17th ACM Symposium on Operating System Principles (SOSP'99)
Published as *Operating Systems Review* **34**(5):1–15, Dec. 1999

Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service

Yasushi Saito, Brian N. Bershad, and Henry M. Levy
Department of Computer Science and Engineering
University of Washington
{*yasushi, bershad, levy*}*@cs.washington.edu*

Abstract

This paper describes the motivation, design, and performance of Porcupine, a scalable mail server. The goal of Porcupine is to provide a highly available and scalable electronic mail service using a large cluster of commodity PCs. We designed Porcupine to be easy to manage by emphasizing dynamic load balancing, automatic configuration, and graceful degradation in the presence of failures. Key to the system's manageability, availability, and performance is that sessions, data, and underlying services are distributed homogeneously and dynamically across nodes in a cluster.

1 Introduction

The growth of the Internet has led to the need for highly scalable and highly available services. This paper describes the Porcupine scalable electronic mail service. Porcupine achieves scalability by clustering many small machines (PCs), enabling them to work together in an efficient manner. In this section, we describe system requirements for Porcupine, relate the rationale for choosing a mail application as our target, and review clustering alternatives.

1.1 System requirements

Porcupine defines scalability in terms of three essential system aspects: manageability, availability, and performance. Requirements for each follow:

1. **Manageability requirements.** Although a system may be physically large, it should be easy to manage. In particular, the system must *self-configure* with respect to load and data distribution and *self-heal* with respect to failure and recovery. A system manager can simply add more machines or disks to improve throughput and replace them when they break. Over time, a system's nodes will perform at differing capacities, but these differences should be masked (and managed) by the system.
2. **Availability requirements.** With so many nodes, it is likely that some will be down at any given time. Despite component failures, the system should deliver good service to *all* of its users at all times. In practice, the failure of one or more nodes may prevent some users from accessing some of their mail. However, we strive to avoid failure modes in which whole groups of users find themselves without any mail service for even short periods of time.
3. **Performance requirements.** Porcupine's single-node performance should be competitive with other single-node systems; its aggregate performance should scale linearly with the number of nodes in the system. For Porcupine, we target a system that scales to hundreds of machines, which is sufficient to service a few billion mail messages per day with today's commodity PC hardware and system area networks.

Porcupine meets these scalability requirements uniquely. First, the system is *functionally homogeneous*. That is, any node can execute part or all of any transaction, e.g., for the delivery or retrieval of mail. Second, every transaction is *dynamically scheduled* to ensure that work is uniformly distributed across all nodes in the cluster. Third, the system *automatically reconfigures* whenever nodes are added or removed even transiently. Finally, system and user data are automatically *replicated* across a number of nodes to ensure availability.

Figure 1 shows the relationships among the central goal of scalability, the requirements for manageability, availability, and performance, and the key features or techniques used in the system. For example, dynamic transaction scheduling, automatic reconfiguration, and functional homogeneity together make the system manageable, since changes to the number or quality of machines, users, and load are handled automatically. Similarly, dynamic scheduling and replication improve performance, because load can be distributed dynamically to less busy machines.

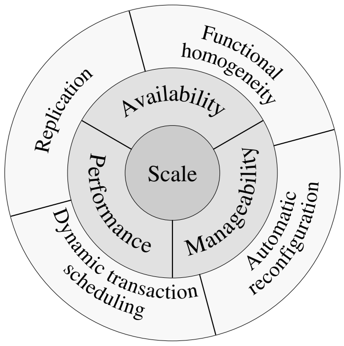

Figure 1. The primary goal of Porcupine is scalability defined in terms of manageability, availability, and performance requirements. In turn, these requirements are met through combinations of the four key techniques shown above.

Today, Porcupine runs on a cluster of thirty PCs connected by a high-speed network, although we show that it is designed to scale well beyond that. Performance is linear with respect to the number of nodes in the cluster. The system adapts automatically to changes in workload, node capacity, and node availability. Data is available despite the presence of a few or many failures.

1.2 Rationale for a mail application

Although Porcupine is a mail system, its underlying services and architecture are appropriate for other systems in which data is frequently written and good performance, availability, and manageability at high volume are demanded. For example, Usenet news, community bulletin boards, and large-scale calendar services are good candidates for deployment using Porcupine. Indeed, we have configured Porcupine to act as a web server and a Usenet news node. In this paper, however, we focus on the system's use as a large scale electronic mail server.

We chose a mail application for several reasons. First is need. Large-scale commercial services now handle more than ten million messages per day. Anticipating continued growth, our goal with Porcupine is to handle billions of messages per day on a PC-based cluster. Second, email presents a more challenging application than that served by conventional web and proxy servers, which have been shown to be quite scalable. In particular, the workload for electronic mail is write intensive. Finally, consistency requirements for mail, compared to those for a distributed file or database system, are weak enough to encourage the use of replication techniques that are both efficient and highly available.

1.3 Clustering alternatives

Existing clustering solutions have taken one of two forms. One approach is seen with services that deliver read-only data, such as Web servers, search engines, or proxy agents. Here, administrators typically replicate data across a large number of back-end hosts and deploy a front-end traffic manager that routes requests to specific servers using a dynamic policy (e.g., based on the server's specific function [11], load or cache affinity [21]). This approach works well in a read-only context; however, it is inappropriate for services such as electronic mail, in which data is frequently written as well as read, and for which persistence and data availability are not provided by some external storage service.

A second approach to clustering, used when data is associated with specific users, is to assign users and their data statically to specific machines [4, 9]. A front-end intelligent router directs an external client's request to the appropriate node. We believe that such statically distributed, write-oriented services scale poorly. In particular, as the user base grows, so does service demand, which can be met only by adding more machines. Unfortunately, each new machine must be configured to handle a subset of the users, requiring that users and their data migrate from older machines. As more machines are added, the likelihood that at least one of them is inoperable grows, diminishing availability for users with data on the inoperable machines. In addition, users whose accounts are on slower machines tend to receive worse service than those on faster machines. Finally, a statically distributed system is susceptible to overload when traffic is distributed non-uniformly across the user base.

To date, systems relying on static distribution have worked for two reasons. First, service organizations have been willing to substantially overcommit computing capacity to mitigate short-term load imbalances. Second, organizations have been willing to employ people to reconfigure the system manually in order to balance load over the long term. Because the degree of overcapacity determines where short-term gives way to long-term, static systems have been costly in terms of hardware, people, or both. For small static systems, these costs have not been substantial; for example, doubling the size of a small but manageable system may yield a system that is also small and manageable. However, once the number of machines becomes large (i.e, on the order of a few dozen), disparate (i.e., fast/slow machines, fast/slow disks, large/small disks), and continually increasing, this gross overcapacity becomes unacceptably expensive in terms of hardware and people.

Porcupine seeks to address these problems by providing a system structure that performs well as it scales, adjusts automatically to changes in configuration and load, and is easy to manage. Our vision is that a single system administrator can be responsible for the hardware that supports the mail requirements of one hundred million users processing a billion messages per day. When the system begins to run out of capacity, that administrator can improve performance for all users simply by adding machines or even disks to the system. Lastly, the administrator can, without inconveniencing users, attend to the failure of machines, replacing them with the same urgency with which one replaces light bulbs.

1.4 Organization of the paper

The remainder of this paper describes Porcupine's architecture, implementation, and performance. Section 2 presents an overview of the system's architecture. Section 3 describes how the system adapts to changes in configuration automatically, while Section 4 reveals Porcupine's approach to availability. In Section 5 we describe the system's scalable approach to fine-grained load balancing. Section 6 evaluates the performance of the Porcupine prototype on our 30-node cluster. Section 7 discusses some of the system's scalability limitations and areas for additional work. In Section 8 we discuss related work, and we draw conclusions in Section 9.

