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

Antony Rowstron
Microsoft Research
St. George House, 1 Guildhall Street
Cambridge, CB2 3NH, United Kingdom.
antr@microsoft.com

Peter Druschel
Rice University
6100 Main Street, MS 132
Houston, TX 77005-1892, USA.
druschel@cs.rice.edu

ABSTRACT

This paper presents and evaluates the storage management and caching in PAST, a large-scale peer-to-peer persistent storage utility. PAST is based on a self-organizing, 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; likewise, 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 such as Napster, Gnutella and FreeNet [1, 2, 13]. While much of the attention has been focused on the copyright issues raised by these particular applications, peer-to-peer systems have many interesting technical aspects like decentralized control, self-organization, adaptation and scalability. 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 [1, 2, 8, 13, 15, 20]. We are developing PAST, an Internet-based, peer-to-peer global storage utility, which aims to provide strong persistence, high availability, scalability and security.

The PAST system is composed of nodes connected to the Internet, where each node is capable of initiating and routing client requests to insert or retrieve files. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing overlay 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, administration, network connectivity, rule of law, etc.

A storage utility like PAST is attractive for several reasons. First, it exploits the multitude and diversity (in geography, ownership, administration, jurisdiction, etc.) of nodes in the Internet to achieve strong persistence and high availability. This obviates the need for physical transport of storage media to protect backup and archival data; likewise, it obviates the need for explicit mirroring to ensure high availability and throughput for shared data. A global storage utility also facilitates the sharing of storage and bandwidth, thus permitting a group of nodes to jointly store or publish content that would exceed the capacity or bandwidth of any individual node.

While PAST offers persistent storage services, its semantics differ from that of a conventional filesystem. Files stored in PAST are associated with a quasi-unique *fileId* that is generated at the time of the file's insertion into PAST. Therefore, files stored in PAST are *immutable* since a file cannot be inserted multiple times with the same fileId. Files can be shared at the owner's discretion by distributing the fileId (potentially anonymously) and, if necessary, a decryption key.

An efficient routing scheme called *Pastry* [27] ensures that client requests are reliably routed to the appropriate nodes. Client requests to *retrieve* a file are routed, with high probability, to a node that is "close in the network" to the client that issued the request[1], 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 logarithmic in the total number of PAST nodes in the system under normal operation.

[1]Network proximity is based on a scalar metric such as the number of IP routing hops, bandwidth, geographic distance, etc.

SOSP-18 11/2001 Canada

To retrieve a file in PAST, a client must know its fileId and, if necessary, its decryption key. PAST does not provide facilities for searching, directory lookup, or key distribution. Layering such facilities on top of Pastry, the same peer-to-peer substrate that PAST is based on, is the subject of current research. Finally, PAST is intended as an archival storage and content distribution utility and not as a general-purpose filesystem. It is assumed that users interact primarily with a conventional filesystem, which acts as a local cache for files stored in PAST.

In this paper, we focus on the storage management and caching in PAST. In Section 2, an overview of the PAST architecture is given and we 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

Any host connected to the Internet can act as a PAST node by installing the appropriate software. The collection of PAST nodes forms an overlay network in the Internet. 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.

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

- `fileId = Insert(name, owner-credentials, k, file)` 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 hash (SHA-1) of the file's name, the owner's public key, and a randomly chosen salt. This choice ensures (with very high probability) that fileIds are unique. Rare fileId collisions are detected and lead to the rejection of the later inserted file.
- `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 (in terms of the proximity metric), among the nodes that store the file.
- `Reclaim(fileId, owner-credentials)` 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. Unlike a delete operation, reclaim does not guarantee that the file is no longer available after it was reclaimed. These weaker semantics avoid complex agreement protocols among the nodes storing the file.

Each PAST node is assigned a 128-bit node identifier, called a nodeId. The nodeId indicates a node's position in a circular namespace, which ranges from 0 to $2^{128} - 1$. The nodeId assignment is quasi-random (e.g., SHA-1 hash of the node's public key) and cannot be biased by a malicious node operator. This process ensures that there is no correlation between the value of the nodeId and the node's geographic location, network connectivity, ownership, or jurisdiction. 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 or threats.

