted replica contributes as much towards the overall availability of the file as a locally stored replica. In particular, we must ensure that (1) failure of node B causes the creation of a replacement replica, and that (2) the failure of node A does not render the replica stored on B inaccessible. If it did, then every diverted replica would double the probability that all k replicas might be inaccessible. The node failure recovery procedure described in Section 3.5 ensures condition (1). Condition (2) can be achieved by entering a pointer to the replica stored on B into the file table of the node C with the $k+1$th closest nodeId to the fileId.

Results presented in Section 5 show that replica diversion achieves local storage space balancing and is necessary to achieve high overall storage utilization and graceful degradation as the PAST system reaches its storage capacity. The overhead of diverting a replica is an additional entry in the file tables of two nodes (A and C, both entries pointing to B), two additional RPCs during insert and one additional RPC during a lookup that reaches the diverted copy. To minimize the impact of replica diversion on PAST's performance, appropriate policies must be used to avoid unnecessary replica diversion.

3.3.1 Policies

We next describe the policies used in PAST to control replica diversion. There are three relevant policies, namely (1) acceptance of replicas into a node's local store, (2) selecting a node to store a diverted replica, and (3) deciding when to divert a file to a different part of the nodeId space. In choosing appropriate policies for replica diversion, the following considerations are relevant.

First, it is not necessary to balance the remaining free storage space among nodes as long as the utilization of all nodes is low. Doing so would have no advantage but incur the cost of replica diversion. Second, it is preferable to divert a large file rather than multiple small ones. Diverting large files not only reduces the insert overhead of replica diversion for a given amount of free space that needs to be balanced; taking into account that workloads are often biased towards lookups of small files, it can also minimize the impact of the lookup overhead of replica diversion.

Third, a replica should always be diverted from a node whose remaining free space is significantly above average to a node whose free space is significantly below average; when the free space gets uniformly low in a leaf set, it is better to divert the file into another part of the nodeId space than to attempt to divert replicas at the risk of spreading locally high utilization to neighboring parts of the nodeId space.

The policy for accepting a replica by a node is based on the metric S_D/F_N, where S_D is the size of a file D and F_N is the remaining free storage space of a node N. In particular, a node N rejects a file D if $S_D/F_N > t$, i.e., D would consume more than a given fraction t of N's remaining free space. Nodes that are among the k numerically closest to a fileId (primary replica stores) as well as nodes not among the k closest (diverted replica stores) use the same criterion, however, the former use a threshold t_1 while the latter use t_2, where $t_1 > t_2$.

There are several things to note about this policy. First, assuming that the average file size is much smaller than a node's average storage size, a PAST node accepts all but oversized files as long as its utilization is low. This property avoids unnecessary diversion while the node still has plenty of space. Second, the policy discriminates against large files , and decreases the size threshold above which files get rejected as the node's utilization increases. This bias minimizes the number of diverted replicas and tends to divert the large files first, while leaving room for small files. Third, the criterion for accepting diverted replicas is more restrictive than that for accepting primary replicas; this ensures that a node leaves some of its space for primary replicas, and that replicas are diverted to a node with significantly more free space.

A primary store node N that rejects a replica needs to select another node to hold the diverted replica. The policy is to choose the node with maximal remaining free space, among the nodes that are (a) in the leaf set of N, (b) have a nodeId that is not also one of the k nodes closest to the fileId, and (c) do not already hold a diverted replica of the same file. This policy ensures that replicas are diverted to the node with the most free space, among the eligible nodes. Note that a selected node may reject the diverted replica based on the above mentioned policy for accepting replicas.

Finally, the policy for diverting an entire file into another part of the nodeId space is as follows. When one of the k nodes with nodeIds closest to the fileId declines to store its replica, and the node it then chooses to hold the diverted replica also declines, then the entire file is diverted. In this case, the nodes that have already stored a replica discard the replica, and a negative acknowledgment message is returned to the client node, causing a file diversion.

3.4 File diversion

The purpose of file diversion is to balance the remaining free storage space among different portions of the nodeId space in PAST. When a file insert operation fails because the k nodes closest to the chosen fileId could not accommodate the file nor divert the replicas locally within their leaf set, a negative acknowledgment is returned to the client node. The client node in turn generates a new fileId using a different salt value and re-tries the insert operation.

A client node repeats this process for up to three times. If the insert operation still fails, the operation is aborted and an insert failure is reported to the application. Such a failure indicates that the system was not able to locate the necessary space to store k copies of the file. In such cases, an application may choose to re-try the operation with a smaller file size (e.g. by fragmenting the file) and/or a smaller number of replicas.