2 System architecture overview

A key aspect of Porcupine is its *functional homogeneity*: any node can perform any function. This greatly simplifies system configuration: the system's capacity grows and shrinks with the number and aggregate power of the nodes, not with how they are logically configured. Consequently, there is no need for a system administrator to make specific service or data placement decisions. This attribute is key to the system's manageability.

Functional homogeneity ensures that a service is always available, but it offers no guarantees about the data that service may be managing. *Replicated state* serves this purpose. There are two kinds of replicated state that Porcupine must manage: hard state and soft state. *Hard state* consists of information that cannot be lost and therefore must be maintained in stable storage. For example, an email message and a user's password are hard state. Porcupine replicates hard state so that its derived soft state can be reconstructed during failure. *Soft state* consists of information that, if lost, can be reconstructed from existing hard state. For example, the list of nodes containing mail for a particular user is soft state, because it can be reconstructed by a distributed disk scan. Most soft state is maintained on only one node at a given instant. The exception is directories that are *themselves* soft state. Such directories are replicated on every node to improve performance.

This approach minimizes persistent store updates, message traffic, and consistency management overhead. The disadvantage is that soft state may need to be reconstructed from distributed persistent hard state after a failure. Our design seeks to ensure that these reconstruction costs are low and can scale with the size of the system. In Section 6, we demonstrate the validity of this assumption by showing that reconstruction has nominal overhead.

The following subsections describe Porcupine's data structures and their management.

2.1 Key data structures

Porcupine is a cluster–based, Internet mail service that supports the SMTP protocol [22] for sending and receiving messages across the Internet. Users retrieve their messages using any mail user agent that supports either the POP or IMAP retrieval protocols [20, 7].

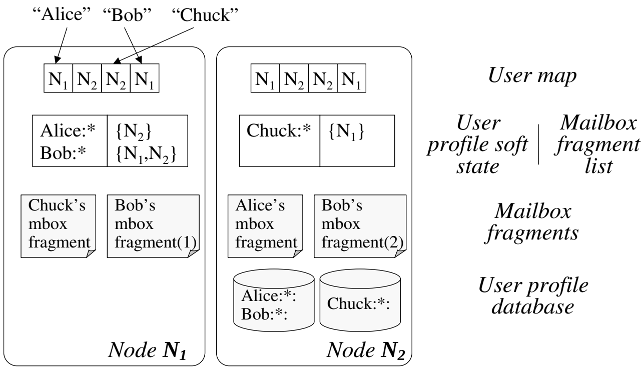

Figure 2. An example showing how user information and mail messages might be distributed across a two-node Porcupine cluster.

Porcupine consists of a collection of data structures and a set of internal operations provided by managers running on every node. The key data structures found in Porcupine are:

Mailbox fragment. The collection of mail messages stored for a given user at any given node is called a *mailbox fragment*; the fragment is also the unit of mail replication. A Porcupine mailbox is therefore a logical entity consisting of a single user's mailbox fragments distributed and replicated across a number of nodes. There is no single mailbox structure containing all of a user's mail. *A mailbox fragment is hard state.*

Mailbox fragment list. This list describes the nodes containing mailbox fragments for a given user. *The mailbox fragment list is soft state.*

User profile database. This database describes Porcupine's client population, i.e., it contains user names, passwords, etc. It is persistent, changes infrequently for a given user, and is partitioned and replicated across nodes. *The user profile database is hard state.*

User profile soft state. Each entry in the user profile database is kept as soft state on exactly one node in the cluster. Accesses and updates to the user profile database are done first at that node. *This data structure is soft state.*

User map. The user map is a table that maps the hash value of each user name to a node currently responsible for managing that user's profile soft state and mailbox fragment list. *The user map is soft state and is replicated on each node.*

Cluster membership list. Each node maintains its own view of the set of nodes currently functioning as part of the Porcupine cluster. Most of the time, all nodes perceive the same membership, although a node's arrival or departure may cause short-term inconsistencies as the system establishes the new membership. During network partition, inconsistencies may last for a long time. Various system services automatically respond to changes in the cluster membership list. *The cluster membership list is soft state and is replicated on each node.*

2.2 Data structure managers

The preceding data structures are distributed and maintained *on each node* by several essential managers. The *user manager* manages soft state that can be reconstructed from disk. By spreading the responsibility for servicing accesses to the user profile database across all nodes in the system, larger user populations can be supported simply by adding more machines. The user manager also maintains the mailbox fragment list for each user.

The *membership manager* on each node maintains that node's view of the overall cluster state. It tracks which nodes are up or down and the contents of the user map. It also participates in a membership protocol to track that state.

Two other managers, the *mailbox manager* and the *user profile manager*, maintain persistent storage and enable remote access to mailbox fragments and user profiles.

Each node runs a *replication manager*, which ensures the consistency of replicated objects stored in that node's local persistent storage.

On top of these managers, each node runs a *delivery proxy* to handle incoming SMTP requests and a *retrieval proxy* to handle POP and IMAP requests.

The Porcupine architecture leads to a rich distribution of information in which mail storage is decoupled from user management. For example, Figure 2 shows a sample Porcupine configuration consisting of two nodes and three users. For simplicity, messages are not shown as replicated. The user manager on node N_1 maintains Alice's and Bob's soft state, which consists of their user profile database entries and their mailbox fragment lists. Similarly, the user manager on node N_2 maintains Chuck's soft state.

2.3 A mail transaction in progress

In failure-free operation, mail delivery and retrieval work as follows.

2.3.1 Mail delivery

An external mail transfer agent (MTA) delivers a message to a user who is hosted on a Porcupine cluster by discovering the IP address of any Porcupine cluster node using the Internet's Domain Name Service [3]. Because any function can execute on any node, there is no need for special front-end request routers, although nothing in the system prevents their use.

To initiate mail delivery, the MTA uses SMTP to connect to the designated Porcupine node, which acts as a delivery proxy. The proxy's job is to store the message on disk. To do this, it retrieves the mailbox fragment list from the recipient's user manager and then chooses the best node from that list. If the list is empty or all choices are poor (for example, overloaded or out of disk space), the proxy is free to select any other node. The proxy then forwards the message to the chosen node's mailbox manager for storage. The storing node ensures that its participation is reflected in the user's mailbox fragment list. If the message is to be replicated (based on information in the user's profile), the proxy selects multiple nodes on which to store the message.

2.3.2 Mail retrieval

An external mail user agent (MUA) retrieves messages for a user whose mail is stored on a Porcupine cluster using either the POP or IMAP transfer protocols. The MUA contacts any node in the cluster to initiate the retrieval. The contacted node, acting as a proxy, authenticates the request through the user manager for the client and discovers the mailbox fragment list. It then contacts the mailbox manager at each node storing the user's mail to request mail digest information, which it returns to the MUA. Then, for each message requested, the proxy fetches the message from the appropriate node or nodes. If the external agent deletes a message, the proxy forwards the deletion request to the appropriate node or nodes. When the last message for a user has been removed from a node, that node removes itself from the user's mailbox fragment list.

2.4 Advantages and tradeoffs

By decoupling the delivery and retrieval agents from the storage services and user manager in this way, it is always possible to deliver or retrieve mail for a user, even when nodes storing the user's existing mail are unavailable. Another advantage is that mail delivery can be load balanced dynamically; any node can store mail for any user, and no single node is permanently responsible for a user's mail or soft profile information. A user's mail can be replicated arbitrarily, independent of the replication factor for other users. If a user manager goes down, another will take over for that manager's users. In contrast, in a system where user mailboxes and/or profile information are fixed to particular nodes, some nodes may become overloaded while others idle.