During an insert operation, PAST stores the file on the k PAST nodes whose nodeIds are numerically closest to the 128 most significant bits (msb) of the file's fileId. This invariant is maintained over the lifetime of a file, despite the arrival, failure and recovery of PAST 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. This property follows from the quasi-random assignment of nodeIds and fileIds; it ensures 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; they are described in Section 3.

The number k is chosen to meet the availability needs of a file, relative to 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 is layered on top of *Pastry*, a peer-to-peer request routing and content location scheme. Pastry is fully described and evaluated in [27]. To make this paper self-contained, we give a brief overview of Pastry.

2.1 Pastry

Pastry is a peer-to-peer routing substrate that is efficient, scalable, fault resilient and self-organizing. Given a fileId, Pastry routes an associated message towards the node whose nodeId is numerically closest to the 128 msbs of the fileId, among all live nodes. Given the PAST 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).

Assuming a PAST network consisting of N nodes, Pastry can route to the numerically closest node for a given fileId in less than $\lceil log_{2^b} N \rceil$ steps under normal operation (b is a configuration parameter with typical value 4). Despite 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 can be restored by exchanging $O(log_{2^b} N)$ messages among the affected nodes. In the following, we briefly sketch 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 $\lceil log_{2^b} N \rceil$ 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. Each entry in the routing table points to one of potentially many nodes whose nodeId have the appropriate prefix; in practice, a node is chosen that is close to the present node, according to the proximity metric. If no node is known with a suitable nodeId, then the routing table entry is left empty. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, only $\lceil log_{2^b} N \rceil$ levels are populated in the routing table.

In addition to the routing table, each node maintains IP addresses for the nodes in its *leaf set* and its *neighborhood set*. The leaf set is the set of nodes with the $l/2$ numerically closest larger nodeIds, and the $l/2$ nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. The neighborhood set is a set of l nodes that are near the present node, according to the proximity metric. It is not used in routing, but is useful during node addition/recovery. Figure 1 depicts the state of a PAST node with the nodeId 10233102 (base 4), in a hypothetical system that uses 16 bit nodeIds and values of $b = 2$ and $l = 8$.

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 1: State of a hypothetical Pastry node with nodeId 10233102, $b = 2$, and $l = 8$. All numbers are in base 4. The top row of the routing table represents level zero. The shaded cell at each level of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102* **-** *next digit* **-** *rest of nodeId*. **The associated IP addresses are not shown.**

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 is known, 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. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be live.

Locality Next, we briefly discuss Pastry's properties with respect to the network proximity metric. Recall that the entries in the node routing tables are chosen to refer to a nearby node, in terms of the proximity metric, with the appropriate nodeId prefix. As a result, in each step a message is routed to a "nearby" node with a longer prefix match (by one digit). This local heuristic obviously cannot achieve globally shortest routes, but simulations have shown that the average distance traveled by a message, in terms of the proximity metric, is only 50% higher than the corresponding "distance" of the source and destination in the underlying network [27].

Moreover, since Pastry repeatedly takes a locally "short" routing 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 of a file, Pastry is able to find the "nearest" copy in 76% of all lookups and it finds one of the two "nearest" copies in 92% of all lookups [27].

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 [27].

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 with the destination set to X. This message is routed to the existing node Z with nodeId numerically closest to X[2]. 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 its state and notify all 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 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 [27].

Pastry, as described so far, is deterministic and thus vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they are likely to take the same route. To overcome this problem, the routing is actually randomized. To avoid routing loops, a message must always be forwarded to a node that shares at least as long a prefix with, but is numerically closer to the destination node in the namespace than the current node. The choice among multiple such nodes is random. In practice, the probabil-

[2] In the exceedingly unlikely event that X and Z are equal, the new node must obtain a new nodeId.

ity distribution is heavily biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the client may have to issue several requests, until a route is chosen that avoids the bad node.

A full description and evaluation of Pastry can be found in [27]. In principle, it should be possible to layer PAST on top of one of the other peer-to-peer routing schemes described in the literature, such as Tapestry [31], Chord [30] or CAN [25]. However, some of PAST's properties with respect to network locality and fault resilience may change in this case, depending on the properties of the underlying routing scheme.

