Experiences in Building and Operating ePOST, a Reliable Peer-to-Peer Application

Alan Mislove[†‡] Ansley Post[†‡] Andreas Haeberlen[†‡] Peter Druschel[†]

{amislove,abpost,ahae,druschel}@rice.epostmail.org

[†]Max Planck Institute for Software Systems, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.
[‡]Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA.

ABSTRACT

Peer-to-peer (p2p) technology can potentially be used to build highly reliable applications without a single point of failure. However, most of the existing applications, such as file sharing or web caching, have only moderate reliability demands. Without a challenging proving ground, it remains unclear whether the full potential of p2p systems can be realized.

To provide such a proving ground, we have designed, deployed and operated a p2p-based email system. We chose email because users depend on it for their daily work and therefore place high demands on the availability and reliability of the service, as well as the durability, integrity, authenticity and privacy of their email. Our system, ePOST, has been actively used by a small group of participants for over two years.

In this paper, we report the problems and pitfalls we encountered in this process. We were able to address some of them by applying known principles of system design, while others turned out to be novel and fundamental, requiring us to devise new solutions. Our findings can be used to guide the design of future reliable p2p systems and provide interesting new directions for future research.

Categories and Subject Descriptors

C.2.4 [**Computer-Communications networks**]: Distributed Systems—*Peer-to-peer applications*; H.4.3 [**Information Systems**]: Communications Applications—*Electronic mail*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.0 [**Software**]: General—*Peer-to-peer systems*

General Terms

Algorithms, Design, Measurement, Reliability, Experimentation

EuroSys'06, April 18–21, 2006, Leuven, Belgium.

Keywords

Peer-to-peer, electronic mail, reliability, decentralized systems

1. INTRODUCTION

Decentralized, cooperative systems (also known as p2p systems) have received much attention in recent years because they promise robustness to a wide range of workloads and failures, are highly scalable with respect to nodes, users, and data, and enable cooperative sharing of otherwise underutilized resources. The technology was first adopted for use in file sharing and content distribution systems like KaZaA, Gnutella, eDonkey and BitTorrent [3, 20, 29, 40], which now enjoy widespread popular use. While these systems have grown to a considerable scale, they have only very modest reliability demands. In general, their users seem to tolerate slow downloads, repeated failures, and even corrupted data, as long as they get the content eventually.

Prototypes of more reliable p2p systems, including content distribution [3, 10, 16, 55], information retrieval [24, 52, 53, 56], and name resolution [41, 54], have mostly focused on availability, scalability, and resilience to overload. Cooperative storage systems [5, 12, 30] additionally require durability and consistency, but to our knowledge, these systems have not yet been developed and deployed for production use. Thus, it remains unclear whether the full reliability potential of p2p systems can be realized in practice.

In an effort to address this question, we have designed, deployed and operated ePOST, a serverless email system. We have chosen email not because it is a killer application for p2p (it isn't). Rather, we chose it as a proving ground because it is well understood, widely used, and has stringent reliability requirements. Many users depend on email for their daily work and therefore place high demands on the availability of the service, as well as the durability, integrity, authenticity, and privacy of their email. Hence, by demonstrating that users can rely on a p2p-based email system as their primary email service, we believe that we have taken a major step towards establishing p2p technology as a viable platform for applications with high reliability demands.

As an additional benefit, the basic elements of an email system — event notification, durable storage of mostly immutable data, and some per-user mutable metadata — are characteristic of many collaborative applications. Demonstrating that highly reliable email services can be provided using p2p technology suggests that the technology can sup-

port other collaborative applications and even enable new applications.

In this paper, we report on our experience in designing, deploying, and operating ePOST. Since ePOST users expect the system to perform equally well or better than a conventional server-based system, our challenge was to provide an acceptable user experience from a system built upon unreliable machines and networks. Surprisingly, the problems we encountered were generally the result of incorrect assumptions made in the design of underlying component technologies like the structured overlay and the distributed hash table (DHT) used by ePOST. Seemingly unrelated user-perceivable errors and inconsistencies in ePOST operation often led us to discover fundamental problems with these components.