The system architecture reveals a key tension that must be addressed in the implementation. Specifically, while a user's mail may be distributed across a large number of machines, doing so complicates both delivery and retrieval. On delivery, each time a user's mail is stored on a node not already containing mail for that user, the user's mailbox fragment list (a potentially remote data structure) must be updated. On retrieval, aggregate load increases somewhat with the number of nodes storing the retrieving user's mail. Consequently, it is beneficial to limit the spread of a user's mail, widening it primarily to deal with load imbalances and failure. In this way, the system behaves (and performs) like a statically partitioned system when there are no failures and load is well-balanced, but like a dynamically partitioned system otherwise. Section 5 discusses this tradeoff in more detail.

3 Self management

Porcupine must deal automatically with diverse changes, including node failure, node recovery, node addition, and network failure. In addition, change can come in bursts, creating long periods of instability, imbalance and unavailability. It is a goal of Porcupine to manage change automatically in order to provide good service even during periods of system flux.

The following sections describe the Porcupine services that detect and respond to configuration changes.

3.1 Membership services

Porcupine's cluster membership service provides the basic mechanism for tolerating change. It maintains the current membership set, detects node failures and recoveries, notifies other services of changes in the system's membership, and distributes new system state. We assume a symmetric and transitive network in steady state, so that nodes eventually converge on a consistent membership set provided that no failure occurs for a sufficiently long period (i.e., a few seconds).

The cluster membership service uses a variant of the Three Round Membership Protocol [5] (TRM) to detect membership changes. In TRM, the first round begins when any node detects a change in the configuration and becomes the coordinator. The coordinator broadcasts a "new group" message together with its Lamport clock [14], which acts as a proposed epoch ID to identify a particular membership incarnation uniquely. If two or more nodes attempt to become a coordinator at the same time, the one proposing the largest epoch ID wins.

In the second round, all nodes that receive the "new group" message reply to the coordinator with the proposed epoch ID. After a timeout period, the coordinator defines the new membership to be those nodes from which it received a reply. In the third round, the coordinator broadcasts the new membership and epoch ID to all nodes.

Once membership has been established, the coordinator periodically broadcasts probe packets over the network. Probing facilitates the merging of partitions; when a coordinator receives a probe packet from a node not in its current membership list, it initiates the TRM protocol. A newly booted node acts as the coordinator for a group in which it is the only member. Its probe packets are sufficient to notify others in the network that it has recovered.

There are several ways in which a node may discover the failure of another. The first is through a timeout that occurs normally during part of a remote operation. In addition, nodes within a membership set periodically "ping" their next highest neighbor in IP address order, with the largest IP address pinging the smallest. If the ping is not responded to after several attempts, the pinging node becomes the coordinator and initiates the TRM protocol.

3.2 User map

The purpose of the user map is to distribute management responsibility evenly across live nodes in the cluster. Whenever membership services detect a configuration change, the system must reassign that management responsibility. Therefore, like the membership list, the user map is replicated across all nodes and is recomputed during each membership change as a side effect of the TRM protocol.

After the second round, the coordinator computes a new user map by removing the failed nodes from the current version and uniformly redistributing available nodes across the user map's hash buckets (the user map has many buckets, so a node typically is assigned to more than one bucket). The coordinator minimizes changes to the user map to simplify reconstruction of other soft state, described in the next section. For each bucket with a changed assignment, the coordinator assigns to and includes with the bucket a timestamp equal to the current epoch ID. The ID is used by other nodes to determine which entries in the user map have changed. The new user map is piggybacked on the final broadcast message of the TRM protocol.

3.3 Soft state reconstruction

Once the user map has been reconstructed, it is necessary to reconstruct the soft state at user managers with new user responsibilities. Specifically, this soft state is the user profile soft state and the mailbox fragment list for each user. Essentially, every node pushes soft state corresponding to any of its hard state to new user managers responsible for that soft state.

Reconstruction is a two-step process, completely distributed, but unsynchronized. The first step occurs immediately after membership reconfiguration. Here, each node compares the previous and current user maps to identify any buckets having fresh assignments. A node considers a bucket assignment fresh if the bucket's previous epoch ID does not match the current epoch ID. Recall that the user map associates nodes with hash buckets, so the relevant soft state belonging on a node is that corresponding to those users who hash into the buckets assigned to the node.

Each node proceeds independently to the second step. Here, every node identifying another node's fresh bucket assignment sends it any soft state corresponding to the hard state for that bucket maintained on the sending node. First, the node locates any mailbox fragments belonging to users in the freshly managed bucket and requests that the new manager include this node in those users' mailbox fragment lists. Second, the node scans its portion of the stored user profile database and sends to the fresh manager all pertinent user profiles. As the user database is replicated, only the replica with the largest IP address among those functioning does the transfer. The hard state stored on every node is "bucketed" into directories so that it can be quickly reviewed and collected on each change to the corresponding bucket in the user map.

The cost of rebuilding soft state during reconfiguration is intended to be constant regardless of cluster size. The cost is mostly determined by the number of nodes redistributed within the user map after each failure. It therefore decreases linearly with cluster size. Although the rate of reconfiguration increases linearly with cluster size (assuming independent failures), the two effects cancel each other out, and the work done by each node after a failure remains about the same.

3.4 Node addition

Porcupine's automatic reconfiguration structure makes it easy to add a new node to the system. A system administrator simply installs the Porcupine software on the node. When the software boots, it is noticed by the membership protocol and added to the cluster. Other nodes see the configuration change and upload soft state onto the new node. To make the host accessible outside of Porcupine, the administrator may need to update border naming and routing services. Occasionally, a background service rebalances replicated user database entries across the nodes in the cluster[1].

3.5 Summary

Porcupine's dynamic reconfiguration protocols ensure that the mail service is always available for any given user and facilitate the reconstruction and distribution of soft state. The next section discusses the maintenance of hard state.

4 Replication and availability

This section describes object replication support in Porcupine. As in previous systems (e.g., [11]), Porcupine defines semantics tuned to its application requirements. This permits a careful balance between behavior and performance.

Porcupine replicates the user database and mailbox fragments to ensure their availability. Our replication service provides the same guarantees and behavior as the Internet's electronic-mail protocols. For example, Internet email may arrive out of order, on occasion more than once, and may sometimes reappear after being deleted. These anomalies are artifacts of the non-transactional nature of the Internet's mail protocols. Porcupine never loses electronic mail unless all nodes on which the mail has been replicated are irretrievably lost.

4.1 Replication properties

The general unit of replication in Porcupine is the *object*, which is simply a named byte array that corresponds to a single mail message or the profile of a single user. A detailed view of Porcupine's replication strategy includes these five high-level properties:

1. **Update anywhere.** An update can be initiated at any replica. This improves availability, since updates need not await the revival of a primary. This strategy also eliminates the requirement that failure detection be precise, since there need not be agreement on which is the primary node.
2. **Eventual consistency.** During periods of failure, replicas may become inconsistent for short periods of time, but conflicts are eventually resolved. We recognize that single-copy consistency [12] is too strong a requirement for many Internet-based services, and that replica inconsistencies are tolerable as long as they are resolved eventually. This strategy improves availability, since accesses may occur during reconciliation or even during periods of network partitioning.
3. **Total update.** An update to an object totally overwrites that object. Since email messages are rarely modified, this is a reasonable restriction that greatly simplifies update propagation and replica reconciliation, while keeping costs low.
4. **Lock free.** There are no distributed locks. This improves performance and availability and simplifies recovery.
5. **Ordering by loosely synchronized clocks.** The nodes in the cluster have loosely synchronized clocks [17, 18] that are used to order operations on replicated objects.

[1] In the current implementation, the rebalancer must be run manually.