2.2 PAST operations

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

In response to an *insert request*, a fileId is computed as the SHA-1 hashcode of the file's textual name, the client's public key, and a random salt. The required storage (file size times k) is debited against the client's storage quota, and a *file certificate* is issued and signed with the owner's private key. The certificate contains the fileId, a SHA-1 hash of the file's content, the replication factor k, the salt, a creation date and other optional file metadata.

The file certificate and the associated file are then routed via Pastry, using the fileId as the destination. When the message reaches the first among the k nodes closest to the fileId, that node verifies the file certificate, recomputes the content hashcode and compares it with the content hashcode in the file 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 replica storing nodes attach a *store receipt*. The client verifies the store receipts to confirm that the requested number of copies have been created. If something goes wrong at any point during the insertion process, such as an illegitimate file 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 sends an appropriate request message, using the requested fileId as the destination. As soon as the request message reaches a node that stores the file, that node responds with the content and the stored file certificate; the request message is not routed 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 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 client's node issues a *reclaim certificate*, which allows the replica storing nodes to verify that the file's legitimate owner is requesting the operation. The storing nodes each issue and return a *reclaim receipt*, which the client node verifies for a credit against the user's storage quota. More detail about quota management can be found in [16].

2.3 Security

While the details of security in PAST are beyond the scope of this paper, we give here a brief overview. More detail can be found in [16] and in a forthcoming paper.

Each PAST node and each user of the system hold a smartcard (read-only clients don't need a card). A private/public key pair is associated with each card. Each smartcard's public key is signed with the smartcard issuer's private key for certification purposes. The smartcards generate and verify the various certificates and they maintain storage quotas. It is possible to operate PAST without smartcards; however, providing comparable security and a quota system without smartcards complicates the system [16].

The following assumptions underly PAST's security model: (1) It is computationally infeasible to break the public-key cryptosystem and the cryptographic hash function used in PAST; (2) while clients, node operators and node software are not trusted and attackers may control the behavior of individual PAST nodes, it is assumed that most nodes in the overlay network are well-behaved; and, (3) an attacker cannot control the behavior of the smartcards.

The smartcards ensure the integrity of nodeId and fileId assignments, thus preventing an attacker from controlling adjacent nodes in the nodeId space, or directing file insertions to a specific portion of the fileId space. Store receipts prevent a malicious node from causing the system to create fewer than k diverse replicas of a file without the client noticing it. The file certificates allow storage nodes and clients to verify the integrity and authenticity of stored content. File and reclaim certificates help enforce client storage quotas. If desired, a client can ensure file privacy by encrypting the content before inserting the file into PAST.

The Pastry routing scheme can be randomized, thus preventing a malicious node along the path from repeatedly intercepting a request message and causing a denial of service. All routing table entries (i.e. nodeId to IP address mappings) are signed by the associated node and can be verified by other nodes. Therefore, a malicious node may at worst suppress valid entries, but it cannot forge entries. Also, routing information in Pastry is inherently redundant and not globally disseminated. Owing to all these factors, Pastry is highly resilient to malicious nodes, and limits their impact to a degradation in routing performance as long as most nodes are well-behaved. In the worst case, widespread corruption of nodes could cause routing failures and thus denial of service.

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. While it is difficult to predict if systems such as PAST will be typically operated at high levels of storage utilization, it is our contention that any highly efficient and reliable system must remain robust in the event of extreme operating conditions.

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 the fileId to alternatively store the file, if it is in the leaf set of one of 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 node'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 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 may have 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 of attached disks) is irrelevant, although considerations of cost and fault resilience normally favor clusters of PCs. 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 the 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 exist 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.

If node A fails then node C still maintains a pointer to the replica stored on B, maintaining the invariant that the k closest nodes maintain either a replica or a reference to a distinct diverted replica. If node C fails then node A installs a reference on the now $k+1$th closest node.

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 insertion 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 below average to a node whose free space is significantly above 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_{pri} while the latter use t_{div}, where $t_{pri} > t_{div}$.

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 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 retries the insert operation.

A client node then repeats this process for up to three times. If, after four attempts 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 retry 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 constitute 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.

Given the current ratio of disk storage versus wide-area Internet bandwidth, it is time-consuming and inefficient for a node to request replicas of all files for which it has just become one of the k numerically closest nodes. 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). The affected files can then be gradually migrated to the joining node as part of a background operation.

