Policy Driven Replication

Abstract

The increasingly commodity nature of storage and our insatiable tendency to produce, store, and use large amounts of data exacerbates the problem of ensuring data survivability. The advent of large robust networks has gained the idea of replicating data on remote hosts wide-spread acceptance. Unfortunately, the growth of network bandwidth is far outstripped by both the growth of both storage capacity [17] and our ability to fill it. Thus, most replication systems, which traditionally replicate data blindly, fail under the onslaught of this lopsided mismatch.

We propose a Policy Driven Replication (PDR) system that prioritizes the replication of data, based on user-defined policies that specify which data is to be protected, from which failures, and to what extent. By prioritizing which data is replicated, our system conserves limited resources and ensures that data which is deemed most important to and by the user is protected from failures that are deemed most likely to occur.

1 Introduction

Digital storage is in the midst of a revolution. Improvements in storage density combined with a proliferation of new digital media are dramatically changing what people store on disk. Gone are the days when the average home computer stored only letters and spreadsheets. A typical PC today stores financial records, tax returns, music libraries, photo albums and much more. Computer storage is moving to the centre of people's lives. As it does, the consequences of a data-destroying failure become increasingly catastrophic. The digital-storage revolution thus requires not just that storage be cheap, but also that it be reliable.

Current reliability techniques, however, typically fail to adequately protect data, are very expensive, or both. Reliable-storage administration, for example, is an order of magnitude more expensive than physical storage itself [17, 22]. A key reason for this high cost is that many existing techniques such as tape and optical-disk backup require too much human intervention to scale to modern disk capacities and reliability standards. RAID [16, 28] and related distributed-disk solutions [7, 11] provide a significant degree of automation, but do not protect data from site failures and are often expensive and complex. High-end commercial file systems [3, 4] protect data from site failure by tightly coupling to an off-site mirror, using a specialized high-bandwidth connection. Some research systems provide similar protection [2, 13, 26] without requiring a specialized connection, but they also tightly couple primary and secondary sites. This tight coupling is a major source of complexity that inhibits scalability, increases cost and makes the system less resilient.

Recent research has examined the use of a collection of peer-to-peer nodes to replicate data and thus protect it from failure [5, 19]. This approach has the potential advantage of low cost, loose coupling and low complexity. Most recent peer-to-peer systems are organized as a distributed hash table [18, 24] that stores a file, or file block, and the node whose ID is closest to the file's. Multiple copies of a file are stored on the i nodes closest to the file's ID. The fact that node ID's are assigned randomly means that replicas are each stored on randomly chosen nodes with presumably independent failure modes. A limitation of this approach, however, is that when a new node is added, many of the files stored on its immediate successor must be copied to the new node.

Farsite [1] is a peer-to-peer system that takes a different approach. Instead of using a distributed hash table to index storage, Farsite designates a subset of nodes to co-operate to store a hierarchical index of the file system. A file's index entry lists the IDs of the nodes that store it, more than one node if the file is replicated. A file's storage nodes are initially chosen at random from the entire system. The system subsequently swaps file-storage nodes to improve average availability: files with replicas on fault-prone nodes swap a replica with other files. One of the advantages of Farsite is that, unlike the distributed hash table approaches, no file-coping is required when adding new nodes.

This paper describes the design and implementation of a peer-to-peer replication system called Policy Driven Replication (PDR). The key novel features of PDR are that it uses user-specified, file-grain policies to direct replication and that it uses a non-random approach to selecting storage nodes.

The motivation for the first feature is the observation that not all data requires the same amount of failure protection and that replication can be expensive, particularly replication to off-site nodes. Our approach allows users to categorize their files based on how much failure protection they require and can afford. Temporary and derivative files (e.g., object files), for example, require no replication. Other, moderately important files may benefit from infrequent replication within a local-area network, with critical data periodically copied to an off-site node.

There are two potential benefits to PDR's non-random approach to selecting storage nodes. First, it provides the system and users with the flexibility to make smart decisions about where to replicate files. This flexibility, for example, allows the system choose two close-by, but failure-independent, nodes for efficient and safe storage of two copies of a file. The other peer-to-peer systems achieve similar safety through random selection, at the cost of performance, particularly in a wide-area network. Another benefit of this flexibility relates to how users choose other nodes willing to store their files. The peer-to-peer approach requires that all participants contribute equally to store a random subset of file replicas. Our approach is less onerous and more flexible. Multiple replication models can coexist. Some might, for example, involve commercial services, others non-profit organizations, still others grass-roots confederations of friends or neighbours, or simply a set of machines used by a single user in different locations (e.g., work and home). Finally, PDR's flexibility also allows users to select the degree of trust they require from nodes that store their files. The random-selection approaches require elaborate mechanisms to provide security under the assumption that nodes are untrusted.

A second argument against random replica-node selection relates to the cost of managing replication in the face of failure. When a node with replicas fails permanently, the system must re-replicate files stored there to a new node(s). The cost of this replication is a function both of the number of files replicated on the failed node and of the number of *nodes* that store other copies of those files. If the storage nodes for each file are chosen randomly, as they are in Farsite for example, re-replication requires global communication. PDR, on the other hand, ensures that replica nodes form cliques that minimize the number of nodes involved in reconstituting a failed replica, thus localizing the impact of node failure. Distributed hash tables have a similar advantage when they replicate files on nodes that are neighbours in the node-ID name space.

2 Design

The PDR system is a collection of independently operating nodes (physical machines). There are no centralized services and all nodes run the same software (Figure 1). A node is either a client, a replica, or both. Client nodes only push data to replica nodes and are not responsible for storing replicated data or ensuring that replication policies are not violated. Replica nodes, as the name suggests, store replicated data for client nodes, and ensure that replication policies are not violated, re-replicating data when policy violations occur.