We describe in some detail the challenges we encountered in providing users with an acceptable service. This means providing strong data durability, high availability, and a mostly consistent view of the user's email folders despite correlated node failures, network partitions, network anomalies, and node churn. In some cases, we were able to address problems by applying known principles of system design; other issues turned out to be novel and fundamental, requiring us to devise new solutions. After a partial re-design to address initial problems, ePOST has supported a small user base, including the authors, who have relied on the system as their primary email service for over two years.

Overall, this paper makes four contributions: (*i*) We report on our experiences in designing and operating a cooperative, serverless email service that supports real users. (*ii*) We identify two problems, routing anomalies and non-transitivity, that emerged during our use of the system, and we describe and evaluate our solutions for them. (*iii*) We note some pitfalls we encountered that are of interest to designers of similar systems, though we were able to address them using known principles of system design. Finally, (*iv*) we demonstrate that it is possible to support applications with high availability, data durability, and consistency using p2p technology.

The rest of this paper is structured as follows: Section 2 describes related work, and Section 3 presents background on the design and deployment of ePOST. Section 4 describes four general lessons we distilled from our experience building and operating ePOST, and we present and evaluate solutions for several problems we had to solve. Section 5 details some pitfalls we encountered, and Section 6 describes interesting directions for future research. Section 7 presents our conclusions.

2. RELATED WORK

In this section, we present a general overview of related work. More related work is discussed in the context of specific problems and solutions later in the paper.

The most visible type of deployed peer-to-peer systems are file sharing systems such as eDonkey2000, KaZaA, Gnutella, and BitTorrent [3,15,20,29,31]. Since users of these systems are usually downloading media content, they tend not to have high expectations about system performance or data integrity (e.g., they will commonly tolerate slow downloads or having to redownload content).

Skype [21] is a popular p2p system providing voice-over-IP (VoIP) services. To provide a satisfactory user experience, Skype must focus on service availability and timely delivery of streaming data, while ePOST's focus is on availability, durability and consistency.

End System Multicast (ESM) [10, 25, 49] is a deployed system used to broadcast streaming video using cooperating end hosts to disseminate the data. ESM, like VoIP, is used for streaming media data, which has high requirements of timeliness but not reliability or durability. This makes the challenges it faces very different, though no less significant, from those addressed in ePOST.

OpenHash [28, 45] is an publicly open DHT system that provides a public hashtable interface allowing users to store and retrieve data. OpenHash is a public service infrastructure for others to build applications and therefore does not emphasize data durability and consistency to the extent required for running ePOST.

CoDoNS and CoDNS [41, 44] are decentralized systems that seek to replace or augment the existing DNS infrastructure to improve its reliability, availability, and performance. Since DNS entries are infrequently updated, consistency problems are not likely to be visible to the user. Coral [11, 16] is a cooperative web cache, which is centrally administered on the PlanetLab [43] network and does not allow untrusted nodes to join the network. Coral-CDN forms a cache of data that can be found on the Internet, and as such all data contained can be recovered from the source.

The Resilient Overlay Networks (RON) [1] project has deployed a network of nodes that can be used to increase the reliability of routing in the Internet. It augments the service traditionally provided by the Internet and does not store data on behalf of users.

3. BACKGROUND

In this section, we give a brief overview of the ePOST system, and we describe our current deployment. The initial design of ePOST was previously sketched in [35].

3.1 Overview

ePOST provides secure, decentralized email services to its users. Each user runs the ePOST software on her local machine and contributes some of that machine's CPU, storage and network bandwidth to the system. Users' email messages and metadata are stored cooperatively and redundantly in a DHT backed by the participants' disks. All data is signed for authenticity and encrypted for confidentiality before it is inserted into the cooperative storage.

In order to deliver an email in ePOST, the sender inserts the email and a special delivery request into the cooperative store. Nodes that store such a delivery request subscribe to a multicast group under the recipient's identifier. When the recipient's node is online, it periodically sends a notification to this group, causing any pending emails to be delivered to it.