3.5 Maintaining replicas

PAST maintains the invariant that k copies of each inserted file are maintained on different nodes within a leaf set. This is accomplished as follows.

First, recall that as part of the Pastry protocol, neighboring nodes in the nodeId space periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed and Pastry triggers an adjustment of the leaf sets in all affected nodes. Specifically, each of the l nodes in the leaf set of the failed node removes the failed node from its leaf set and includes instead the live node with the next closest nodeId.

Second, when a new node joins the system or a previously failed node gets back on-line, a similar adjustment of the leaf set occurs in the l nodes that comprise the leaf set of the joining node. Here, the joining node is included and another node is dropped from each of the previous leaf sets.

As part of these adjustments, a node may become one of the k closest nodes for certain files; the storage invariant requires such a node to acquire a replica of each such file, thus re-creating replicas that were previously held by the failed node. Similarly, a node may cease to be one of the k nodes for certain files; the invariant allows a node to discard such copies.

In practice, requiring that a node requests replicas of files for which is has just become one of the k numerically closest nodes can be time-consuming and inefficient, given the current ratio of disk storage versus wide-area Internet bandwidth. This is particularly obvious in the case of a new node or a recovering node whose disk contents were lost as part of the failure.

To solve this problem, the joining node may instead install a pointer in its file table, referring to the node that has just ceased to be one of the k numerically closest to the fileId, and requiring that node to keep the replica. This process is semantically identical to replica diversion, and the existing mechanisms to ensure availability are reused (see Section 3.3.

A second issue can arise when a node fails and the storage utilization is so high that the remaining nodes in the leaf set are unable to store additional replicas. To allow PAST to maintain its storage invariants under these circumstances, a node asks the two most distant members of its leaf set (in the nodeId space) to locate a node in their respective leaf sets that can store the file. Since exactly half of the node's leaf set overlaps with each of these two node's leaf sets, a total of $2l$ nodes can be reached in this way. Should none of these nodes be able to accommodate the file, then it is unlikely that space can be found anywhere in the system and the number of replicas may

temporarily drop below k until more nodes or disk space become available.

The observant reader may have noticed at this point that maintaining k copies of a file in a PAST system with high utilization is only possible if the total amount of disk storage in the system is non-decreasing. If total disk storage were to decrease due to node and disk failures that are not eventually balanced by node and disk additions, the system will eventually exhaust all of its storage. Beyond a certain point, the system will be unable to re-replicate files to make up for replicas lost due to node failures.

Maintaining adequate resources and utilization is a problem in system like PAST that are not centrally managed. Any solution will have to provide strong incentives for users to balance their resource consumption with the resources they contribute to the system. PAST solves this problem by maintaining storage quotas using smartcards, thus ensuring that demand for storage cannot exceed the supply. A full discussion of these management and security aspects of PAST is beyond the scope of this paper.

3.6 File encoding

Storing k complete copies of a file is not the most storage-efficient method to achieve high availability. With Reed-Solomon encoding, for instance, adding m additional checksum blocks to n original data blocks (all of equal size) allows recovery from up to m losses of data or checksum blocks [10]. This reduces the storage overhead required to tolerate m failures from m to $(m+n)/n$ times the file size. By fragmenting a file into a large number of data blocks, the storage overhead for availability can be made very small.

Independent of the encoding, storing fragments of a file at separate nodes (and thereby striping the file over several disks) can also improve bandwidth. However, these potential benefits must be weighed against the cost (in terms of latency, aggregate query and network load, and availability) of having to contact several nodes to retrieve a file. These costs may outweigh the benefits for all but large files. We intend to explore this option in the future. The storage management issues discussed in this paper, however, are largely orthogonal to the choice of a file encoding and file striping.

4 Caching

In this section, we describe the cache management in PAST. The goal of cache management is to minimize client access latencies (fetch distance), to maximize the query throughput and to balance the query load in the system.

The k replicas of a file are maintained by PAST primarily for reasons of availability, although some degree of query load balancing and latency reduction results. To see this, recall that the k nodes with adjacent nodeIds that store copies of a file are likely to be widely dispersed and that Pastry is likely to route client lookup request to the replica closest to the client.

However, a highly popular file may demand many more then k replicas in order to sustain its lookup load while minimizing client latency and network traffic. Furthermore, if a file is popular among one or more loci of clients, it is advantageous to store a copy near each center of client interest. Creating and maintaining such additional copies is the task of cache management in PAST.

PAST nodes use the "unused" portion of their advertised disk space to cache file. Cached copies can be evicted and discarded at any time. In particular, when a node stores a new primary or redirected replica of a file, it typically evicts one or more cached files to make room for the replica. This approach has the advantage that unused disk space in PAST is used to improve performance; on the other hand, as the storage utilization of the system increases, cache performance degrades gracefully.