Each node comprises of two parts: the *replicator* and the *policy oracle*. The *replicator* replicates data, and the *policy oracle* oversees policy creation. These tasks are separated to help PDR manage state. The replicator manages the low-level state and metadata associated with each file. The policy oracle manages the high-level state; in particular, it tracks the effect of changes of the low-level state on the high-level state.

The *replicator* process runs on every node and is responsible for replicating data; it interprets, executes, and enforces replication policies by re-replicating data if a node fails. It is integrated with the local file system and receives upcalls whenever a file or a directory is created or modified. The replicator invokes the local policy oracle when node selection is needed and communicates with replicators on other nodes for replication or recovery purposes.

The *policy oracle* is responsible for the majority of the decision making in PDR. Primarily, the policy oracle is responsible for selecting replica nodes to satisfy both new replication policies and existing policies when a replica node has failed. The policy oracle is also responsible for delegating the work for recovering a failed node. It also communicates with other policy oracles to maintain up-to-date connectivity (network topology) information.

2.1 Replication policies

Replication policies control where and when data is replicated. At the level of the replicator a policy is a list of pairs consisting of a replica node and a *staleness factor*. This bounds the datedness of data for a replica node. A staleness factor of x means that data is replicated to the replica node within x from the time it is modified.

Users specify replication policies by either explicitly specifying the nodes, or by specifying the number of nodes of a particular node class. The latter allows users to specify the desired level of protection more naturally. In both cases the user must specify the staleness factor.

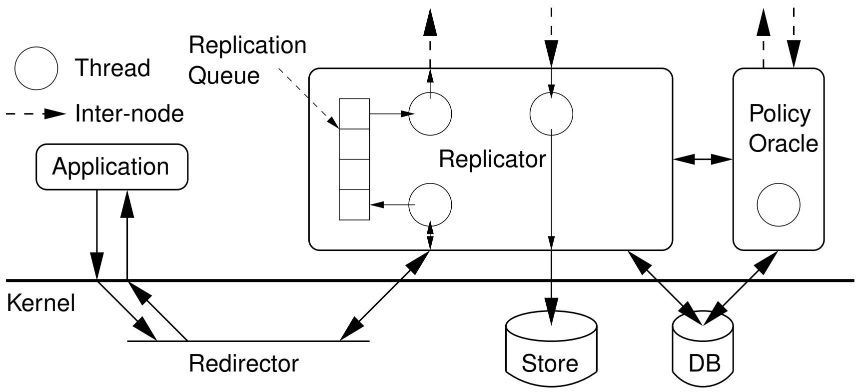

Figure 1: A PDR Node.

Nodes are partitioned into classes based on the node's reliability. The reliability of a node is based on the node's connection, distance from the primary site, physical location, and administrative domain. Using these measures nodes are partitioned into five classes: local workstation, local server room, remote workstations, remote server room, and data centre. In general, the closer the replica node is to the primary site the smaller the replication cost, because the bandwidth is cheaper. Server rooms and data centres are more reliable, but more expensive, because the hardware is specifically designed for reliability and there is explicit administrative support.

2.2 Data structure and metadata

There are five key data structures in PDR: the replication policy, the replication queue, and three maps that comprise the high-level state and are maintained by the policy oracle. These maps aid the policy oracle in selecting new and replacement nodes for replication policies. They implicitly track the topology of the system, and enable the policy oracle to determine the effect of adding a node to a replication policy on the topology.

A replication policy is the main data structure that stores the low-level state. Replication policies are managed by the replicator, and there is one for every file and directory in the file system. In this structure (Table 2a) the important fields are the policy key, the policy version number, and the list of replica nodes. The policy key is a 64-bit MD5 hash of the file name, and it is the identifier for the policy. The policy version number is used to ensure that nodes do not use stale policies; it is incremented each time the policy is modified. Each replica node entry is a pair storing the node and associated staleness factor. Replication policies are stored in the database.

The three maps map policies to nodes, nodes to policies, and nodes to nodes. Replication policies implicitly create the policy-to-node map, which the policy oracle uses to determine the set of nodes associated with a replication policy.

The node-to-policy map is used by the policy oracle determines the set of policies a node is participating in. A node-to-policy record (Table 2b) consists of a list of policy keys and information to compute the cost of maintaining the node and recovering the node if it fails. The maintenance cost is dependent upon the number of policies on the node, and the recovery cost depends on the file system size and the network connection.

The node-to-node map stores the topology of the network. The policy oracle uses the policy-to-node and the node-to-policy maps to create the node to node mappings. A node-to-node record stores the node's reliability characteristics and the list of nodes it is dependent on by virtue of participating in the same replication policy.

The replication queue is part of the mechanism that enables PDR to remove the replication process from the critical path and perform replication asynchronously. It is a persistent priority queue that is used to schedule replications by the replicator; the entries are ordered by the staleness factor, smallest first. This queue is made persistent so that updates are not lost between system restarts, thus enabling PDR to deal with transient client node failures. Replication events comprising of the name of the file to be replicated, the node the file is to be replicated to, and the time at which the replication should occur, are stored in the replication queue.

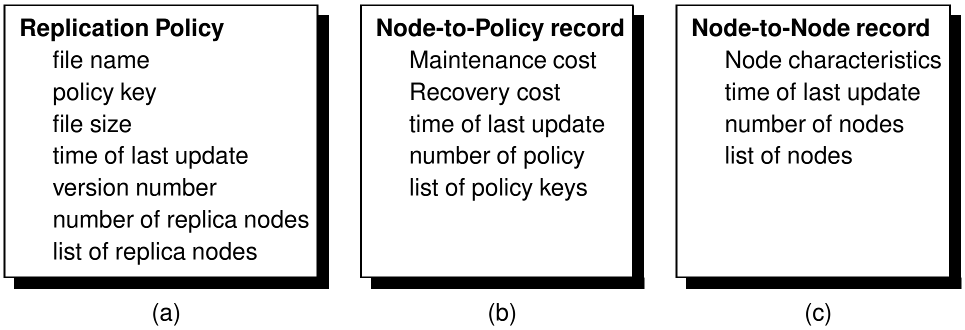

Figure 2: Key data structures.