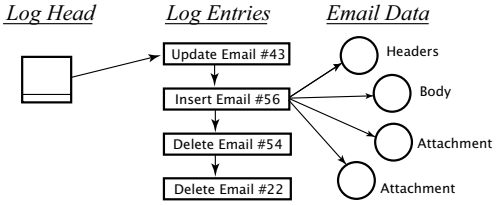

Figure 1: Log structure in ePOST.

Every ePOST user maintains a single-writer log representing the user's view of the data in the system, i.e., her mailbox hierarchy and contents. Each user has a mutable *log head* which points to the most recent entry in the log, shown in Figure 1. When an email is inserted, deleted, moved, or otherwise altered, a log entry is inserted into the log, and the log head is updated to point to this entry, thus updating the status of her mailbox. More details, including several optimizations, can be found in [33, 35].

3.2 Components

ePOST is built upon several known decentralized components. We provide just enough detail about these components to make the paper self-contained. The choice of these components is largely irrelevant to the issues described in this paper. That is, we could have built ePOST using similar components like Chord [50] without affecting the experiences, problems, or solutions described in this paper.

Pastry [46] is a structured p2p overlay network; it provides the foundation on which ePOST is built. In Pastry, every node and every object is assigned a unique identifier randomly chosen from a 160-bit identifier space, referred to as a *nodeId* and *key*, respectively. Given a message and a key, Pastry can route the message to the live node whose nodeId is numerically closest to the key, within an expected *log N* forwarding hops, where N is the number of participating nodes. To do this, Pastry contains two sets of pointers to other overlay nodes called a *routing table* and a *leaf set*. The leaf set contains the nearest neighboring nodes in the identifier space and is used to determine which node is responsible for a given key. Since node identifier assignment is random, each leaf set represents a random sample of the Pastry overlay membership. The routing table exists solely to improve routing efficiency; it refers to additional overlay nodes in specific locations around the identifier space. To join the ring, a Pastry node contacts one of a set of well-known *bootstrap nodes*. These nodes are ordinary Pastry nodes that were selected, for instance, because they have proved to be highly available.

Multi-ring support [34] for Pastry provides a hierarchy of logically independent organizational overlays ('rings'), which are seamlessly connected via a global ring. By setting up its own subring, an organization can ensure that its email data is stored only on machines under its control. Using multiple rings also improves performance in the common case, since most overlay traffic stays within the same subring. Additionally, the hierarchy ensures that each organization has control over its own ring, and that any malicious participants can be tracked down and punished.

Scribe [8] is a scalable group communication system built on top of Pastry; ePOST uses it to multicast the notification messages mentioned earlier. Each Scribe group has a *groupId*, which serves as the address of the group. The nodes subscribed to each group form a tree, consisting of the union of Pastry routes from all group members to the node with nodeId numerically closest to the groupId.

PAST [47] provides a storage service on top of Pastry, which is organized as a DHT. ePOST uses it to keep persistent state such as emails, logs, and delivery requests. When an object is stored in PAST, it is assigned a key; PAST then stores copies of the object at the k live nodes whose nodeIds are numerically closest to the object's key. When one of these copies is lost, e.g., because of a node failure, PAST creates another copy to replace it.

3.3 Implementation

To use ePOST, a new user visits `www.epostmail.org` and chooses her ePOST email address. She then downloads the latest version of the software and a certificate, which links her new email address to her public key. After installing these on her computer, her ePOST node is ready for use.

The ePOST software running on the user's node acts both as an overlay participant and as a local email server, or *proxy*, for the user. It supports common protocols (SMTP, IMAP, POP3, SSL), so the user can use existing email clients, like Mozilla Thunderbird, Microsoft Outlook, Apple Mail, and pine. All ePOST communication is encrypted and authenticated, and the user's local proxy is the only node that is required to hold the user's key material.

All data stored and replicated in the DHT is self-authenticating, either via Merkle hash trees [32] (immutable log elements and email data) or signed data (mutable log head). This prevents emails from being read by unauthorized users, and it makes forged or corrupted message and metadata replicas evident. Participating nodes hold certified nodeIds [7], preventing users from selecting their own nodeId and mounting Sybil attacks [14].

3.4 Deployment

ePOST has been under development since early 2003 and in production use since January 2004, so we have more than two years of experience with the system. The authors have used the system as their primary email service for over two years.