The cache insertion policy in PAST is as follows. A file that is routed through a node as part of a lookup or insert operation is inserted into the local disk cache if its size is less than a fraction c of the node's current cache size, i.e., the portion of the node's storage not currently used to store primary or diverted replicas.

The cache replacement policy in PAST is based on the GreedyDual-Size (GD-S) policy, which was originally developed for caching Web proxies [11]. GD-S maintains a weight for each cached file. Upon insertion or use (cache hit), the weight H_d associated with a file d is set to $c(d)/s(d)$, where $c(d)$ represents a cost associated with d, and $s(d)$ is the size of the file d. When a file needs to be replaced, the file v is evicted, such that H_v is minimal among all cached files. Then, H_v is subtracted from the H values of all remaining cached files.

If the value of $c(p)$ is set to one, the policy maximizes the cache hit rate of a single cache. To maximized the system-wide cache hit rate in PAST, the cost must take into account the utility of caching a file at a particular node. It follows from the Pastry routing scheme that the probability that a request for a file is routed through a node depends on how long a prefix is shared between the node's nodeId and the file's fileId. This suggests that one should assign costs to a file based on the length of the shared prefix.

Assuming that every node is equally likely to originate a request for a given file, the probability P_r that a node n will receive a request for fileId d is $P_r = 2^{bl}/N$, where l is the number of digits shared between nodeId n and fileId d. To maximize the system-wide cache hit rate, we

can therefore set the cost $c(d) = 2^{bl}$. Results presented in Section 5 show that this policy indeed yields a high global cache hit rate.

5 Experimental results

In this section, we present experimental results obtained with a prototype implementation of PAST. The PAST node software was implemented in Java. To be able to perform experiments with large networks of Pastry nodes, we also implemented a network simulation environment.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 4 GBytes of main memory, running True64 UNIX, version 4.0F. The Pastry node software was implemented in Java and executed using Compaq's Java 2 SDK, version 1.2.2-6 and the Compaq FastVM, version 1.2.2-4.

The Pastry nodes normally use Java remote object invocation (RMI) to communicate with each other. However, in all experiments reported in this paper, the Pastry nodes were configured to run in a single Java VM. This is largely transparent to the Pastry implementation—the Java runtime system automatically reduces communication among the Pastry nodes to local object invocations.

The experimental results are divided into those analyzing the effectiveness of the PAST storage management, and those examining the effectiveness of the caching used in PAST.

5.1 Storage

For the experiments exploring the storage management, two different workloads were used. The first consists of a set of 8 web proxy logs from NLANR[3] for 5th March 2001, which were truncated to contain 4,000,000 entries, referencing 1,863,055 unique URLs, totaling 18.7 GBytes of content, with a mean file size of 10,517 bytes, a median file size of 1,312 bytes, and a largest/smallest file size of 138 MBytes and 0 bytes, respectively. The second of the workloads was generated by combining file name and file size information from several file systems at the author's home institutions. The files were sorted alphabetically by filename to provide an ordering. The trace contained 2,027,908 files with a combined file size of 166.6 GBytes, with a mean file size of 88,233 bytes, a median file size of 4,578 bytes, and a largest/smallest file size of 2.7 GBytes and 0 bytes, respectively.

In all experiments, the number of replicas k for each file was fixed at 5, b was fixed at 4, and the number of PAST nodes was fixed at 2250. The number of replicas was chosen based on the analysis in [2]. The storage space contributed by each PAST node was chosen from a truncated normal distribution with mean m, standard deviation σ, and with upper and lower limits at $m + x\sigma$ and $m - x\sigma$, respectively.

Dist. name	m	σ	Lower bound	Upper bound	Total capacity
d_1	27	10.8	2	51	61,009
d_2	27	9.6	4	49	61,154
d_3	27	54.0	6	48	61,493
d_4	27	54.0	1	53	59,595

Table 1: The parameters of four normal distributions of node storage sizes used in the experiments. All figures in Mb.

Table 1 shows the values of m and σ for four distributions used in the first set of experiments. The lower and upper bound indicate where the tails of the normal distribution were cut, in the case of d_1 and d_2. For instance, the lower and upper bound were defined as $m - 2.3\sigma$ and $m + 2.3\sigma$, respectively. The mean storage capacities of these distributions are approximately a factor of 1000 below what one might expect in practice. This scaling was necessary to experiment with high storage utilization and a substantial number of nodes, given that the workload traces available to us have only limited storage requirements. Notice that reducing the node storage capacity in this way makes storage management more difficult, so our results are conservative.