When a PAST network is growing, node additions may create the situation where a node that holds a diverted replica and the node that refers to that replica are no longer part of the same leaf set. Because such nodes are not automatically notified of each other's failure, they must explicitly exchange keep-alive messages to maintain the invariants. To minimize the associated overhead, affected replicas are gradually migrated to a node within the referring node's leaf set whenever possible.

Consider the case 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 nodes' 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 does not decrease. If total disk storage were to decrease due to node and disk failures that were not eventually balanced by node and disk additions, then the system would eventually exhaust all of its storage. Beyond a certain point, the system would 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 systems like PAST that are not centrally managed. Any solution must provide strong incentives for users to balance their resource consumption with the resources they contribute to the system. PAST addresses this problem by maintaining storage quotas, 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 [23]. 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 contacting several nodes to retrieve a file. This cost may outweigh the benefits for all but large files; exploring this option is future work. The storage management issues discussed in this paper, however, are largely orthogonal to the choice of file encoding and striping.

4. CACHING

In this section, we describe 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. Note that because PAST is running on an overlay network, fetch distance is measured in terms of Pastry routing hops.

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 than 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 local clusters of clients, it is advantageous to store a copy near each cluster. 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 files. 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 used 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 whose 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 $c(d)$ is set to one, the policy maximizes the cache hit rate. Results presented in Section 5 demonstrate the effectiveness of caching in PAST.

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 emulation environment, through which the instances of the node software communicate.

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.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 6 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.

It was verified that the storage invariants are maintained properly despite random node failures and recoveries. PAST is able to maintain these invariants as long as Pastry is able to maintain proper leaf sets, which is the case unless $\lfloor l/2 \rfloor$ nodes with adjacent nodeIds fail within a recovery period. The remaining 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 authors' 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.

Selecting an appropriate workload to evaluate a system like PAST is difficult, since workload traces for existing peer-to-peer systems are not available and relatively little is known about their characteristics [28]. We chose to use web proxy and filesystems workloads to evaluate storage management and caching in PAST. The file-size distributions

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 MBytes.

in the two workloads are very different and should bracket the range of size distributions likely to be encountered by PAST. For the purposes of evaluating caching, the locality properties in the web proxy log should give a rough idea of the locality one would expect of a PAST workload.

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 measurements and analysis in [8], which considers availability of desktop computers in a corporate network environment. 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.

Table 1 shows the values of m and σ for four distributions used in the first set of experiments. The lower and upper bounds indicate where the tails of the normal distribution were cut. In the case of d_1 and d_2, the lower and upper bound was defined as $m - 2.3\sigma$ and $m + 2.3\sigma$, respectively. For d_3 and d_4, the lower and upper bound was fixed arbitrarily, and a large σ was used. We have also experimented with uniform distributions, and found that the results were not significantly affected.

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. Unless 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 the threshold[4] for primary replica stores to $t_{pri} = 1$, setting the threshold for the diverted replica stores to $t_{div} = 0$ 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

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

[4]Recall that file size/free storage space > threshold t for a file to be stored on a node.

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, $t_{pri} = 0.1$, $t_{div} = 0.05$, for the various distributions of storage node sizes, and for two settings of the leaf set size l, 16 and 32. The table shows the percentage of successful and unsuccessful inserts, "Success" and "Fail" respectively. 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. "Util" shows the global storage utilization of the PAST system at the end of the trace.

Dist. Name	Succeed	Fail	File diversion	Replica diversion	Util.
$l = 16$					
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 $t_{pri} = 0.1$ and $t_{div} = 0.05$.

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 is because a larger leaf set increases the scope for local load balancing. With the storage size distributions used in our experiments, 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_{pri} and t_{div}, which control replica and file diversion. In the first of these experiments, the value of t_{pri} was varied between 0.05 and 0.5 while keeping t_{div} 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 data, in conjunction with Table 3, shows that as t_{pri} is increased, fewer files are successfully inserted, but higher storage utilization is achieved. This can be explained by considering that, in

t_{pri}	Succeed	Fail	File divers.	Replica divers.	Util.
0.5	88.02%	11.98%	4.43%	18.80%	99.7%
0.2	96.57%	3.43%	4.41%	18.13%	99.4%
0.1	99.34%	0.66%	3.47%	16.10%	98.2%
0.05	99.73%	0.27%	2.17%	12.86%	97.4%

Table 3: Insertion statistics and utilization of PAST as t_{pri} is varied and $t_{div} = 0.05$.

general, the lower the value of t_{pri} the less likely it is that a large file can be stored on a particular PAST node. Many small files can be stored in place of one large file; therefore, the number of files stored increases as t_{pri} decreases, 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_{pri}.