At the time of this writing, there are several ePOST rings in operation: An *open* ring is open to individuals from the general public, while an *organizational* ring spanning both Rice University and the Max Planck Institute for Software Systems (MPI-SWS) is limited to members of these organizations. A *global* ring provides seamless connectivity among all the other rings. The ePOST deployment currently consists of 295 nodes overall, most of which are from PlanetLab [43] and form the base of the open ring. In the Rice and MPI-SWS ring, there are 9 users who use ePOST as their primary email system, and 8 users who use their ePOST account as a secondary email account into which they forward a copy of all their email. Since ePOST is a research project, we have not tried to acquire a large user base. This allowed us to quickly respond to problems and minimize user support overhead.

ePOST imposes little overhead on participating nodes. Nodes in the Rice and MPI-SWS ring transmit an average of 8.52 messages per node per minute, resulting in an overall bandwidth usage of 634 bytes per second on each node. The total amount of data stored on disk averaged approximately 1,112 MB per node, which is actually somewhat inflated since a number of our users never delete their mail in ePOST. The memory footprint of the ePOST Java Virtual Machines averaged 132 MB. We also found the availability of ePOST to be high – on a number of occasions, our department mail servers were unavailable, while ePOST users never noticed the outage. Additionally, even after a disastrous 89% correlated failure over the course of a day in the PlanetLab ring, the system automatically recovered and nodes rejoined as soon as they were rebooted. In fact, during this failure, the majority of the email data was still

available, as the DHT replication was able to migrate data to working nodes faster than the node failures occurred.

4. EXPERIENCE: PROBLEMS, SOLUTIONS AND LESSONS

In this section, we report on our experiences in operating ePOST while it was in active use by a small group of participants. Since initially deploying the system, we encountered several challenges not foreseen in the original design. We had to go through a round of redesigns to improve the reliability and data durability of the underlying components in order to provide a satisfactory user experience in ePOST.

The section is structured around four general lessons we distilled from this experience. Each lesson states the initial assumptions and design, describes the problem we encountered, devises a solution and evaluates the resulting improvements. The first three lessons deal with providing users with an acceptable degree of consistency despite the possibility of network partitions, Internet routing anomalies, and churn in the overlay membership. The final lesson deals with providing data durability despite the possibility of correlated failures. It is not surprising that consistency and correlated failures figure prominently in these lessons, since they are among the most important challenges in distributed, and especially decentralized, systems.

4.1 Network partitions

LESSON 1: *A reliable decentralized system must tolerate network partitions.*

The Pastry overlay relies on the underlying physical network to transmit messages between nodes. A partition in the underlying network can cause the overlay to split into more than one connected component. The original design of ePOST and Pastry had no specific provisions for network partitions. This resulted in four consequences for ePOST under a network partition:

- The nodes in each partition perceived the nodes in the other partitions as having failed, and they formed separate rings.
- Mutable data with replicas in different rings potentially diverged.
- In small partitions, insufficient redundancy caused data to be unavailable.
- After the physical network partition healed, there was no guarantee that the multiple rings would eventually reintegrate; if they did, modifications to mutable data were potentially be lost.

In this section, we first report on the frequency and nature of the partitions we observed in our deployment and show that partitions cannot be ignored. Then, we present mechanisms that allow ePOST to tolerate and gracefully recover from partitions. Other proposed systems, e.g., [38, 48], have provisions for detecting network partitions, however, leveraging the properties of ePOST allows us to recover from partitions in an efficient manner.

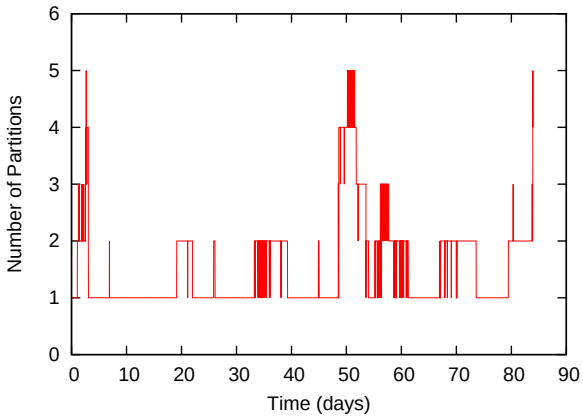

Figure 2: Number of connected components observed over a 85 day period.