The first set of experiments use the NLANR traces. The eight separate web traces were combined, preserving the temporal ordering of the entries in each log to create a single log. The first 4,000,000 entries of that log were used in sequence, with the first appearance of a URL being used to insert the file into PAST, and with subsequent references to the same URL ignored. Unlike otherwise stated, the node storage sizes were chosen from distribution d_1.

In the first experiment, both replica diversion and file diversion were disabled by setting $t_1 = 1$, $t_2 = 0$ and by declaring a file insertion rejected upon the first insert failure (i.e., no re-salting). The purpose of this experiment is to demonstrate the need for explicit storage load balancing in PAST.

The entire web log trace was played against the PAST system. With no replica and file diversion, 51.1% of the file insertions failed and the global storage utilization of the PAST system at the end of the trace was only 60.8%. This clearly demonstrates the need for storage management in a system like PAST.

In Table 2 shows the results of the same experiment with file and replica diversion enabled, $t_1 = 0.1$, $t_2 = 0.05$, for the various distributions of storage node sizes,

[3]National Laboratory for Applied Network Research, ftp://ircache.nlanr.net/Traces. National Science Foundation (grants NCR-9616602 and NCR-9521745).

and for two settings of the leaf set size l, 16 and 32. The table shows the percentage of successful inserts. The "File diversion" column shows the percentage of successful inserts that involved file diversion (possibly multiple times), and "Replica diversion" shows the fraction of stored replicas that were diverted. "Utilization" shows the global storage utilization of the PAST system at the end of the trace.

Dist. Name	Succeeded	Failed	File diversion	Replica diversion	Utilization
$l = 16$					
d_1	97.6%	2.4%	8.4%	14.8%	94.9%
d_2	97.8%	2.2%	8.0%	13.7%	94.8%
d_3	96.9%	3.1%	8.2%	17.7%	94.0%
d_4	94.5%	5.5%	10.2%	22.2%	94.1%
$l = 32$					
d_1	99.3%	0.7%	3.5%	16.1%	98.2%
d_2	99.4%	0.6%	3.3%	15.0%	98.1%
d_3	99.4%	0.6%	3.1%	18.5%	98.1%
d_4	97.9%	2.1%	4.1%	23.3%	99.3%

Table 2: Effects of varying the storage distribution and leaf set size, when $t_1 = 0.1$ and $t_2 = 0.05$.

The results in Table 2 show that the storage management in PAST is highly effective. Compared to the results with no replica or file diversion, the utilization has risen from 60.8% to $> 94\%$ and $> 98\%$ with $l = 16$ and $l = 32$, respectively. Furthermore, the distribution of node storage sizes has only a minor impact on the performance of PAST, for the set of distributions used in this experiment. As the number of small nodes increases in d_3 and d_4, the number of replica diversions and, to a lesser degree, the number of file diversions increases, which is expected.

There is a noticeable increase in the performance when the leaf set size is increased from 16 to 32. This can be explained because the more PAST nodes in a leaf set, the larger is the scope for local load balancing. Increasing the leaf set size beyond 32 yields no further increase in performance, but does increase the cost of PAST node arrival and departure. Therefore, for the remainder of the experiments a leaf set size (l) of 32 is used.

The next set of experiments examines the sensitivity of our results to the setting of the parameters t_1 and t_2, which control replica and file diversion. In the first of these experiments, the value of t_1 was varied between 0.05 and 0.5 while keeping t_2 constant at 0.05 and using d_1 as the node storage size distribution. Table 3 shows the results.

Figure 2 shows the cumulative failure ratio versus storage utilization for the same experiment. The cumulative failure ratio is defined as the ratio of all failed file insertions over all file insertions that occurred up to the point where the given storage utilization was reached. This

t_1	Succeeded	Failed	File redirect	Replica redirect	Utilization
5	99.30%	0.27%	2.17%	12.86%	97.4%
10	99.33%	0.66%	3.47%	16.10%	98.2%
20	96.57%	3.42%	4.41%	18.13%	99.4%
50	88.02%	11.98%	4.43%	18.80%	99.7%

Table 3: Insertion statistics and utilization of PAST as t_1 is varied and $t_2 = 0.05$.

shows, in conjunction with Table 3, that as t_1 is increased, less files are successfully inserted, but higher storage utilization is achieved. This can be explained by considering that, in general, the lower the value of t_1 the less likely a large file can be stored on a particular PAST node. Many small files can be stored in the place of one large file, therefore, the number of files stored increases, but the utilization drops because large files are being rejected at low utilization levels. Therefore, when the storage utilization is low a higher rate of insertion failure is observed for smaller values of t_1.