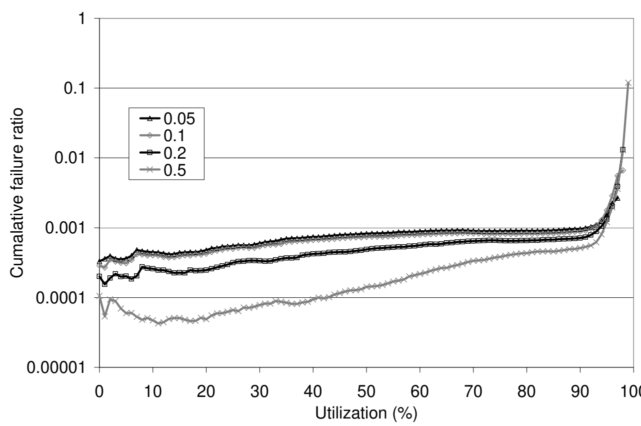

Figure 2: Cumulative failure ratio versus storage utilization achieved by varying the t_{pri} parameter and $t_{div} = 0.05$.

Table 4 shows the effect of varying the t_{div} parameter between 0.1 and 0.005, when $t_{pri} = 0.1$ and storage size distribution d_1 is used. Figure 3 shows the cumulative failure ratio versus storage utilization for the same experiments. As in the experiment varying t_{pri}, as the value of t_{div} is increased the storage utilization improves, but fewer insertions complete successfully, for the same reasons.

t_{div}	Succeed	Fail	File divers.	Replica divers.	Util.
0.1	93.72%	6.28%	5.07%	13.81%	99.8%
0.05	99.33%	0.66%	3.47%	16.10%	98.2%
0.01	99.76%	0.24%	0.53%	15.20%	93.1%
0.005	99.57%	0.43%	0.53%	14.72%	90.5%

Table 4: Insertion statistics and utilization of PAST as t_{div} is varied and $t_{pri} = 0.1$.

We repeated the sensitivity experiments using the filesystem trace and, despite the different file size distribution in that trace, the results were similar. Based on these experiments, we conclude that $t_{pri} = 0.1$ and $t_{div} = 0.05$ provide a good balance between maximal storage utilization and a low file insertion failure rate at low storage utilization.

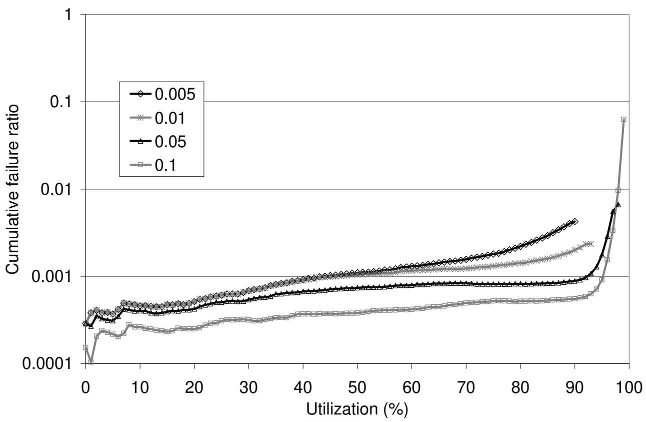

Figure 3: Cumulative failure ratio versus storage utilization achieved by varying the t_{div} parameter and $t_{pri} = 0.1$.

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 file diversions are negligible as long as storage utilization is below 83%. A maximum of three file diversion attempts are made before an insertion is considered failed.