2.3 Replicator

The design of the replication mechanism was driven by several goals: (1) avoid performing redundant replication, (2) avoid inducing overhead into the local file system, and (3) avoid tight coupling of nodes.

The *replicator* has two primary responsibilities. First, the replicator is responsible for interpreting the low-level replication policies and scheduling replication events. Second, the replicator is responsible for monitoring all nodes that are participating in the replication policies. If a node failure is detected, the replicator initiates recovery procedures.

2.3.1 Replicating data

In general, beneficial redundancy is attained by maintaining extra replicas of data. In PDR, however, because it is adhering to a cost model that relates cost to replication level, performing any extra replication is avoided. When a file or directory is modified, the changes are not replicated immediately. Instead, the replicator schedules replication based on the replication policy.

The scheduling of the replication removes the replication process from the critical path of the file system. The replicator is structured as an event driven system (Figure 1). A scheduling thread, a dispatching thread, and the replication queue are used to schedule and perform replication. The amount of work done by the scheduling thread is small, thus adding a minimal amount of latency to file system calls. The dispatching thread asynchronously performs the replication.

In PDR, progress is never impeded due to a node failure. The system quickly re-configures itself to avoid failed nodes and to maintain replication policy conformance. If a replica node fails a new node is chosen and the data that was on the failed node is re-replicated. Each replication policy has a *leader* node that is responsible for coordinating failure recovery in the event a replica node fails. The leader is implicitly chosen as having the smallest staleness factor because it most likely has the most up-to-date metadata and data for the file.

2.3.2 Monitoring other nodes

The other responsibility of the replicator is to monitor other replica nodes for liveliness. In PDR, policies are treated as invariants, and if they are violated the system attempts to correct the violations. A violation occurs when a node that is participating in one or more replication policies fails. To detect these violations a heartbeat is used.

A node that is participating in a policy monitors all the other nodes in the policy. The rate of the heartbeat messages depends on the staleness factor for that node, which further reduces the number of heartbeat messages. In addition, heartbeat messages are not duplicated for nodes that are participating in multiple policies.

2.4 Policy oracle

In PDR, the policy oracle manages the high-level state. Low-level state, i.e., replication policies, is not significantly affected by node selection. Any node suffices as long as the resulting replication policy conforms to the desired level or protection that was specified by the user. Node selection, however, does impact the high-level state. A bad node selection could create an undesirable topology. Thus, in PDR, the policy oracle is responsible for selecting nodes for policies.

The replicator is the policy oracle's primary client, but the policy oracles also communicate with each other to exchange topology information. The replicator predominantly queries the policy oracle for policy information, or inserts new or updates existing policy information.

3 Normal system operations

There are three types of operations, those performed by a client node, those performed by the replicator, and those that are performed by the policy oracle. Section 3.1 briefly describes how PDR interacts with the file system. The main operations are described in Sections 3.2 through 3.5. Section 3.6 presents the algorithm used by the policy oracle to select nodes for replication policies.

3.1 File system redirection

The file system redirector is the means by which the replicator interacts with the local file system. Although the replicator and the policy oracle are portable to any platform that is Posix compliant, there is only a redirector for Linux. In particular, the Coda [9] redirector is used and is available as a standard Linux kernel module.

All the main file system calls are intercepted by the redirector and propagated to the replicator. PDR is only interested in `creat`, `open`, `close`, `mkdir`, `rmdir`, `remove`, and `rename`.

3.2 Client node operations

The replicator on the client node is responsible for handling file system upcalls, scheduling replication, replicating data, and setting policies.

3.2.1 File system upcalls

The replicator interacts with the redirector to service file system requests. On receiving an upcall the replicator has to create or update metadata, indicate that a file is modified, and or schedule a replication.

For `creat` and the `mkdir` upcalls the replicator instantiates a replication policy for the file or directory. If an `open` upcall is received with write flags for a file, then that file is marked as modified. On receiving a `close` upcall for a modified file the replicator schedules replication for the file. When a `remove`, `rmdir`, or `rename` upcall is received the replicator performs the necessary cleanup and updates the metadata.

3.2.2 Scheduling replication

When the replicator receives a `creat` or an `open` upcall with the intent to modify the file, the file is flagged and a temporary copy of the file is created, so that previously scheduled replication events can proceed instead of waiting for the file to be closed. In this way, no file system requests are ever blocked due to replication. On receiving a `close` upcall the replicator retrieves the replication policy for the file and schedules, if necessary, a replication event.

Scheduling a replication consists of creating a replication event for each replica node listed in the replication policy and inserting it into the replication queue. Multiple events for the same file–node pair are merged.

3.2.3 Replicating data

Replication events are dispatched by a separate thread in the replicator. Dispatching a replication event involves sending the file to the replica nodes. The client sends the file plus a small amount of metadata that reflects the changes to the file's attributes. The leader node monitors the client for the duration of the longest staleness factor in the policy. If the client node is reachable for the entire period then the client must have replicated the data to all replicas listed in the policy. If the client becomes unreachable at some point then the policy has been violated and the leader node assumes the responsibility to replicate the data to the other nodes in the policy. Other failure scenarios are discussed in Section 4.