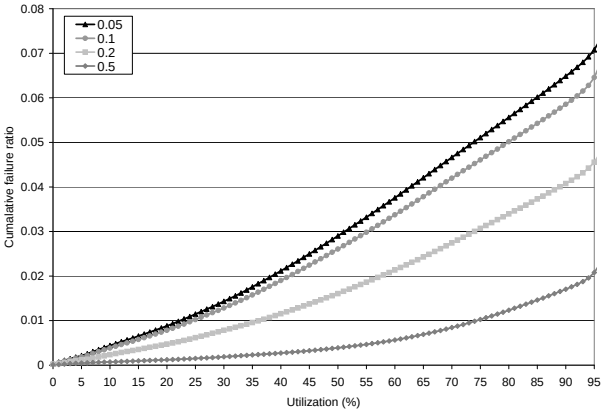

Figure 2: Cumulative failure ratio versus storage utilization achieved by varying the t_1 parameter and $t_2 = 0.05$.

Table 4 shows the effect of varying the t_2 parameter between 0.1 and 0.005, when $t_1 = 0.1$ and storage size distribution d_1 is used. Figure 3 shows the cumulative failure ratio versus storage utilization for the same experiments. As with the experiments varying t_1, the larger the value of t_2 the better the storage utilization, but the fewer insertions complete successfully, for the same reasons.

t_2	Succeeded	Failed	File redirect	Replica redirect	Utilization
10	93.72%	6.28%	5.07%	13.81%	99.8%
5	99.33%	0.66%	3.47%	16.10%	98.2%
1	99.76%	0.24%	0.53%	15.20%	93.1%
0.5	99.57%	0.43%	0.53%	14.72%	90.5%

Table 4: Insertion statistics and utilization of PAST as t_2 is varied and $t_1 = 0.1$.

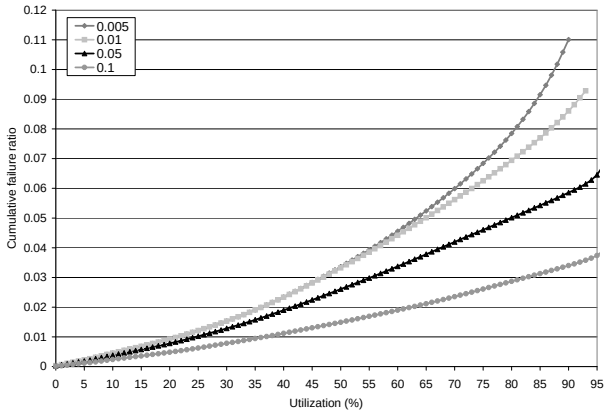

Figure 3: Cumulative failure ratio versus storage utilization achieved by varying the t_2 parameter and $t_1 = 0.1$.

Based on these experiments, we conclude that $t_1 = 0.1$ and $t_2 = 0.05$ provide a good balance between maximal storage utilization and low insertion failure rate of files at low storage utilization.

The next set of results explore in more detail at what utilization levels the file diversion and replica diversion begin to impact on PAST's performance. Figure 4 shows the percentage of inserted files that are diverted once, twice or three times, and the cumulative failure ratio versus storage utilization. The results show that redirects are negligible as long as the storage utilization is below 83%. A maximum of four file redirection attempts is made before an insertion is considered failed.

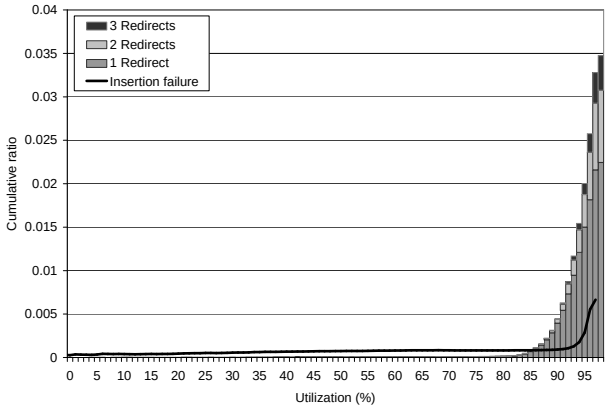

Figure 4: Ratio of file diversions and cumulative insertion failures versus storage utilization, $t_1 = 0.1$ and $t_2 = 0.05$.

Figure 5 shows the ratio of replicas that are diverted over the total number of replicas stored in PAST, versus storage utilization. As can be seen, the number of diverted replicas remains small even at high utilization; at 80% utilization less than 10% of the replicas stored in PAST are diverted replicas. These last two sets of results show that the overhead imposed by replica and file diversion is moderate as long as the utilization is less than about 95%. Even at higher utilization the overhead remains acceptable.