4.1.1 Partitions in PlanetLab

Each node in our PlanetLab ePOST deployment periodically records the contents of its Pastry leafset in its log. We combined these logs and counted the number of connected components, i.e., the number of separate rings. Figure 2 shows our results. During 3 months with an average of about 280 nodes, the overlay was partitioned 110 times. These partitions would only heal when either the partitioned nodes restarted or we noticed the problem and manually intervened. Often, the unhealed partitions would last for days, as there were many more nodes than users in the PlanetLab ring, and the partitions were small enough to not cause any data loss in the main partition.

We also gathered statistics on the size of the components. Almost all partitions created one large component, which contained the majority of the nodes, and one or more small components. The average size of the small component was 12.4 nodes, representing 4.4% of the network. This is not surprising, since link failures in the Internet core can usually be routed around and masked. Link failures closer to end hosts, on the other hand, generally result in an group of machines becoming partitioned.

The data clearly shows that network partitions occur frequently in PlanetLab. Our initial ePOST design had no specific provisions to handle partitions, which lead to inconsistencies when the system was operated in PlanetLab.

4.1.2 Operating under a partition

It is well known that in a system where partitions can occur, there is a fundamental tradeoff between availability and consistency [19]. In the context of ePOST, this tradeoff was resolved in favor of availability. This makes sense because the data structures used in ePOST are mostly immutable with the exception of the log heads, and each log head is modified by a single writer only. Hence, inconsistencies can arise only when a user attaches to one partition and then to another, and makes changes in both.

ePOST can still provide useful services even in minority partitions: It can deliver email between users in that partition, and it can store other messages for later delivery. However, if the partition is small, it can offer only limited access to previously stored email data, since some objects are likely to be missing. In Section 4.4, we describe a tech-

nique that can mitigate this problem.

Operating under a partition thus required no changes to ePOST's design. The surviving rings on each side of a partition continue to operate, oblivious of each others' existence.

4.1.3 Overlay reintegration

In order to reconnect multiple disjoint overlays, the system must solve two problems: It must first establish at least one overlay link between nodes in different components, and then form a consistent topology without destabilizing the system. The original ePOST design did neither.

We solved these problems in the following way: Each node sends a join message every T_R seconds to either a random bootstrap node or a leaf set node that it has observed fail recently. It checks whether the resulting leafset is identical to its own. If this is not the case, the other node is in a different partition, and the node merges the two leaf sets. Since leaf sets are periodically exchanged among neighbors, this causes a cascading effect that eventually merges the two partitions. The routing table invariants are then re-established lazily by Pastry's built-in maintenance mechanism.

This mechanism slowly heals the overlay partition by integrating more nodes during each leafset exchange period. Thus, the overlay grows very quickly while the partition is being healed, but it remains stable. In fact, the process may start at multiple locations throughout the overlay, as more nodes detect that the partition has healed and rejoin a single ring.

To test this mechanism under controlled conditions, we set up a simple overlay ring consisting of 22 nodes. The nodes were configured with $T_R = 5$ minutes. We first let the entire network boot and stabilize, and then simulated a network partition by adding a firewall rule on 15 of the machines disallowing any traffic from the other 7 nodes. As expected, the nodes split into two separate overlays, one with 15 nodes and the other with 7. Once these two overlays stabilized, we deleted the firewall rule, removing the simulated partition. Within 4.7 minutes of fixing the firewall, the overlay had healed and all of the leafsets were consistent.

The overhead of this mechanism is rather low: just one extra overlay-routed message per node every T_R. Additionally, the mechanism does not cause an increase in messages during partition heals, as the Pastry leafset consistency mechanism in FreePastry 1.4.1 [18] is periodic and not reactive.