3.2.4 Setting policies

Files acquire replication policies in three ways. One, the system administrator sets a default replication policy that is automatically inherited when a file or a directory is created. Two, users can explicitly set and change policies for files and directories with a set of file system tools; the policy tool communicates to the replicator, which passes the replication policy changes to the policy oracle. These tools support traditional name globbing and thus setting policies for particular types of files is straight forward. Three, the replication policy that is set for a directory becomes the default policy for all files and subdirectories in that directory. When a policy on a directory is modified the changes are not automatically propagated to the contents of the directory; only newly created files or directories inherit this new policy. To change all policies in a directory, the user must explicitly specify that the new policy should be recursively applied to all the existing policies in the directory. The semantics are identical to that of the *sticky* bit for directories.

3.3 Replica node operations

Replica nodes are responsible for storing data and ensuring that replication policies are satisfied. Once a client replicates a file, the data becomes the responsibility of the replica nodes.

3.3.1 Storing data

Replica nodes receive *store* requests from clients. A *store* request consists of the file data and a small amount of metadata that specifies the modification time of the file and the version number of the replication policy. If the version number of the replication policy sent by the client matches the version number of the policy held by the replica node then the file data is written to disk, replacing the file data previously recorded. Otherwise, either the client or the replica node is using a stale replication policy. The procedure to deal with stale metadata is described in Section 4.2.

3.3.2 Monitoring nodes

Since replica nodes are responsible for ensuring that policies are not violated, they use a heartbeat mechanism to track the liveliness of other nodes. A replica node monitors all adjacent nodes with respect to the node-to-node map. While this set could be large, the frequency of the heartbeat is usually low. Thus in most cases the monitoring load is manageable for the heartbeat mechanism.

Nodes are not automatically monitored. When a new replication policy is created, the nodes in it do not automatically start monitoring each other. Only when they receive the first store request for the replication policy do they start monitoring the other nodes. The optimization follows from the observation that a node is not storing the data cannot help the recovery process, since there is nothing to re-replicate from.

3.4 Heartbeat operations

The heartbeat monitor is a self contained module that pings other heartbeat monitor modules. To reduce the number of heartbeat messages that are sent out, an upcall that resets the heartbeat timer is used to cancel unnecessary heartbeats. In effect, the heartbeats are piggybacked on regular messages.

3.5 Policy oracle operations

The policy oracle is responsible for all decisions regarding node selection in PDR. The replicator and the policy oracle have a client–server relationship, while a peer-to-peer relationship exists between the policy oracles.

The policy oracle handles four types of requests from the replicator: *set policy*, *get policy*, *update policy*, and *get node information*. The most common requests are get policy and set policy. Retrieving policies is the simplest request, the policy oracle performs a database query and returns the results. Setting policies is more complex and is broken up into two cases.

The first case is for a new policy. For each node in the policy, if the policy oracle does not have current node characteristics, it sends a get node information request to the node to obtain the current information. Then the policy oracle inserts the policy into its database and updates the policy and node maps. Once all the maps are updated, the policy oracle sends an update policy request to all the nodes that are explicitly listed in the policy; they in turn update their database. For some nodes the view of the network topology changes even though they are not explicitly listed in the new policy. Their view of the topology eventually becomes consistent via a gossip protocol.

The second case is for an existing policy. The existing policy is retrieved and modified as per request; current information about nodes is obtained as necessary. When the policy is written back into the database its version number is incremented and a copy of the old policy is stored along with the new policy; the reason for keeping a copy of the old policy is discussed in Section 4.2. The updated policy is propagated to nodes in the policy and also to nodes that were removed from the policy.

Apart from communicating with the replicator and other policy oracles that are directly connected by replication policies, the policy oracle also distributes topology update information to other nodes in the system to keep the view of the overall network topology up to date. This is accomplished through the use of a gossip protocol [6]. On a regular basis the policy oracle propagates information about node-to-node and policy-to-node mappings. The information consists of the inter-node connections. Node characteristics are not propagated along with the topology information.

3.6 Node selection

When the policy oracle is selecting a node, either for a new policy or a replacement node, it works to minimize the size of the connected components that form the topology of the PDR system. As mentioned earlier, replication policies create dependencies among the nodes in the policy (with respect to monitoring and maintaining the policy). These dependencies form a clique on the nodes. If a node is involved in more than one policy then that node connects two or more cliques into a single connected component. Ideally, the goal is to minimize the size of these connected components because the number of messages needed to recover a failed node is reduced.

The input to our selection algorithm is the class of node required, say (C), and the policy P that requires the new

```
U = {v ∈ V\P | Class(v) = C};
for each v ∈ U do
    v.overlap = 0;
rof
for each v ∈ P do
    for each u ∈ Γ(v) ∩ U do
        Increment u.overlap;
    rof
rof
T = {t ∈ U | t.overlap = min_{v∈U} (|Γ(v)| − v.overlap)};
S = {t ∈ T | |Γ(t)| = min_{v∈T} |Γ(t)|};
Uniformly select s ∈ S;
```

Figure 3: The node selection algorithm.

node. A benefit of using a database to store the policies and the maps is that by establishing a secondary node index based on the class, queries based on class can be efficiently performed.

For the purpose of conciseness, let $G = (V, E)$ be the graph induced by the node-to-node map, where V corresponds to the nodes and E, the set of edges, corresponds to the dependencies that are encoded by the map. Let $\Gamma(v)$ denote the set of all nodes adjacent to v and assume that each vertex has an "overlap" field, initially set to 0. Although policy $P \subset V \times \mathbb{Z}$, we treat P as a subset of V.