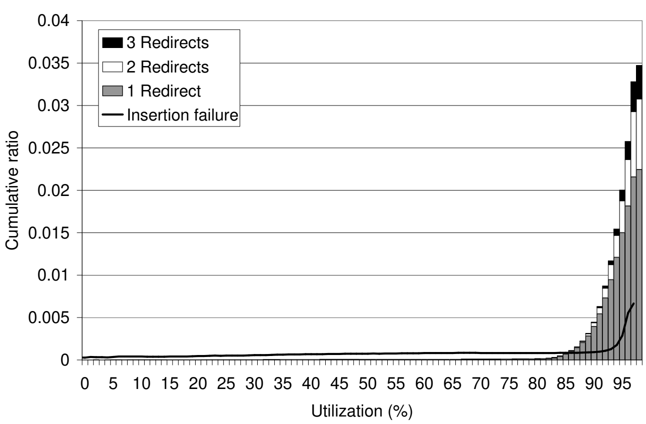

Figure 4: Ratio of file diversions and cumulative insertion failures versus storage utilization, $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Figure 5 shows the ratio of replicas that are diverted to the total 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.

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 vertical axis) versus utilization level at which the failure occurred. Also shown is the fraction of failed insertions ver-

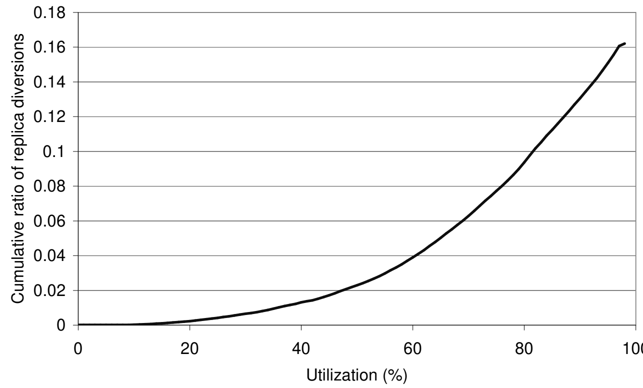

Figure 5: Cumulative ratio of replica diversions versus storage utilization, when $t_{pri} = 0.1$ and $t_{div} = 0.05$.

sus utilization (right vertical axis). In the graph, files larger than the lower bound of the storage-capacity distribution are not shown; in the NLANR trace, 6 files are larger than the upper storage capacity bound, 20 are larger than the mean storage capacity, and 964 are larger than the lower storage capacity bound. The number of files larger than the lower storage capacity bound that were successfully inserted was 9. None of the files larger than the mean storage capacity were successfully inserted.

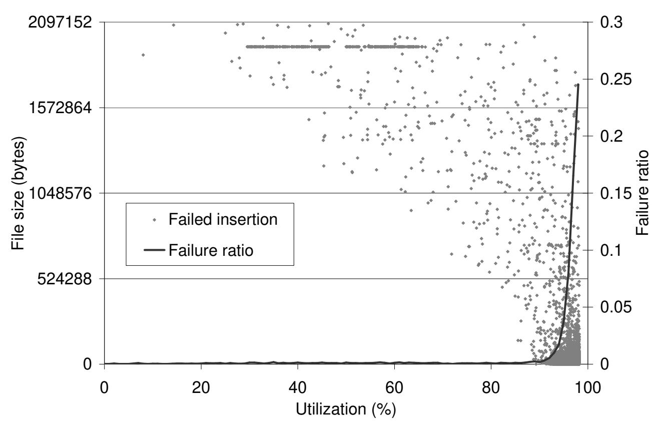

Figure 6: File insertion failures versus storage utilization for the NLANR trace, when $t_{pri} = 0.1$, $t_{div} = 0.05$.

Figure 6 shows that as the storage utilization increases, smaller files fail to be inserted. However, the utilization reaches 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 than 0.5 MBytes (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 the 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 filesystem 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; in the filesystem load, 3 files are larger than the upper storage capacity, 11 are larger than the mean storage capacity, and 679 are larger than the lower storage capacity bound. The number of files larger 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.

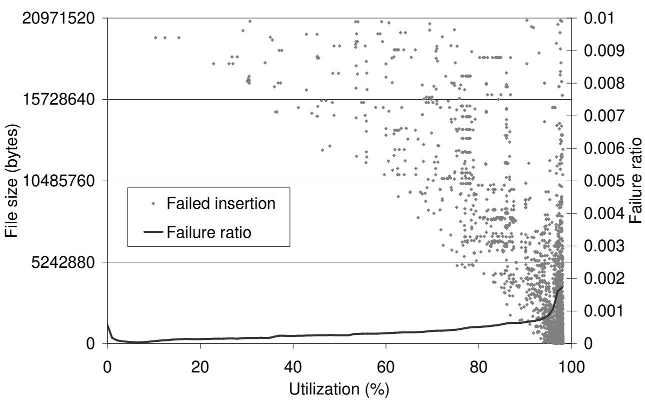

Figure 7: File insertion failures versus storage utilization for the filesystem workload, when $t_{pri} = 0.1$, $t_{div} = 0.05$.