The update-anywhere attribute, combined with the fact that any Porcupine node may act as a delivery agent, means that incoming messages are never blocked (assuming at least one node remains functional). If the delivery agent crashes during delivery, the initiating host (which exists outside of Porcupine) can reconnect to another Porcupine node. If the candidate mailbox manager fails during delivery, the delivery agent will select another candidate until it succeeds. Both of these behaviors have the same potential anomalous outcome: if the failure occurs after the message has been written to stable storage but before any acknowledgement has been delivered, the end user may receive the same message more than once. We believe that this is a reasonable price to pay for service that is continually available.

The eventual-consistency attribute means that earlier updates to an object may "disappear" after all replica inconsistencies are reconciled. This behavior can be confusing, but we believe that this is more tolerable than alternatives that block access to data when replica contents are inconsistent. In practice, eventual consistency for email means that a message once deleted may temporarily reappear. This is visible only if users attempt to retrieve their mail during the temporary inconsistency, which is expected to last at most a few seconds.

The lock-free attribute means that multiple mail-reading agents, acting on behalf of the same user at the same time, may see inconsistent data. However, POP and IMAP protocols do not require a consistent outcome with multiple clients concurrently accessing the same user's mail.

The user profile database is replicated with the same mechanisms used for mail messages. Because of this, it is possible for a client to perceive an inconsistency in its (replicated) user database entry during node recovery. Operations are globally ordered by the loosely synchronized clocks; therefore, a sequence of updates to the user profile database will eventually converge to a consistent state. We assume that the maximum skew of the loosely synchronized clocks is less than the inter-arrival time of externally initiated, order-dependent operations, such as Create-User and Change-Password. In practice, clock skew is usually on the order of tens of microseconds[18], whereas order-dependent operations are separated by networking latencies of at least a few milliseconds. Wall clocks, not Lamport clocks [14], are used to synchronize updates, because wall clocks can order events that are not logically related (e.g., an external agent contacting two nodes in the cluster serially).

We now describe the replication manager, email operations using replicas, and the details of updating replicated objects.

4.2 Replication manager

A replication manager running on each host exchanges messages among nodes to ensure replication consistency. The manager is oblivious to the format of a replicated object and does not define a specific policy regarding when and where replicas are created. Thus, the replication manager exports two interfaces: one for the creation and deletion of objects, which is used by the higher level delivery and retrieval agents, and another for interfacing to the specific managers, which are responsible for maintaining replicated objects. The replication manager does not coordinate object reads. Mail retrieval proxies are free to pick any replica and read them directly.

4.3 Sending and retrieving replicated mail

When a user's mail is replicated, that user's mailbox fragment list reflects the set of nodes on which each fragment is replicated. For example, if Alice has two fragments, one replicated on nodes N_1 and N_2 and another replicated on nodes N_2 and N_3, the mailbox fragment list for Alice records $\{\{N_1,N_2\}, \{N_2,N_3\}\}$. To retrieve mail, the retrieval agent contacts the least-loaded node for each replicated mailbox fragment to obtain the complete mailbox content for Alice.

To create a new replicated object (as would occur with the delivery of a mail message), an agent generates an object ID and the set of nodes on which the object is to be replicated. An *object ID* is simply an opaque, unique string. For example, mail messages have an object ID of the form ⟨*type, username, messageID*⟩, where *type* is the type of object (mail message), *username* is the recipient, *and messageID* is an unique mail identifier found in the mail header.

4.4 Updating objects

Given an object ID and an intended replica set, a delivery or retrieval agent can initiate an update request to the object by sending an update message to any replica manager in the set. A delivery agent's update corresponds to the storing of a message. The retrieval agent's update corresponds to the deletion and modification of a message.

The receiving replica acts as the update coordinator and propagates updates to its peers. The replication manager on every node maintains a persistent update log, used to record updates to objects that have not yet been accepted by all replica peers maintaining that object. Each entry in the update log is the tuple ⟨*timestamp, objectID, target-nodes, remaining-nodes*⟩.

- *Timestamp* is the tuple ⟨*wallclock time, nodeID*⟩, where *wallclock time* is the time at which the update was accepted at the coordinator named by *nodeID*. Timestamp uniquely identifies and totally orders the update.
- *Target-nodes* is the set of nodes that should receive the update.
- *Remaining-nodes* is the set of peer nodes that have not yet acknowledged the update. Initially, *remaining-nodes* is equal to *target-nodes* and is pruned by the coordinator as acknowledgments arrive.

The coordinating replication manager works through the log, attempting to push updates to all the nodes found in the *remaining-nodes* field of an entry. Once contact has been made with a remaining node, the manager sends the replica's contents and the log entry to the peer. Since updates to objects are total, multiple pending updates to the same object on a peer are synchronized by discarding all but the newest. If no pending update exists, or if the update request is the newest for an object, the peer adds the update to the log, modifies the replica, and sends an acknowledgement to the coordinator. Once the coordinator receives acknowledgements from all replica peers, it *retires* the update entry in its own log (freeing that log space) and then notifies the peers that they may also retire the entry.

If the coordinator fails before responding to the initiating agent, the agent will select another coordinator. For updates to a new object, as is the case with a new mail message, the initiating agent will create another new object and select a new, possibly overlapping, set of replicas. This helps to ensure that the degree of replication remains high even in the presence of a failed coordinator. (In the current implementation, if a peer fails during replication, the initiating agent does not select an alternative replica, forcing the remote client to restart the entire session.) The coordinators and participants force their update log to disk before applying the update to ensure that the replicas remain consistent. As an optimization, a replica receiving an update message for which it is the only remaining node need not force its log before applying the update. This is because the other replicas are already up to date, so the sole remaining node will never have to make them current for this update. In practice, this means that only the coordinator forces its log for two-way replication.

Should the coordinator fail after responding to the initiating target but before the update is applied to all replicas, any remaining replica can become the coordinator and bring others up to date. Multiple replicas can become the coordinator in such case, since replicas can discard duplicate updates by comparing timestamps.

In the absence of node failures, the update log remains relatively small for two reasons. First, the log never contains more than one update to the same object. Second, updates are propagated as quickly as they are logged and are deleted as soon as all replicas acknowledge. Timely propagation also narrows the window during which an inconsistency could be perceived.

When a node fails for a long time, the update logs of other nodes could grow indefinitely. To prevent this, updates remain in the update log for at most a week. If a node is restored after that time, it must reenter the Porcupine cluster as a "new" node, rather than as a recovering one. A node renews itself by deleting all of its hard state before rejoining the system.

4.5 Summary

Porcupine's replication scheme provides high availability through the use of consistency semantics that are weaker than strict single-copy consistency, but strong enough to service Internet clients using non-transactional protocols. Inconsistencies, when they occur, are short lived (the update propagation latency between functioning hosts) and, by Internet standards, unexceptional.

5 Dynamic load balancing

Porcupine uses dynamic load balancing to distribute the workload across nodes in the cluster in order to maximize throughput. As mentioned, Porcupine clients select an initial contact node either to deliver or to retrieve mail. That contact node then uses the system's load-balancing services to select the "best" set of nodes for servicing the connection.

In developing the system's load balancer, we had several goals. First, it needed to be fine-grained, making good decisions at the granularity of message delivery. Second, it needed to support a heterogeneous cluster, since not all the nodes are of equivalent power. Third, it had to be automatic and not require the use of any "magic constants," thresholds, or tuning parameters that would need to be manually adjusted as the system evolved. Fourth, with throughput as the primary goal, it needed to resolve the tension between load and affinity. Specifically, in order to best balance load, messages should be stored on idle nodes. However, it is less expensive to store (and retrieve) a message on nodes that already contain mail for the message's recipient. Such *affinity-based scheduling* reduces the amount of memory needed to store fragment lists, increases the sequentiality of disk accesses, and decreases the number of inter-node RPCs required to read, write, or delete a message.

In Porcupine, delivery and retrieval proxies make load-balancing decisions. There is no centralized load-balancing node service; instead, each node keeps track of the load on other nodes and makes decisions independently.