The algorithm (Figure 3) returns s, the selected node. The degree of a vertex, $|\Gamma(v)|$, is already stored in the node-to-node map, we do not need to recompute it. Thus, all set selection statements can each be implemented using single loops. Therefore, the algorithm is linear in the size of V, except the doubly nested loop, which is quadratic in $\gamma = \max_{v \in V} |\Gamma(v)|$. However, since we expect that $\gamma \ll |V|$, the double loop will not dominate the computation, resulting in an algorithm that is linear in the expected case.

4 Failure mode operations

An important aspect of any distributed system is the ability to handle failure conditions. In PDR, the ability to recover from failure conditions is further heightened because it must ensure the replication level of data. Given, that replication is performed asynchronously at the granularity of files and that PDR's architecture is peer-to-peer, special care must be taken to ensure consistency is maintained. PDR considers three general failure modes. The first two are transient network partitions and transient network failures. The third is permanent node failure.

For transient failures the main issue that needs to be addressed is the consistency of metadata. In the design of PDR an assumption is made that transient failures are short term, and thus replication policies are never violated due to transient failures. To deal with transient failures, the system must handle updates that are late, lost, or delivered out of order.

For permanent failures there are two additional issues. Nodes fail in a fail-stop manner, and do not inject corrupted information into the system. First, the system must quickly handle the failure of a node so that replication policies are not violated for extended periods of time and the corrective process should not put excessive load on the system. Second, the detection of a node failure could potentially cause a message storm during the recovery process. The topology of the system must be maintained to prevent such storms from occurring, or are localized.

4.1 Transient errors

Transient failures, such as network partitions, are handled by a combination of heartbeat messages and retransmission of unsent messages. If a client node is unable to send a message to a replica node, it initially retries several times. If after several attemps the client is still unsuccessful, the heartbeat monitor is invoked to monitor the node and inform the client when the node becomes reachable again. At some point, if the node is still not reachable, then it is considered to have failed permanently.

4.2 Stale metadata

During transient outages it is possible that some metadata updates arrive late or out of order. In particular, this can be problematic during policy updates. For example, a node is added to a replication policy but is not informed of this until it receives the first store request from a client.

PDR relies on the version number contained in the replication policy to determine if the policy a replica node is holding is current; the policy version number is attached to each store request. On receiving a store request a replica node compares the received version number with the version number it already stores. If the two match then all is fine. If the received version number is bigger, then the replicator contacts the leader or one of the other nodes in the policy and requests the current policy. If the received number is smaller, then the client is using a stale policy. In this case the current policy is sent to the client and the client re-replicates based on the updated policy.

To avoid disseminating stale information the policy oracles do not send detailed node information or node characteristics when gossipping to other policy oracles. If a node needs the characteristics of another node, then it contacts that node directly. Although node characteristics do not change often, by forcing nodes to obtain this information directly the potential to disseminate and use stale information is reduced.

Upon updating a replication policy a node keeps a copy of the previous policy. In the event of a failure, if the current replica nodes do not have the necessary data, it may be possible to obtain the data from the nodes listed in the old policy; this scenario may occur if the policy was recently changed. One could argue that a history of policy changes should be maintained to facilitate this scenario. It is assumed that policies are not going to change often and that a history of length one is sufficient.

4.3 Client node failure

There are three client node failure scenarios that need to be handled. In the first scenario the client's replication queue is empty. No recovery actions are necessary, whether the client fails permanently or eventually restarts.

In the second scenario the client's replication queue is not empty and no replication policies are partially satisfied. That is, a file was modified and replication events were inserted into the replication queue, but no replication events were dispatched before the node crashed. If the node fails permanently then the modifications are lost. If the node eventually restarts then the modifications are recoverable because the replication queue is persistent. On a restart the client re-builds the replication queue and immediately dispatches all replication events that were missed while the client was down.

In the third scenario the client's replication queue is not empty and some replication policies are partially satisfied. As described earlier, the leader node monitors a client node until in progress replication policies are satisfied. On detecting a client failure, permanent or transient, the leader node immediately replicates the latest update to the other nodes in the policy. If the client returns, on restart all replication events that were part of a partially satisfied policy are removed from the replication queue since the leader node has already propagated the updates to the other replica nodes.

4.4 Replica node failure

The bulk of the failure handling machinery is used to handle failed replica nodes.

4.4.1 Non-leader replicas

When the failure of a replica node is detected, the leader of the replica group is informed by the node(s) that detected the failure. The other nodes in the policy commence heartbeating the leader node at a higher frequency until the recovery process is finished. This is done so that if the leader fails, it can be quickly detected and a new leader established to continue with the restoration process.

Upon receiving notification of a node failure the leader queries the policy oracle to find a replacement node. A cache for recently selected nodes is kept, thus allowing for reuse of recent node selection results. Once a new node is selected the leader sends a policy update message to all the nodes listed in the replication policy; a policy update message is sent for all replication policies affected by the node failure. For example, all files in a directory may have the same replication policy. To reduce the work done by the policy oracle in selecting a new node a cache for recently selected nodes is kept.

Once a replacement node has been selected and all affected replication policies are updated, the leader starts the re-replication process for the data that was hosted on the failed node. The leader queries the policy oracle for a list of files it is responsible for, and propagates the list to the newly selected node. The newly selected node pulls the required data from the available replica nodes and then informs the leader when finished.

4.4.2 Leader nodes

The procedure for handling leader node failures is similar to those of regular replica nodes. When the failure of a leader node is detected, the node with the next smallest staleness factor is automatically selected as the temporary leader. Then the recovery process described in Section 4.4.1 is performed. As a last step, if the newly selected node has the smallest staleness factor then it takes over the duties of the leader. If there are multiple nodes with the same staleness factor then the current policy is to select the leader as the node that is closest with respect to physical proximity and has the smallest IP address.