4.1.4 Log reconciliation

Once the overlay has been reconnected, ePOST needs to resolve any inconsistencies that resulted from the partition. The original ePOST was not designed to do this. We opted for an automatic reconciliation mechanism, similar to one found in LOCUS [37], as manual conflict resolution is too inconvenient for users.

Since log heads are the only mutable data objects in ePOST, inconsistencies in logs can only occur at the log heads, when a user moves from one partition to another. In practice this happened rarely, yet the annoyance to users was too great to ignore. To detect that a conflict has occurred, PAST notifies ePOST whenever it encounters different versions of a replica log head, so ePOST can check the log entries for consistency. If ePOST detects that a fork in the logs has occurred, it first determines the last common log entry in the two divergent logs. Then, ePOST merges the newer log entries from both logs using a temporal ordering on the log entries, taking special care to reassign UIDs of newly inserted messages. Merging the logs, however, may result in conflicting entries when destructive operations occur in one of the divergent logs (e.g., moving an email in one partition to a folder that was deleted in the other partition). These conflicts are resolved conservatively by ignoring the destructive operation (e.g., the folder delete is ignored). While this policy may mildly inconvenience users, it is guaranteed to not lose any email.

4.2 Routing anomalies

LESSON 2: *Reliable decentralized systems must be designed to tolerate partial network connectivity.*

Handling network partitions alone, as is described above, is not sufficient to ensure that nodes have a consistent view of the overlay. The initial design of Pastry assumed that the underlying physical network would provide full connectivity among all pairs of overlay nodes. Consistent with this assumption, the failure of a TCP connection attempt to a remote node was interpreted as an indication that the remote node had failed. This assumption held for the most part within a LAN.

However, when we deployed ePOST on PlanetLab, we observed frequent *routing anomalies*, or instances where live nodes were unable to route packets to each other. Others have noted similar behavior on the general Internet [1,22,42]. The main consequence for ePOST was that some objects would become inaccessible during routing anomalies. To see why, assume that a routing anomaly has caused the path from node A to node B to fail. Now, if B wants to retrieve an object (e.g., an e-mail or a log entry) from PAST that is currently stored on A, it uses overlay routing to send a request to A, which includes B's network address. When the request is delivered, A attempts to open a direct connection to B to deliver the object, which fails. For example, a user would try to open an email message she had received earlier, but ePOST could not display the content of the message.

A simple workaround would have been to fall back on using overlay routes when such a routing problem occurs. However, this approach is inefficient, since it is known that most connectivity problems can be circumvented with one forwarding hop, while overlay routes are typically longer [1, 22]. Thus, using overlay routes would inflate network utilization, congest overlay links with bulk data traffic, and place unnecessary forwarding burden on overlay nodes.

We instead chose a different solution, which is described below. Its implementation in FreePastry 1.4.1 was, to the best of our knowledge, the first comprehensive solution to the partial connectivity problem in a decentralized system. Freedman et al. [17] have since proposed an alternate approach, which solves the problem at the application layer.

4.2.1 Anomaly frequency

We examined a subset of the all-pairs ping dataset [51] from PlanetLab, which covered September 1 through September 10, 2004. At the time, PlanetLab consisted of 435 nodes spread over 201 sites, including nodes in the Americas, Europe, the Middle East, Asia, and Australia. The data we examined consisted of node-to-node pings collected every 15 minutes over the course of the run.

In order to investigate the impact of routing anomalies, we limited our evaluation to nodes which were online and

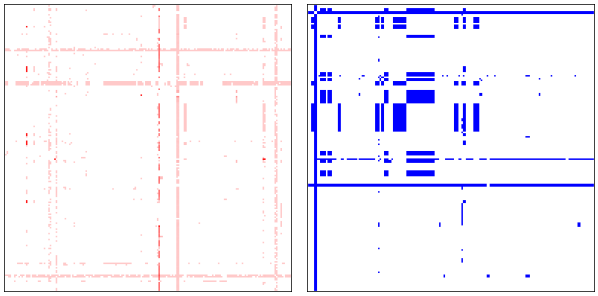

Figure 3: Transient (left) and permanent (right) path failures in PlanetLab during 10 days in September 2004