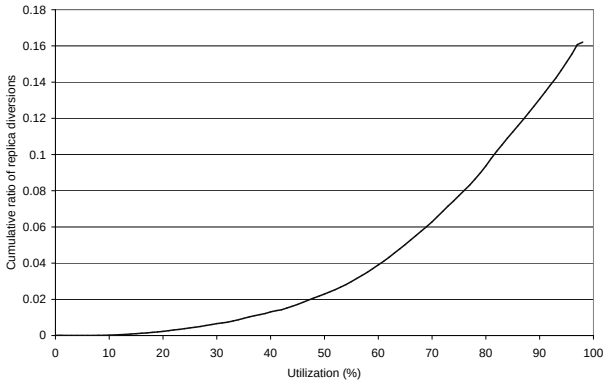

Figure 5: Cumulative ratio of replica diversions versus storage utilization, when $t_1 = 0.1$ and $t_2 = 0.05$.

The next result shows the size distribution of the files that could not be inserted into PAST, as a function of utilization. Figure 6 shows a scatter plot of insertion failures by file size (left axis) versus utilization level at which the failure occurred. Also shown is the fraction of failed insertions versus utilization (right axis). In the graph, files larger than the lower bound of the storage capacity distribution are not shown; the number of files in the NLANR trace that are greater than the upper storage capacity bound is 6, greater than the mean storage capacity is 20, and greater than the lower storage capacity bound is 964. The number of files greater than the lower storage capacity bound that were successfully inserted was 9. None of the files larger than the average storage capacity were successfully inserted.

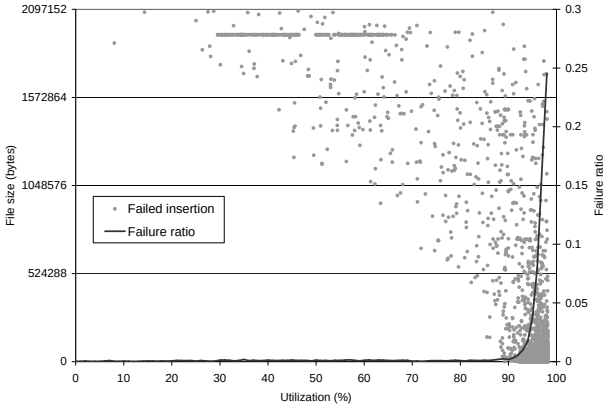

Figure 6: File insertion failures versus storage utilization for the NLANR trace, when $t_1 = 0.1$, $t_2 = 0.05$.

Figure 6 shows that as the storage utilization increases, the smaller the files that are failing to be inserted. How-

ever, the utilization has to reach 90.5% before a file of average size (10,517 bytes) is rejected for the first time. Up until just over 80% utilization no files smaller that 0.5 MByte (e.g., 25% of the minimal node storage capacity) is rejected. Moreover, the total rate of failed insertions is extremely small at a utilization below 90% and even at 95% utilization that total rate of failures is below 0.05, reaching 0.25 at 98%.

Having shown the properties of PAST using the NLANR traces, we now consider results using the file system workload. The total size of all the files in that workload is significantly larger than in the NLANR web proxy trace. The same number of PAST nodes (2250) is used in the experiments, therefore the storage capacity contributed by each node has to be increased. For this experiment, we used d_1 to generate the storage capacities, but increased the storage capacity of each node by a factor of 10. The resulting lower/upper bound on storage capacity is 20 Mbytes and 510 Mbytes, respectively, whilst the mean is 270 Mbytes. The total storage capacity of the 2250 nodes is 596 GBytes.

Figure 7 shows results of the same experiment as Figure 6, but using the filesystem workload. As before, files larger than the smallest storage capacity are not shown; the number of files in the filesystem load that are greater than the upper storage capacity bound is 3, greater than the mean storage capacity is 11, and greater than the lower storage capacity bound is 679. The number of files greater than the smallest storage capacity that were successfully inserted was 23, and none of the files larger than the mean storage capacity were inserted successfully.

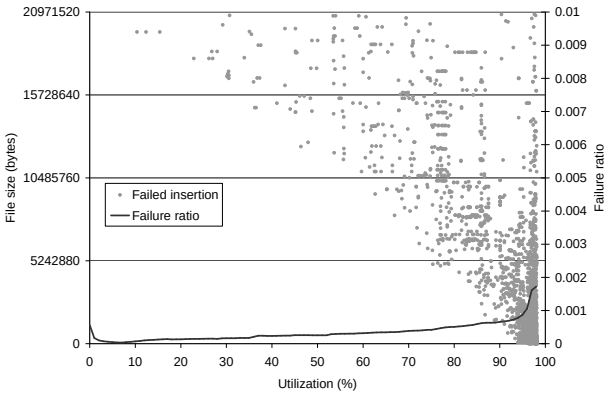

Figure 7: File insertion failures versus storage utilization for the filesystem workload, when $t_1 = 0.1$, $t_2 = 0.05$.