4.4.3 Informing client nodes of replica node failure

Client nodes are notified by replica nodes of changes to replication policies as a result of a replica node failing. These notifications are primarily done lazily when the client performs a replication. Leader nodes immediately notify client nodes of replication policy changes if the leader is already monitoring the client node.

Changes to an in-progress replication policy affect both client and replica nodes. When the client node receives a notification that a replication policy has change, it examines what part of the replication policy has been satisfied. If the change does not affect what already has been replicated, then the existing replication entries in the replication queue are simply updated. Otherwise, the client node immediately replicates the data to satisfy the updated policy.

Client nodes cannot be notified of changes to any replication policies if the leader node fails. Instead a client node is informed of changes to replication policies when it sends a store request to one of the other replica nodes.

4.4.4 Failure during recovery

Finally, the system must handle double failures when a replica node fails while the system is in recovery mode. Currently, PDR simply recomputes and re-replicates. Some work could potentially be salvaged that was done in the previous attempt to recover, but this adds substantial complexity to the system.

It should be noted that transient failures, such as network partitions, are still expected to occur during recovery. These transient failures are treated in the same manner as when the system is not in recovery mode. Thus, its assumed that the recovery process may take some time to complete if a storm of transient failures is encountered.

5 Evaluation

In the evaluation of PDR we want to demonstrate that the overhead of the system is reasonable, that the system is scalable, and that there is a benefit to using PDR. To show that the overhead added by PDR is reasonable we present micro benchmarks of the normal file system calls. In addition, two common tasks are performed, untarring and compiling, to demonstrate that normal file system usage is not hindered by PDR. To show that the system is scalable we analyze the number of messages required for various operations.

Finally, we want to show that a benefit is obtained from using PDR. We use file system trace and look at what would be replicated using PDR with a simple replication policy and a traditional backup technique. We compute the benefit as the number of bytes that are not replicated by PDR.

5.1 The prototype

The replicator and the policy oracle are user-level servers written in C and use the Berkeley DB [23] for storing all metadata. Both the replicator and policy oracle are trivially portable to any system that supports the POSIX interface and is supported by the Berkeley DB.

The client side replicator requires a file system redirector to interface with the local file system. Currently, the client side replicator only runs under Linux because it interfaces with the Linux Coda [9] module. However, the interface layer is written in such a way that using a different redirector is straight forward, as long as it supports the required set of upcalls. The file system calls of interest are *creat*, *open*, *close*, *mkdir*, *rmdir*, *remove*, and *rename*.

The prototype implements a large subset of the design described in this paper. We have not yet implemented the gossip protocol to disseminate topology information or the handling of cascading failures. Currently, policies are explicitly set by a command line tool, by specifying the staleness factor and, a set of IP addresses or a particular node class. We are aiming to have a GUI interface in the near future.

5.2 Overhead

We use a combination of micro-benchmarks and real-world tasks to demonstrate that the overhead of PDR performance is reasonable. Our experimental setup consists of Pentium III PCs running at 866MHz and Pentium II PCs running at 266MHz. The machines have 256MB of memory and are connected by a 100Mb switched ethernet network. The Pentium IIIs are used as the clients and the Pentium IIs are used as the servers. All numbers reported here are the median of 1000 trials on otherwise unloaded machines and network.

Operation	PDR (μs)	EXT3 (μs)
`creat`	8635.0	251.5
`open`	2010.3	6.1
`close`	1483.8	1.2
`mkdir`	697.3	182.4
`rmdir`	592.7	52.8
`remove`	428.1	49.1
`rename`	1080.0	134.8

Table 1: Timings for *creat*, *open*, *close*, *mkdir*, *rmdir*, *remove*, and *rename*, operations.

We compared PDR to the standard Linux in-kernel EXT3 file system. Table 1 presents measurements for the basic file system operations. For all system calls there is about a 300μs overhead for the upcall to the replicator. For operations that require a `lookup` operation there is an additional cost of another upcall. There is more overhead for `open`, `creat`, and `close`. The `creat` call must instantiate new metadata for a file. The `open` call makes a copy of the file before replying to the client to enable the replicator to replicate it in the background. The `close` call creates the replication event for the recently modified file.

Most file system calls require that metadata be modified in PDR. On average this adds an addition $9000\mu s$ of overhead; the database is configured to be transactional for consistency purposes, which requires the transaction log to be continuously flushed to disk. The majority of the work is considered to be house-cleaning, and is performed after PDR has replied to the redirector. This house-cleaning reduces the file system throughput, but `read` and `write` calls are unaffected because they are not redirected.

We believe that these overheads can be mitigated by using a redirector that performs more of the work in the kernel. Currently, the redirector acts like a detour and PDR performs the bulk of the work for file system calls, and returns the result of the operation to the redirector. If file system operations were entirely performed in the kernel and PDR was simply informed of their occurrence, then a significant amount of the overhead would be eliminated; that is, the redirector should behave like a tee. As we show next, the current overhead does not significantly affect normal file system usage.

To determine the impact of PDR on normal file system usage we ran two additional measurements. First, we untarred the source for the Linux kernel which is 180MB of source in 14500 files. Second, we compiled the PDR system which consists of 560KB of source in 70 files. To untar the Linux kernel source took 132s on PDR and 58s on the local file system. Tar is a file system intensive task; especially when creating a large number of small files. The factor of two slow down is attributed to our reduced throughput. Compiling PDR we get 53s on PDR and 62s on the local file system. In theory, compiling PDR should take approximately the same amount of time, we believe the above discrepancy arises from the caching done by the redirector. We see that for tasks that are not file system intensive, such as compiling, our performance is on par with the local file system.