Load information is collected in the same ways we collect liveness information (Section 3.1): (1) as a side-effect of RPC operations (i.e., each RPC request or reply packet contains the load information of the sender), and (2) through a virtual ring in which load information is aggregated in a message passed along the ring. The first approach gives a timely but possibly narrow view of the system's load. The second approach ensures that every node eventually discovers the load from every other node.

The load on a node has two components: a boolean, which indicates whether or not the disk is full, and an integer, which is the number of pending remote procedure calls that might require a disk access. A node with a full disk is always considered "very loaded" and is used only for operations that read or delete existing messages. After some experimentation, we found that it was best to exclude diskless operations from the load to keep it from becoming stale too quickly. Because disk operations are so slow, a node with many pending disk operations is likely to stay loaded for some time.

A delivery proxy that uses load information alone to select the best node(s) on which to store a message will tend to distribute a user's mailbox across many nodes. As a result, this broad distribution can actually reduce overall system throughput for the reasons mentioned earlier. Consequently, we define for each user a *spread*; the spread is a soft upper bound on the number of different nodes on which a given user's mail should be stored. The bound is soft to permit the delivery agent to violate the spread if one of the nodes storing a user's mail is not responding.

As shown in Section 6, the use of a spread-limiting load balancer has a substantial effect on system throughput even with a relatively narrow spread. The benefit is that a given user's mail will be found on relatively few nodes, but those nodes can change entirely each time the user retrieves and deletes mail from the server.

6 System evaluation

This section presents measurements from the Porcupine prototype running synthetic workloads on a 30-node cluster. We characterize the system's scalability as a function of its size in terms of the three key requirements:

- **Performance.** We show that the system performs well on a single node and scales linearly with additional nodes. We also show that the system outperforms a statically partitioned configuration consisting of a cluster of standard SMTP and POP servers with fixed user mapping.
- **Availability.** We demonstrate that replication and reconfiguration have low cost.
- **Manageability.** We show that the system responds automatically and rapidly to node failure and recovery, while continuing to provide good performance. We also show that incremental hardware improvements can automatically result in system-wide performance improvements. Lastly, we show that automatic dynamic load balancing efficiently handles highly skewed workloads.

6.1 Platform and workload

The Porcupine system runs on Linux-based PCs with all system services on a node executing as part of a multi-threaded process. For the measurements in this paper, we ran on a cluster of thirty nodes connected by 1Gb/second Ethernet hubs. As would be expected in any large cluster, our system contains several different hardware configurations: six 200MHz machines with 64MB of memory and 4GB SCSI disks, eight 300 MHz machines with 128MB of memory and 4GB IDE disks, and sixteen 350 MHz machines with 128MB of memory and 8GB IDE disks.

Some key attributes of the system's implementation follow:

- The system runs on Linux 2.2.7 and uses the ext2 file system for storage [25].
- The system consists of fourteen major components written in C++. The total system size is about forty-one thousand lines of code, yielding a 1MB executable.
- A mailbox fragment is stored in two files, regardless of the number of messages contained within. One file contains the message bodies, and the other contains message index information.
- The size of the user map is 256 entries.
- The mailbox fragment files are grouped and stored in directories corresponding to the hash of user names (e.g., if Ann's hash value is 9, then her fragment files are `spool/9/ann` and `spool/9/ann.idx`). This design allows discovery of mailbox fragments belonging to a particular hash bucket – a critical operation during membership reconfiguration – to be performed by a single directory scan.
- Most of a node's memory is consumed by the soft user profile state. In the current implementation, each user entry takes 76 bytes plus 44 bytes per mailbox fragment. For example, in a system with ten million users running on 30 nodes, about 50 MB/node would be devoted to user soft state.

We developed a synthetic workload to evaluate Porcupine because users at our site do not receive enough email to drive the system into an overload condition. We did, however, design the workload generator to model the traffic pattern we have observed on our departmental mail servers. Specifically, we model a mean message size of 4.7KB, with a fairly fat tail up to about 1MB. Mail delivery (SMTP) accounts for about 90% of the transactions, with mail retrieval (POP) accounting for about 10%. Each SMTP session sends a message to a user chosen from a population according to a Zipf distribution with $\alpha = 1.3$, unless otherwise noted in the text.

For purposes of comparison, we also measure a tightly configured conventional mail system in which users and services are statically partitioned across the nodes in the cluster. In this configuration, we run SMTP/POP redirector nodes at the front end. At the back end, we run modified versions of the widely used Sendmail-8.9.3 and ids-popd-0.23 servers. The front-end nodes accept SMTP and POP requests and route them to back-end nodes by way of a hash on the user name. To keep the front ends from becoming a bottleneck, we determined empirically that we need to run one front end for every fifteen back ends. The tables and graphs that follow include the front ends in our count of the system size. Based on *a priori* knowledge of the workload, we defined the hash function to distribute users perfectly across the back-end nodes. To further optimize the configuration, we disabled all security checks, including user authentication, client domain name lookup, and system log auditing.

For both Porcupine and the conventional system, we defined a user population with size equal to 160,000 times the number of nodes in the cluster (or about 5 million users for the 30-node configuration). Nevertheless, since the database is distributed in Porcupine, and no authentication is performed for the conventional platform, the size of the user base is nearly irrelevant to the measurements. Each POP session selects a user according to the same Zipf distribution, collects and then deletes all messages awaiting the user. In the Porcupine configuration, the generator initiates a connection with a Porcupine node selected at random from all the nodes. In the conventional configuration, the generator selects a node at random from the front-end nodes. By default, the load generator attempts to saturate the cluster by probing for the maximum throughput, increasing the number of outstanding requests until at least 10% of them fail to complete within two seconds. At that point, the generator reduces the request rate and resumes probing.

We demonstrate performance by showing the maximum number of messages the system receives per second. Only message deliveries are counted, although message retrievals occur as part of the workload. Thus, this figure really reflects the number of messages the cluster can receive, write, read, and delete per second. The error margin is smaller than 5%, with 95% confidence for all values presented in the following sections.

6.2 Scalability and performance

Figure 3 shows the performance of the system as a function of cluster size. The graph shows four different configurations: without message replication, with message replication, with message replication using NVRAM for the logs, and finally for the conventional configuration of sendmail+popd. Although neither replicates, the Porcupine no-replication case outperforms and outscales conventional sendmail. The difference is primarily due to the conventional system's use of temporary files, excessive process forking, and the use of lock-files. With some effort, we believe the conventional system could be made to scale as well as Porcupine without replication. However, the systems would not be functionally identical, because Porcupine continues to deliver service to all users even when some nodes are down.

For replication, the performance of Porcupine scales linearly when each incoming message is replicated on two nodes. There is a substantial slowdown relative to the non-replicated case, because replication increases the number of

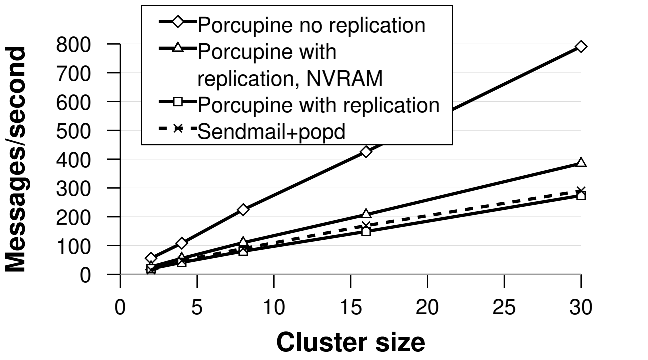

Figure 3. Throughput scales with the number of hosts. This graph shows how Porcupine and the sendmail-based system scale with respect to cluster size.