5.2 Caching

The results presented in this section demonstrate the impact of caching in PAST. Our experiment uses the NLANR trace. The trace contains 775 unique clients, which are mapped onto PAST nodes such that a request from a client in the trace is issued from the corresponding PAST node. The mapping is achieved as follows. There are eight individual web proxy traces which are combined, preserving temporal ordering to create the single trace used in the experiments. These eight traces come from top-level proxy servers distributed geographically across the USA. When a new client identifier is found in a trace, a new node is assigned to it in such a way to ensure that requests from the same trace are issued from PAST nodes that are close to each other in our simulated network.

The first time a URL is seen in the trace, the referenced file is inserted into PAST; subsequent occurrences of the URL cause a lookup to be performed. Both the insertion and lookup are performed from the PAST node that matches the client identifier for the operation in the trace. Files are cached at PAST nodes during successful insertions and during successful lookups, on all the nodes through which the request is routed. The c parameter is set to 1. As before, the experiment uses 2250 PAST nodes with the d_1 storage capacity distribution, and $t_1 = 0.1$ and $t_2 = 0.05$.

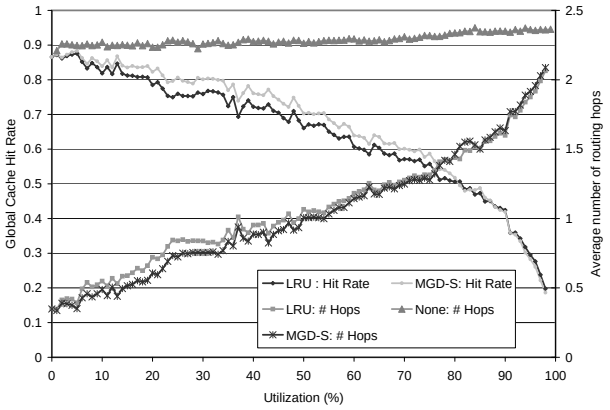

Figure 8: Global cache hit ratio and average number of message hops versus utilization using Least-Recently-Used (LRU), Modified GreedyDual-Size (MGD-S), and no caching, with $t_1 = 0.1$ and $t_2 = 0.05$.

Figure 8 shows both the number of routing hops required to perform a successful lookup and the global cache hit ratio versus utilization. The modified GreedyDual-Size (MGD-S) algorithm described in Section 4, and a Least-Recently-Used (LRU) algorithm were compared for the caching.

When the caching is disabled, the number of routing hops on average required is constant to about 70% utilization and then begins to rise slightly. This is due to replica diversion occurring; therefore, on a small percentage of the lookups a diverted replica is retrieved, adding an extra routing hop. The global cache hit rate for both the

LRU and the MGD-S algorithms decrease as the storage utilization increases. Due to the Zipf-like distribution of web requests [12] it is likely that a small number of files are being requested very often. Therefore, when the system has low utilization, these files are likely to be widely cached. As the storage utilization increases, and the number of files increases, the caches begin to replace some files. This leads to the global cache hit rate dropping.

The average number of routing hops for both LRU and MGD-S indicate the performance benefits of caching, in terms of client latency and network traffic. At low storage utilization, clearly the files are being cached in the network close to where they are requested. As the global cache hit ratio lowers with increasing storage utilization, the average number of routing hops increases. However, even at a storage utilization of 99% the average number of hops is below the result with no caching. This is likely because the file sizes in the proxy trace have a median value of only 1,312 bytes, hence even at high storage utilization there is capacity to cache these small files. It should be noted that in terms of global cache hit ratio and average number of routing hops the MGD-S performs better than the LRU algorithm.

6 Related Work

There are currently many peer-to-peer systems under development. Among the most prominent are file sharing facilities, such as Gnutella [13], Freenet [1], and Napster [3]. These systems are intended the large-scale sharing of music files; persistence and reliable content location are not required in this environment.

PAST instead is a large-scale storage utility that aims at combining scalability and self-configuration with strong persistence. In this regard, it is more closely related with projects like OceanStore [4], FarSite [2], SFS [14, 6], FreeHaven [5], Publius [7] and Eternity [8].