5.3 Scalability

To demonstrate the scalability of PDR we analyze the number of inter-node messages required to complete common tasks in the system, such as replicating data, disseminating policy information, and handling failure. For the remainder of this section let n denote the number of nodes in the system and let p denote the expected number of nodes in a policy.

The most common message that is sent, apart from a heartbeat message, is a store request from a client node to a replica node. The number of store requests made per file modification is proportional to the number of nodes in the file's policy. Store requests differ from most of the other messages; while the size of other messages is at most several kilobytes, the size of store requests can range on the order of megabytes because they contain the modified file in its entirety. Thus, under normal operations, the amount of data sent is p times the size of the file. However, if the file is modified at a higher frequency than it's staleness factor, the modifications are batched. In this case, less data, on average, is sent. If the modifications overwrite files in their entirety, it is impossible to send less than p times the size of the file, if the file is to be replicated to p different nodes.

We have designed and partially implemented a copy-on-write mechanism to only replicate the blocks that have been modified. A drawback of this mechanism is that considerably more interaction between the redirector and the replicator is necessary because the replicator must keep track of which blocks have been modified. Another alternative is to use diffs, but this uses considerably more CPU resources and is only appropriate for text files, which are generally small.

The next most common message is to set a new policy or to change and existing policy. To set a new policy requires at most $3p$ messages. The policy oracle, which creates the policy, requests updates from each of the nodes that has been added to the policy, resulting in p messages being sent. Each of the nodes responds, resulting in another p messages. Finally, the policy oracle sends out the new policy to each of the participating nodes resulting in a final p messages. If the policy oracle believes that it has up to date information about some of the participating nodes, then a fraction of the first $2p$ messages is avoided.

To change a policy requires at most $3p_{\text{new}} + p_{\text{old}}$ because both the new and the old replica nodes must be informed of the change; p_{new} is the number of nodes in the new policy and p_{old} is the number of nodes in the old policy. As before, up to $2p_{\text{new}}$ messages are needed to retrieve up to date node information for the new replica nodes and p_{new} policy updates must be sent. However, the old nodes in the policy must also be notified, of which there are at most p_{old} nodes.

The heart beat messages are small, and usually piggy-backed on other messages. However, if the system is quiescent, p^2 messages per policy are sent over a relatively large period of time. Since we expect that $p \ll n$, the cost of these messages is dwarfed by all others. Therefore, under normal operation the message cost of file or policy modification is linear in the size of the policy.

5.3.1 Failure messages

To restore a replica failed replica node requires the largest number of messages. When a node detects that a replica node has failed, it notifies the leaders of the policies in which the failed node was participating. In the worst case the number of messages sent is n^2; this occurs if $p = n$ and there are p different policies, each with a different leader. is the total number of nodes in the system. However, since $p \ll n$ and the selection algorithm clusters nodes, the expected case is approximately the square of the adjacency set of the failed node. The number of messages, linear in the size of the adjacency set is then sent out by the leaders to change the policies, replacing the dead node. The re-replication costs are costs are linear in the amount of data residing on the failed node.

Other failure scenarios, such as the failure of a leader node, are similar in cost to the failure of a replica node. The predominant cost are the notification messages and the re-replication requests. These other failure scenarios add a small and constant number of messages (a dozen messages or so) to the general recovery procedure.

In general, we foresee users using a small number of policies on a small number of replica nodes to replicate their data; on the order of five to ten policies and nodes are used. Given this usage and the goals of the node selection algorithm, we foresee the topology being many cliques consisting of five to ten nodes, and several cliques creating a single connected component of 20 to 30 nodes. Thus, if we go by these numbers then the recovery process scales well for any number of nodes in the system.

5.4 Benefits

To evaluate the benefit of PDR we compared it to a traditional backup technique. The benefit is calculated as the number of bytes not replicated by PDR. The file system trace was gathered over a period of a month, September 2003, in the Department of Computer Science at the University of British Columbia. The traces consisted of the set of files modified in a 24 hour period for nine users.

Two replication policies were created. The first replication policy is equivalent to traditional backup procedures, where data is replicated to the server room once every 24 hours; the replication policy replicates only important files. The files deemed unimportant are the web browser cache, object files, and auxiliary files generated by applications such as Latex. The second replication policy is similar to the first, in addition, important files, e.g., Latex and Word files, are replicated within 12 hours.

On average, the nine users generated approximately 500MB of data for the nightly backup. One participant owned an unusually large mailbox which contributed 30-40% of the total nightly backup. On average a user's mailbox amounts to 10% of the total. On average, we found that the web browser cache and object files made up for 9% of the data being replicated. This is data that is easily re-creatable and does not need to be replicated. Pictures, postscript, and PDF files comprised another 4%; a large portion of which probably does not need to be replicated. User's documents and source files contributed about 9% of the total. The first policy would provide a savings of about 10%, and the second would enable users to retrieve a file that is at most 12 hours old verses 24 hours at no extra cost.

The other 70% of the data is in unclassified files. For privacy reasons we only had the size, name, and extension of the file. The replication policies were created solely based on file extensions. In addition, no MP3s were found in the participants home directory. We believe the reason for this is that home directories are limited to 100MB, and thus users store large or less important data on storage servers that are backed up less often.

6 Related work

In the last twenty years a large number of backup and replication systems have been proposed and built. These systems provide a varying amount of protection against data loss for different types of failures. The standard approach is to uniformly replicate all the data. When and where the data is replicated is specified by a static policy.

RAID [16, 28] provides solid protection against media failure. A large virtual disk is created from a number of smaller disks. When data is written it is automatically replicated on one or more disks (depending on the configuration), thus a disk failure does not cause data loss. Petal [11] and Zebra [7] as a step further by creating the virtual disk on top of a collection of physical machines, thus adding protection against machine failure.