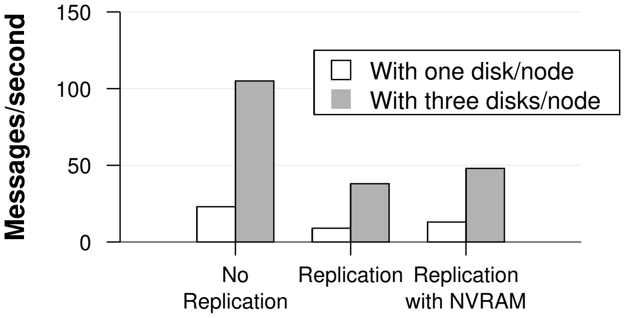

Figure 4. Summary of single-node throughput in a variety of configurations.

synchronous disk writes three-fold: once for each replica and once to update the coordinator's log. Even worse, in this hardware configuration the log and the mailbox fragments share the same disk on each node.

One way to improve the performance of replication is to use non-volatile RAM for the log. Since updates are typically retired, most of the writes to NVRAM need never go to disk and can execute at memory speeds. Although our machines do not have NVRAM installed, we can simulate NVRAM simply by keeping the log in standard memory. As shown in Figure 3, NVRAM improves throughput; however, throughput is still about half that of the non-replicated case, because the system must do twice as many disk operations per message.

Resource	No replication	With replication
CPU utilization	15%	12%
Disk utilization	75%	75%
Network send	2.5Mb/second	1.7Mb/second
Network recv	2.6Mb/second	1.7Mb/second

Table 1. Resource consumption on a single node with one disk.

Table 1 shows the CPU, disk, and network load incurred by a single 350Mhz Porcupine node running at peak throughput. For this configuration, the table indicates that the disk is the primary impediment to single-node performance.

To demonstrate this, we made measurements on clusters with one and two nodes with increased I/O capacity. A single 300MHz node with one IDE disk and two SCSI disks delivered a throughput of 105 messages/second, as opposed to about 23 messages/second with just one disk. We then configured a two node cluster, each with one IDE disk and two SCSI disks. The machines were each able to handle 38 messages/second (48 assuming NVRAM). These results (normalized to single-node throughput) are summarized in Figure 4.

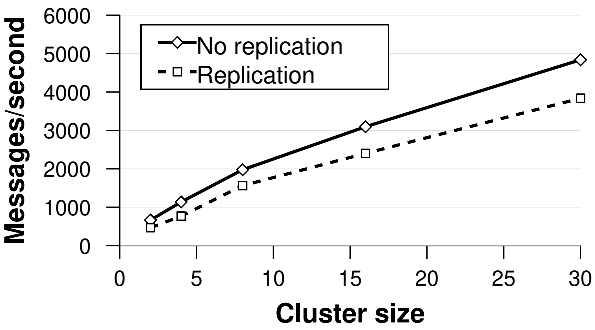

Figure 5. Throughput of the system configured with infinitely fast disks.

Lastly, we measured a cluster in which disks were assumed to be infinitely fast. In this case the system does not store messages on disk but only records their digests in main memory. Figure 5 shows that the simulated system without the disk bottleneck achieves a six-fold improvement over the measured system. At this point, the CPU becomes the bottleneck. Thus Porcupine with replication performs comparatively better than on the real system.

With balanced nodes, the network clearly becomes the bottleneck. In the non-replicated case, each message travels the network four times ((1) Internet to delivery agent (2) to mailbox manager (3) to retrieval agent (4) to Internet). At an average message size of 4.7KB, a 1Gb/second network can then handle about 6500 messages/second. With a single "disk loaded" node able to handle 105 messages/second, roughly 62 nodes will saturate the network as they process 562 million messages/day. With messages replicated on two nodes, the same network can handle about 20% fewer messages (as the message must be copied one additional time to the replica), which is about 5200 messages/second, or about 450 million messages/day. Using the throughput numbers measured with the faster disks, this level of performance can be achieved with 108 NVRAM nodes, or about 137 nodes without NVRAM. More messages can be handled only by

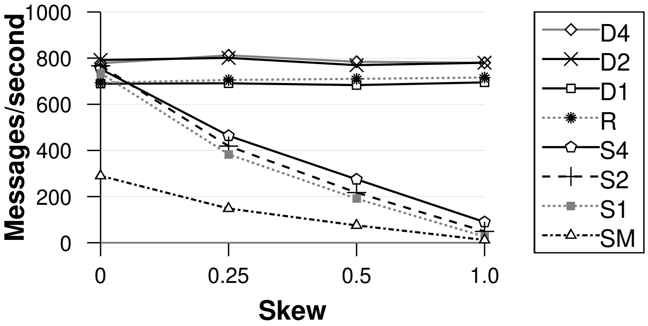

Figure 6. Non-replicated throughput on a 30-node system.

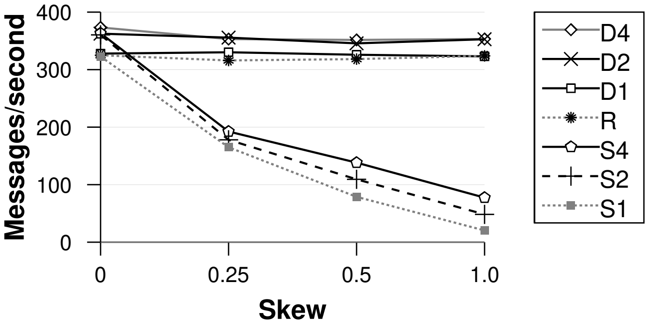

Figure 7. Replicated throughput on a 30-node system.

increasing the aggregate network bandwidth. We address this issue further in Section 7.

6.3 Load balancing

The previous section demonstrated Porcupine's performance assuming a uniform workload distribution and homogeneous node performance. In practice, though, workloads are not uniformly distributed and the speeds of CPUs and disks on nodes differ. This can create substantial management challenges for system administrators when they must reconfigure the system manually to adapt to the load and configuration imbalance.

This section shows how Porcupine automatically handles workload skew and heterogeneous cluster configuration.

6.3.1 Adapting to workload skew

Figures 6 and 7 show the impact of Porcupine's dynamic spread-limiting, load-balancing strategy on throughput as a function of workload skew for our 30-node configuration (all with a single slow disk). Both the non-replicated and replicated cases are shown. Skew along the x-axis reflects the inherent degree of balance in the incoming workload. When the skew equals zero, recipients are chosen so that the hash distributes uniformly across all buckets. When the skew is one, the recipients are chosen so that they all hash into a single user map bucket, corresponding to a highly imbalanced workload.

The graphs compare random, static, and dynamic load balancing policies. The random policy, labeled R on the graph, simply selects a host at random to store each message received; it has the effect of smoothing out any non-uniformity in the distribution. The static spread policy, shown by the lines labeled S1, S2, and S4, selects a node based on a hash of the user name spread over 1, 2 or 4 nodes, respectively. The dynamic spread policy – the one used in Porcupine – selects from those nodes already storing mailbox fragments for the recipient. It is shown as D1, D2 and D4 on the graph. Again, the spread value (1, 2, 4) controls the maximum number of nodes (in the absence of failure) that store a single user's mail. On message receipt, if the size of the current fragment list for the recipient is smaller than the maximum spread, Porcupine increases the spread by choosing an additional node selected randomly from the cluster.

Static spread manages affinity well but can lead to an imbalanced load when activity is concentrated on just a few nodes. Indeed, a static spread of one corresponds to our sendmail+popd configuration in which users are statically partitioned to different machines. This effect is shown as well on the graph for the conventional sendmail+pop configuration (SM on Figure 6). In contrast, the dynamic spread policy continually monitors load and adjusts the distribution of mail over the available machines, even when spread is one. In this case, a new mailbox manager is chosen for a user each time his/her mailbox is emptied, allowing the system to repair affinity-driven imbalances as necessary.