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 experiment. 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 emulated 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, $t_{pri} = 0.1$ and $t_{div} = 0.05$.

Figure 8 shows both the number of routing hops required to perform a successful lookup and the global cache hit ratio versus utilization. The GreedyDual-Size (GD-S) policy described in Section 4 is used. For comparison, we also include results with the Least-Recently-Used (LRU) policy.

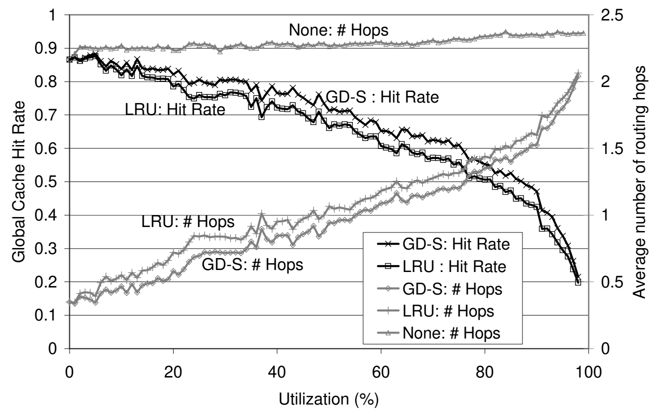

Figure 8: Global cache hit ratio and average number of message hops versus utilization using Least-Recently-Used (LRU), GreedyDual-Size (GD-S), and no caching, with $t_{pri} = 0.1$ and $t_{div} = 0.05$.

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. It should be noted that $\lceil log_{16} 2250 \rceil = 3$. The global cache hit rate for both the LRU and the GD-S algorithms decreases as storage utilization increases. Because of the Zipf-like distribution of web requests [10], 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 GD-S indicates 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. In terms of global cache hit ratio and average number of routing hops, GD-S performs better than LRU.

We have deliberately reported lookup performance in terms of the number of Pastry routing hops, because actual lookup delays strongly depend on per-hop network delays. To give an indication of actual delays cause by PAST itself, retrieving a 1KB file from a node one Pastry hop away on a LAN takes approximately 25ms. This result can likely be improved substantially with appropriate performance tuning in our prototype implementation.

6. RELATED WORK

There are currently several peer-to-peer systems in use, and many more are under development. Among the most prominent are file sharing facilities, such as Gnutella [2] and

Freenet [13]. The Napster [1] music exchange service provided much of the original motivation for peer-to-peer systems, but it is not a pure peer-to-peer system because its database is centralized. All three systems are primarily intended for the large-scale sharing of data files; persistence and reliable content location are not guaranteed or necessary in this environment.

In comparison, PAST aims at combining the scalability and self-organization of systems like FreeNet with the strong persistence and reliability expected of an archival storage system. In this regard, it is more closely related with projects like OceanStore [20], FarSite [8], FreeHaven [15], and Eternity [5]. FreeNet, FreeHaven and Eternity are more focused on providing strong anonymity and anti-censorship.

OceanStore provides a global, transactional, persistent storage service that supports serializable updates on widely replicated and nomadic data. In contrast, PAST provides a simple, lean storage abstraction for persistent, immutable files with the intention that more sophisticated storage semantics (e.g., mutable files) be built on top of PAST if needed.

Unlike PAST, FarSite has traditional filesystem semantics. A distributed directory service is used in FarSite to locate content; this is different from PAST's Pastry scheme, which integrates content location and routing. Currently, there is no published scalability analysis of FarSite. Every node maintains a partial list of the live nodes, from which it chooses nodes that should store replicas. Much of FarSite's design is motivated by a feasibility study that measures a corporate LAN [8]; some of its assumptions may not hold in a wide-area environment.