Like PAST, OceanStore provides a global, persistent storage utility on top of an untrusted, unreliable infrastructure. However, PAST focuses on providing a simple, lean storage abstraction for persistent, immutable files with the intention that more sophisticated storage semantics be build on top of PAST if needed. OceanStore provides a more general storage abstraction that supports updates and guarantees serializability in the presence of widely replicated and nomadic data.

FarSite and SFS have more traditional filesystem semantics, while PAST is more targeted towards global, archival storage. Farsite uses a distributed directory service to locate content, which is very different from PAST's Pastry scheme, which integrates content location and routing. FreeHaven, Publius and Eternity are more focused on providing anonymity and survivability of data in the presence of a variety of threats.

Pastry, PAST's routing scheme, bears some similarity to the work by Plaxton et al. [15, 16]. The general approach of routing using prefix matching on the fileId is used in both systems, which can be seen as a generalization of hypercube routing. However, in the Plaxton scheme there is a special node associated with each file, which forms a single point of failure. Also, Plaxton does not handle automatic node integration and failure recovery.

Oceanstore uses a two phase approach to content location and routing. The first stage is probabilistic, using a generalization of Bloom filters. If that stage fails to find an object, then a location and routing scheme called Tapestry is used [17]. Tapestry is based on Plaxton et al. but extends that earlier work in several dimensions. Like Pastry, Tapestry replicates objects for fault resilience and availability and supports dynamic node addition and recovery from node failures. However, Pastry and Tapestry differ in the approach they take for replicating files and in the way they achieve locality.

More loosely related is work on overlay networks [18], ad hoc network routing [19, 20], naming [21, 22, 23, 24, 25, 26] and Web content replication [27, 28, 29].

7 Conclusion

We presented the design and evaluation of PAST, an Internet based global peer-to-peer storage utility, with a focus on PAST's storage management and caching.

Storage nodes and files in PAST are each assigned uniformly distributed identifiers, and replicas of a files are stored at the k nodes whose nodeIds are numerically closest to the file's fileId. Our results show that the storage load balance provided by this statistical assignment is insufficient to achieve high global storage utilization, given typical file size distributions and non-uniform storage node capacities.

We present a storage management scheme that allows the PAST system to achieve high utilization while rejecting few file insert requests. The scheme relies only on local coordination among the nodes in a leaf set, and imposes little overhead. Detailed experimental results show that the scheme allow PAST to achieve global storage utilization in excess of 98%. Moreover, the rate of failed file insertions remains below 5% at 95% storage utilization and failed insertions are heavily biased towards large files.

Furthermore, we describe and evaluate the caching in PAST, which allows any node to retain an additional copy of a file. We show that caching is effective in achieving load balancing, and that it reduces fetch distance and network traffic.

References

[1] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS'2000*, pages 34–43, 2000.

[3] Napster. http://www.napster.com/.

[4] John Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, November 2000.

[5] David Molnar Roger Dingledine, Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[6] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[7] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[8] R.J. Anderson. The Eternity service. In *Proc. PRAGOCRYPT'96*, pages 242–252. CTU Publishing House, 1996. Prague, Czech Republic.

[9] Author Anonymous. Pastry: Scalable, distributed content location and routing for large-scale peer-to-peer systems, 2001. Submitted for publication.

[10] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience*, 27(9):995–1012, September 1997.

[11] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, December 1997.

[12] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. To appear.

[13] The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[14] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 311–320, June 1997. Newport, Rhode Island, USA.

[16] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[17] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing, 2001. Submitted for publication.

[18] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI 2000*, October 2000.

[19] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, August 2000.

[20] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leask. Piconet - embedded mobile networking. *IEEE Personal Communications*, 4(5):8–15, October 1997.

[21] Butler Lampson. Designing a global name service. In *Proceedings of Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, August 1986.

[22] David R. Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[23] Mic Bowman, Larry L. Peterson, and Andrey Yeatts. Univers: An attribute-based name server. *Software—Practice and Experience*, 20(4):403–424, April 1990.

[24] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, Kiawah Island, SC, December 1999.

[25] J. Reynolds. RFC 1309: Technical overview of directory services using the X.500 protocol, March 1992.

[26] Mark A. Sheldon, Andrzej Duda, Ron Weiss, and David K. Gifford. Discover: A resource discovery system based on content routing. In *Proceedings of the 3rd International World Wide Web Conference*, 1995.

[27] Yair Amir, Alec Peterson, and David Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proceedings of the 12th International Symposium on Distributed Computing*, Andros, Greece, September 1998.

[28] Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proceedings of the 4th Web Caching Workshop*, San Diego, CA, March 1999.

[29] Jussi Kangasharju and Keith W. Ross. A replicated architecture for the domain name system. In *Proceedings of the IEEE Infocom 2000*, Tel Aviv, Israel, March 2000.