The graphs show that random and dynamic policies are insensitive to workload skew, whereas static policies do poorly unless the workload is evenly distributed. Random performs worse than dynamic because of its inability to balance load and its tendency to spread a user's mail across many machines.

Among the static policies, those with larger spread sizes perform better under a skewed workload, since they can utilize a larger number of machines for mail storage. Under uniform workload, however, the smaller spread sizes perform better since they respect affinity. The key exception is the difference between spread=1 and spread=2. At spread=1, the system is unable to balance load. At spread=2, load is balanced and throughput improves. Widening the spread beyond two improves balance slightly, but not substantially. The reason for this has been demonstrated previously [10] and is as follows: in any system where the likelihood that a host is overloaded is p, then selecting the least loaded from a spread of s hosts will yield a placement decision on a loaded host with probability p^s. Thus, the chance of making a good decision (avoiding an overloaded host) improves exponentially with the spread. In a nearly perfectly-balanced system, p is small, so a small s yields good choices.

The effect of the loss of affinity with larger spread sizes is not pronounced in the Linux ext2 file system because it creates or deletes files without synchronous directory modification [25]. On other operating systems, load balancing

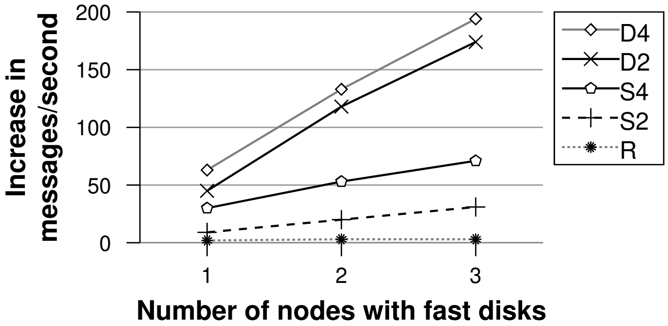

Figure 8. Performance improvement on a 30-node Porcupine cluster without replication when disks are added to a small number of nodes.

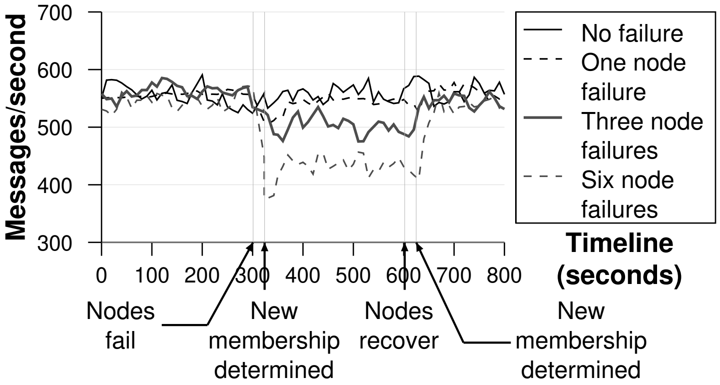

Figure 10. Reconfiguration timeline without replication.

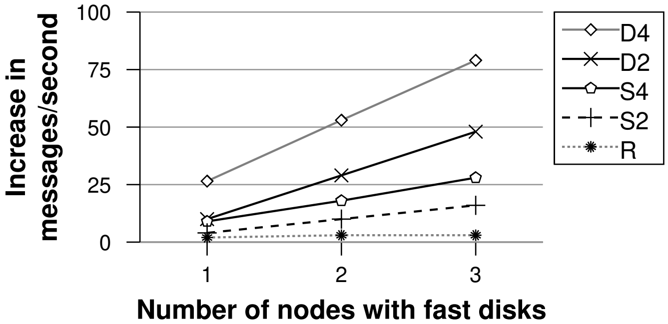

Figure 9. Performance improvement on a 30-node Porcupine cluster with replication when disks are added to a small number of nodes.

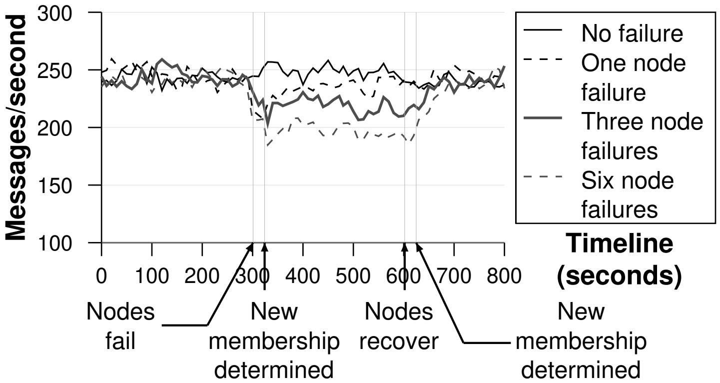

Figure 11. Reconfiguration timeline with replication.

policies with larger spread sizes will be penalized more by increased frequency of directory operations.

6.3.2 *Adapting to heterogeneous configurations*

As mentioned in the previous section, the easiest way to improve throughput in our configuration is to increase the system's disk I/O capacity. This can be done by adding more machines or by adding more or faster disks to a few machines. In a statically partitioned system, it is necessary to upgrade the disks on all machines to ensure a balanced performance improvement. In contrast, because of Porcupine's functional homogeneity and automatic load balancing, we can improve the system's *overall* throughput for all users simply by improving the throughput on a few machines. The system will automatically find and exploit the new resources.

Figures 8 and 9 show the absolute performance improvement of the 30-node configuration when adding two fast SCSI disks to each of one, two, and three of the 300Mhz nodes, with and without replication. The improvement for Porcupine shows that the dynamic load balancing mechanism can fully utilize the added capacity. Here, spread=4 slightly outperforms spread=2, because the former policy is more likely to include the faster nodes in the spread. When a few nodes are many times faster than the rest, as is the case with our setting, the spread size needs to be increased. On the other hand, as described in Section 5, larger spread sizes tend to reduce the system efficiency. Thus, spread size is one parameter that needs to be revisited as the system becomes more heterogeneous.

In contrast, the statically partitioned and random message distribution policies demonstrate little improvement with the additional disks. This is because their assignment improves performance for only a subset of the users.

6.4 Failure recovery

As described previously, Porcupine automatically reconfigures whenever nodes fail or restart. Figures 10 and 11 depict an annotated timeline of events that occur during the failure and recovery of 1, 3, and 6 nodes in a 30-node system without and with replication. Both figures show the same behavior. Nodes fail and throughput drops dramatically as two things occur. First, the system goes through its reconfiguration protocol, increasing its load. Next, during the reconfiguration, SMTP and POP sessions that involve the failed node are aborted. After ten seconds, the system determines the new membership, and throughput increases as the remaining nodes take over for the failed ones. The failed nodes recover 300 seconds later and rejoin the cluster, at which time throughput starts to rise. For the non-replicated case, throughput increases back to the pre-failure level almost immediately. With replication, throughput rises slowly as the failed nodes reconcile while concurrently serving new requests.

7 Limitations and future work

Porcupine's architecture and implementation have been designed to run well in very large clusters. There are, however, some aspects of its design and the environment in which it is deployed that may need to be rethought as the system grows to larger configurations.

First, Porcupine's communication patterns are flat, with every node as likely to talk to every other node. A 1Gb/second heavily switched network should be able to serve about 6500 messages/second (or 560 million messages/day) without replication. With replication, the network can handle 5200 messages/second, or 450 million messages/day. Beyond that, faster networks or more network-topology-aware load balancing strategies will be required to continue scaling.

Our membership protocol may also require adjustments as the system grows. Presently, the membership protocol has the coordinator receiving acknowledgment packets from all participants in a very short period of time. Although participants currently insert a randomized delay before responding to smooth out packet bursts at the receiver, we still need to evaluate whether this works well at very large scale. In other work, we are experimenting with a hierarchical membership protocol that eliminates this problem. In time, we may use this to replace Porcupine's current protocol.