Pastry, along with Tapestry [31], Chord [30] and CAN [25], represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella.

Pastry and Tapestry bear some similarity to the work by Plaxton et al [24]. The approach of routing based on address prefixes, which can be viewed as a generalization of hypercube routing, is common to all three schemes. 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, i.e., it is not self-organizing. Pastry and Tapestry differ in their approach to achieving network locality and to replicating objects, and Pastry appears to be less complex.

The Chord protocol is closely related to both Pastry and Tapestry, but instead of routing towards nodes that share successively longer address prefixes with the destination, Chord forwards messages based on numerical difference with the destination address. Unlike Pastry and Tapestry, Chord makes no explicit effort to achieve good network locality.

CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. Unlike Pastry, the routing table does not grow with the network size, but the number of routing hops grows faster than $logN$.

CFS [14] is a decentralized, cooperative read-only storage system. Like PAST, it is built on top of a peer-to-peer routing and lookup substrate, in this case Chord. Unlike PAST, it is intended solely as a file sharing medium, and thus provides only weak persistence. CFS storage is block-oriented and a conventional UNIX-like filesystem is layered on top of it. Each block is stored on multiple nodes with adjacent Chord node ids and popular blocks can be cached at additional nodes, similar to the way entire files are stored in PAST. Compared to PAST, this increases file retrieval overhead, as each file data and metadata block must be located using a separate Chord lookup. On the other hand, CFS permits parallel block retrievals, which benefits large files.

CFS's design assumes an abundance of free disk space. Combined with its block orientation and weak persistence, this simplifies its storage management, when compared to a system like PAST. To accomodate nodes with more than the minimal storage size, CFS relies on hosting multiple logical nodes per physical nodes, each with a separate Chord id. PAST uses this technique only for nodes whose storage size exceeds the minimal size by more than two orders of magnitude. For both Chord and Pastry, the overhead of maintaining state for multiple logical nodes increases proportionally.

xFS [6] is a serverless filesystem. While it shares its decentralized architecture with peer-to-peer systems like PAST, it is intended as a general-purpose filesystem serving a single organization within a LAN. As such, its design goals and assumptions with respect to performance, network characteristics, security, and administration are very different from PAST's. More loosely related is work on overlay networks [17], ad hoc network routing [7, 22], naming [3, 9, 12, 21, 26, 29] and Web content replication [4, 18, 19].

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.

Acknowledgments

We would like to thank Miguel Castro, Marvin Theimer, Dan Wallach, the anonymous reviewers and our shepherd Andrew Myers for their useful comments and feedback. Peter Druschel thanks Microsoft Research, Cambridge, UK, and the Massachussetts Institute of Technology for their support during his visits in Fall 2000 and Spring 2001, respectively, and Compaq for donating equipment used in this work.

8. REFERENCES

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

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

[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. SOSP'99*, Kiawah Island, SC, Dec. 1999.

[4] Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proc. 12th Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.

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

[6] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. 15th ACM SOSP*, Copper Mountain, CO, Dec. 1995.

[7] F. Bennett, D. Clarke, J. B. Evans, A. Hopper, A. Jones, and D. Leask. Piconet - embedded mobile networking. *IEEE Personal Communications*, 4(5):8–15, October 1997.

[8] 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*, Santa Clara, CA, 2000.

[9] M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software — Practice and Experience*, 20(4):403–424, Apr. 1990.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE Infocom'99*, New York, NY, Mar. 1999.

[11] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[12] D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Trans. Comput. Syst.*, 7(2):147–183, May 1989.

[13] I. Clarke, O. Sandberg, B. Wiley, and T. 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.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[15] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.

[16] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, Schloss Elmau, Germany, May 2001.

[17] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI 2000*, San Diego, CA, October 2000.

[18] J. Kangasharju, J. W. Roberts, and K. W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proc. 4th Web Caching Workshop*, San Diego, CA, Mar. 1999.

[19] J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proc. IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.

[21] B. Lampson. Designing a global name service. In *Proc. Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, Minaki, Canada, Aug. 1986.

[22] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, Boston, MA, August 2000.

[23] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software — Practice and Experience*, 27(9):995–1012, Sept. 1997.

[24] 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.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

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

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[28] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, University of Washington, July 2001.

[29] M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proc. 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[31] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.