Data is not only lost due to failure but also to intentional or accidental corruption. To recover from this type of failure systems such as WAFL [8], Coda [21], and AFS [13] use checkpointing. Although this is an elegant and simple approach to providing protection against data corruption, it is expensive, with respect to bandwidth and time, to replicate due to the large size of present file systems; and they are continuously growing. Thus checkpointing alone is insufficient to provide protection against any other types of failures.

Distributed file systems do a good job of providing protection for data against local machine failure, but have more difficulty providing protection against site and wider area failures. Systems such as xFS [2], AFS [13], Coda [21], and Frangipani [26] must maintain some amount of global state in addition to uniformly replicating all data. This requirement increases the coupling among the nodes, increasing complexity, and hindering scalability, making recovery more difficult. The architects of Ficus [15] recognized this fact and created a system that did not maintain any global data structures and the nodes were loosely coupled. Like AFS, Ficus partitions the file system into volumes. Within each volume a gossip style protocol [6] is used to propagate updates to all the nodes in a volume. Although data is not replicated as strictly as it is in other systems, all data in a volume gets replicated to the same level; the replication policies are static.

Recently, a large body of work has been done in the area of peer-to-peer storage systems [5, 10, 14, 20, 19, 25]. Many use a distributed hash table that locates file data using $O(\log N)$ messages among N peer nodes. Each node is assigned a quasi-random ID; similarly, files (or blocks) are assigned an ID taken from the same space and are stored on nodes with numerically closest IDs. The key to tolerating failure is the random assignment of node IDs; this ensures that nodes with similar IDs are unrelated to each other in the underlying network topology. Data replicated on nodes with consecutive IDs is thus reasonably invulnerable to a single point of failure. The compelling simplicity of this approach comes with endemic limitations. A file's (or block's) location is randomly chosen and fixed when it is first added to the system. Some data, financial or medical for example, is too important to store on some node that you don't know or trust; a malicious node could corrupt the data. Thus, for some data it is desirable to explicitly specify the location of the replicas. In addition, one should not assume that the distribution for node failure is identical or independent [12, 27].

Farsite [1] is a peer-to-peer storage system that is most similar in goals and architecture to PDR. It assumes that all nodes are connected by a high-bandwidth, low-latency network and that all nodes are identical. This assumption becomes invalid as the system moves to a wider area. Nodes are generally not identical, they can vary greatly with respect to reliability and their connection to the network; the cost of replicating is no longer uniform across all nodes. PDR is designed to handle these discrepancies by selecting nodes that satisfy the replication requirements at minimal cost.

7 Conclusions

Traditionally, replication systems replicate data in its entirety, without consideration of its importance, or against what failures it must be protected. However, as commodity storage grows exponentially and is filled at roughly the same rate, the strategy of wholesale replication becomes untenable. This is due to growth of other resources, such as network bandwidth, are simply not keeping pace; the cost of putting in another hard drive is dwarfed by the cost of upgrading the backbone or even the local network routers. Thus, to ensure that important data is protected against the appropriate failures the degree of replication must be related to the value of the data; this value can only be determined by the user.

We have encapsulated this notion in PDR, a Policy Driven Replication system that allocates replication resources based on per file policies that can be dynamically set and changed by the user. The PDR system classifies nodes according to a set of reliability measures and uses this classification scheme to decide where to replicate what data, facilitating the protection of data against specific classes of failures. The system constantly monitors the health of nodes on which data is stored; not only does this facilitate quick detection of failures, but also allows the system to select appropriate replacement nodes. Finally, the system avoids inducing additional overhead on the critical path of file system calls, enabling background replication without affecting system performance.

To achieve the opposing goals of ongoing system monitoring and minimizing overhead, the system uses the policies specified by the users to avoid unnecessary messages. Each policy includes a staleness factor for each node. The factor denotes the maximum time span between data modification and replication, and dictates how often a replica must be monitored. Upon failure, the high level part of the system uses the policies and the monitoring information to select an appropriate replacement node. The node is stored as part of the policy, which is referenced by the low level replicator to schedule replication events, thus, avoiding costly queries to the high level policy oracle.

Our evaluation of the system confirmed that for normal file system usage the performance is reasonable. For tasks such as document processing and software development PDR performs on par with local file systems. A significant portion of user's files are unimportant or easily re-creatable. With a small set of replication policies a user can eliminate the replication of unimportant data, and increase the availability and reliability of important data.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and

Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, December 2002.

[2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[3] Network Appliance. http://www.netapp.com.

[4] EMC^2 Corporation. http://www.emc.com.

[5] Frank Dabek, M. Frans Kasshoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, October 2001.

[6] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *In Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001)*, July 2001.

[7] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[8] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pages 235–246, Winter 1994.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, October 1991.

[10] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, November 2000.

[11] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pages 84–93, October 1996.

[12] Keith Marzullo. Theory and practice for fault-tolerant protocols on the internet, October 2002.

[13] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[14] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, December 2002.

[15] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 28(2):155–180, February 1998.

[16] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference, Chicago, Illinois, June 1–3*, pages 109–116, 1988.

[17] David A. Patterson. A conversation with Jim Gray. *ACM Queue*, 1(4):53–56, June 2003.

[18] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, November 2001.

[19] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer stroage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[20] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systans Design and Implementation (OSDI-02)*, December 2002.

[21] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. In *Second IEEE Workshop on Workstation Operating Systems*, September 1989.

[22] D. Simpson. Does HSM pay? be skeptical! *Datamation*, 41(14):322–337, August 1995.

[23] Sleepycat Software. The berkeley database (berkeley db).

[24] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, Massachusetts Institute of Technology, March 2001.

[25] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[26] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[27] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proceedings of International Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 362–367, October 2002.

[28] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.