Our strategy for reconstructing user profile soft state may also need to be revisited for systems in which a single user manager manages millions of users (many users, few machines). Rather than transferring the user profile soft state in bulk, as we do now, we could modify the system to fetch profile entries on use and cache them. This would reduce node recovery time (possibly at the expense of making user lookups slower, however).

8 Related work

The prototypical distributed mail service was Grapevine [23], a wide-area service intended to support about ten thousand users. Grapevine users were statically assigned to (user-visible) registries. The system scaled through the addition of new registries having sufficient power to handle their populations. Nevertheless, Grapevine's administrators were often challenged to balance users across mail servers. In contrast, Porcupine implements a flat name space managed by a single cluster and automatically balances load. Grapevine provided a replicated user database based on optimistic replication, but it did not replicate mail messages. Porcupine uses optimistic replication for both mail and the user database.

As described earlier, contemporary email cluster systems deploy many storage nodes and partition the user population statically among them, either using a distributed file system [4] or protocol redirectors [9]. As we demonstrate in this paper, this static approach is difficult to manage and scale and has limited fault tolerance.

Numerous fault-tolerant, clustered-computing products have been described in the past (e.g., [13, 26]). These clusters are often designed specifically for database fail-over, have limited scalability, and require proprietary hardware or software. Unlike these systems, Porcupine's goal is to scale to hundreds or thousands of nodes using standard off-the-shelf hardware and software.

Fox et al. [11] describe an infrastructure for building scalable network services based on cluster computing. They describe a data semantics called BASE (Basically Available, Soft-state, Eventual consistency) that offers advantages for the web-search and document-filtering applications they present. Our work shares many of their goals – building scalable Internet services with a weaker semantics than traditional databases. As in the Fox work, we observe that ACID semantics [12] may be too strong for our application and define a data model that is equal to the non-transactional model used by the system's clients. However, unlike BASE, our semantics support write-intensive applications requiring persistent data. Our services are also distributed and replicated uniformly across all nodes for greater scalability. They are not being statically partitioned by function.

A large body of work exists on the general topic of load sharing, but this work has been targeted mainly at systems with long-running, CPU-bound tasks. For example, Eager et al. [10] show that effective load sharing can be accomplished with simple adaptive algorithms that use random probes to determine load. In the context of clusters and the Web, several commercial products automatically distribute requests to cluster nodes, typically using a form of round-robin dispatching [6]. In [8, 19], the authors propose a class of load distribution algorithms using a random spread of nodes and a load-based selection from the spread. Their results show that a spread of two is optimal for a wide variety of situations in a homogeneous cluster. Pai et al. [21] describe a "locality-aware request distribution" mechanism for cluster-based services. A front-end node analyzes the request content and attempts to direct requests so as to optimize the use of buffer cache in back-end nodes, while also balancing load. Porcupine uses load information, in part, to distribute incoming mail traffic to cluster nodes. However, unlike previous load-balancing studies that assumed complete independence of incoming tasks, we also balance the write traffic, taking message affinity into consideration.

The replication mechanism used in Porcupine can be viewed as a variation of optimistic replication schemes [1, 27], in which timestamped updates are pushed to peer nodes to support multi-master replication. Porcupine's total object update property allows it to use a single timestamp per object, instead of timestamp matrices, to order updates. In addition, since updates are idempotent, Porcupine can retire updates more aggressively. These differences make Porcupine's approach to replication simpler and more efficient at scale.

Several file systems have scalability and fault tolerance goals that are similar to Porcupine's [2, 15, 16, 24]. Unlike these systems, Porcupine uses the semantics of the various data structures it maintains to exploit their special properties in order to increase performance or decrease complexity.

9 Conclusions

We have described the architecture, implementation, and performance of the Porcupine scalable mail server. We have shown that Porcupine meets its three primary goals:

Manageability. Porcupine automatically adapts to changes in configuration and workload. Porcupine masks heterogeneity, providing for seamless system growth over time using latest-technology components.

Availability. Porcupine continues to deliver service to its clients, even in the presence of failures. System software detects and recovers automatically from failures and integrates recovering nodes.

Performance. Porcupine's single-node performance is competitive with other systems, and its throughput scales linearly with the number of nodes. Our experiments show that the system can find and exploit added resources for its benefit.

Porcupine achieves these goals by combining four key architectural techniques: functional homogeneity, automatic reconfiguration, dynamic transaction scheduling, and replication. In the future, we hope to construct, deploy and evaluate configurations larger and more powerful than the ones described in this paper.

Acknowledgements

We would like to thank Eric Hoffman, Bertil Folliot, David Becker, and other members of the Porcupine project for the valuable discussions and comments on the Porcupine design.

This work is supported by DARPA Grant F30602-97-2-0226 and by National Science Foundation Grant # EIA-9870740.

References

[1] Divyakant Agrawal, Amr El Abbadi, and R. C. Steike. Epidemic algorithms in replicated databases. In *Proceedings of the 16th Symposium on Principles of Database Systems*, pages 161–172, Montreal, Canada, May 1997.

[2] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Compaq Systems Research Center, Palo Alto, CA, September 1993.

[3] Thomas P. Brisco. RFC1794: DNS support for load balancing, April 1995. http://www.cis.ohio-state.edu/htbin/rfc/rfc1794.html.

[4] Nick Christenson, Tim Bosserman, and David Beckemeyer. A highly scalable electronic mail service using open systems. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[5] Flaviu Christian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995.

[6] Cisco Systems. Local director. http://www.cisco.com/-warp/public/751/lodir/index.html.

[7] Marc Crispin. RFC2060: Internet message access protocol version 4 rev 1, December 1996. http://www.cis.ohio-state.edu/htbin/rfc/rfc2060.html.

[8] Michael Dahlin. Interpreting stale load information. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Austin, TX, May 1999.

[9] James Deroest. Clusters help allocate computing resources. http://www.washington.edu/tech_home/-windows/issue18/clusters.html, 1996.

[10] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986.

[11] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *16th Symposium on Operating Systems Principles*, pages 78–91, St. Malo, France, October 1997.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[13] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Trans. on Computer Systems*, 4(2):130–146, 1986.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, October 1996.

[16] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[17] David L. Mills. RFC1305: Network time protocol (version 3), March 1992. http://www.cis.ohio-state.edu/htbin/rfc/rfc1305.html.

[18] David L. Mills. Improved algorithms for synchronizing computer network clocks. In *ACM SIGCOMM*, pages 317–327, London, UK, September 1994.

[19] Michael Mitzenmacher. How useful is old information? Technical Report 98-002, Compaq Systems Research Center, Palo Alto, CA, February 1998.

[20] John G. Myers and Marshall T. Rose. RFC1939: Post office protocol version 3, May 1996. http://www.cis.ohio-state.edu/htbin/rfc/rfc1939.html.

[21] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 206–216, San Jose, CA, October 1998.

[22] Jonathan Postel. RFC821: Simple mail transfer protocol, August 1982. http://www.cis.ohio-state.edu/htbin/rfc/rfc821.html.

[23] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[24] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: A scalable distributed file system. In *16th Symposium on Operating Systems Principles*, pages 224–237, St. Malo, France, October 1997.

[25] Theodore Ts'o. Ext2 home page, 1999. http://web.-mit.edu/tytso/www/linux/ext2.html.

[26] Werner Vogels, Dan Dumitriu, Ken Birman, Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, and Jim Gray. The design and architecture of the Microsoft cluster service. In *28th International Symposium on Fault-Tolerant Computing*, pages 422–431, Munich, Germany, June 1998.

[27] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the 3rd Symposium on Principles of Distributed Computing*, pages 233–242, Vancouver, Canada, August 1984.