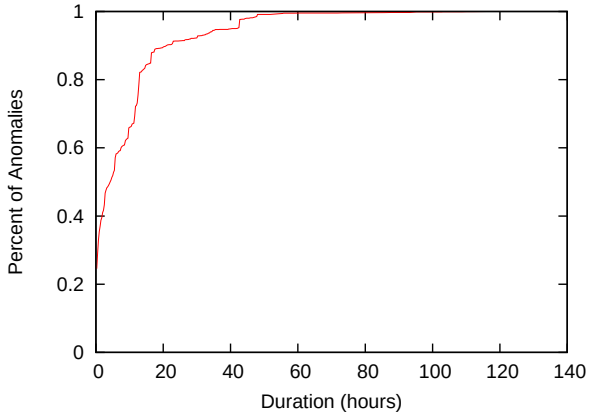

Figure 4: CDF of routing anomaly durations for 192 PlanetLab nodes for September 1 to September 10, 2004.

reachable by at least one other node. This left us with 192 distinct nodes. Figure 3 shows the results of site-to-site pings for the 192 considered nodes. The left graph shows transient failures, where the pixel at the location (x, y) represents the number of times a node x was able to successfully ping node y. White pixels indicate no failures, while darker pixels indicate an increasing frequency of ping failures. The right graph shows permanent failures, or pairs of nodes that were never successful in pinging each other.

These results clearly show that routing anomalies are a common problem in PlanetLab. All of the 192 nodes we examined experienced at least one routing anomaly during the experiment; many of them experienced several. Some of the permanent failures were due to Internet2 nodes, which did not have IP connectivity with some of the others. The average duration of a transient anomaly was 8.8 hours, but the distribution was heavy-tailed; a full 35% of the outages were present for less than one hour, while 9% were longer than a day. Figure 4 shows the cumulative distribution of anomaly duration.

4.2.2 Virtual links

To ensure that all pairs of overlay nodes can efficiently communicate despite routing anomalies, we modified Pastry to allow bulk data connections via intermediate nodes. We view all node connections as *virtual links*, rather than physical links. For example, assume that a link $A \rightarrow B$ has failed. If there is another node C that can still reach B, A can replace its direct link $A \rightarrow B$ with a virtual link $A \rightarrow C \rightarrow B$. In the common case where no path failure has occurred, the virtual link is identical to the actual physical link and thus requires no extra overhead. The use of virtual links is a well-known technique that is widely used in mobile ad-hoc networks [26], where path failures are common. On the general Internet virtual links have been used by several systems to route around an increasing number of observed routing anomalies [1, 22].

We use *source routing* to forward packets over virtual links. Messages sent via source routes are not subject to normal overlay routing—they are either transmitted along the specified path or dropped if an error occurs. This is important to prevent routing loops, as source routing may not observe the normal Pastry routing invariants (i.e., always routing to a node with nodeId closer to the key). Moreover, such source routed virtual links use dedicated TCP connections coupled at the application layer to ensure flow control along the entire virtual link.

We limit our system to using only a node's leaf set members as intermediary hops, which ensures scalability and is sufficient in practice. Every node periodically advertises its best virtual links to its leafset members, who use them to derive virtual links for themselves. For example, if A advertises a link $A \rightarrow B$ to C, and C's best link to A is currently $C \rightarrow A$, then C concludes that B may be reached via $C \rightarrow A \rightarrow B$. Nodes maintain a set of fresh links for each destination, but during normal operation, only the shortest virtual link is used.

When the shortest virtual link to a destination X is not a direct link, the node occasionally sends a probe packet directly to X, using exponential back-off. If the probe is answered, the physical link to X is re-enabled, and all other virtual links are updated accordingly. This ensures that after a transient path failure, the system eventually returns to using physical links.

When a virtual link fails, the sender starts using another link, if one is available. If not, the sender can broadcast a *route request* to all of its leafset members, who attempt to forward it to the destination. The destination responds with a *route reply*. This mechanism was inspired by the DSR ad hoc routing protocol [26] and does not require connectivity to be symmetric. Thus, asymmetric path failures are naturally handled, and nodes behind NATs, firewalls, and advanced traffic shapers, whose connectivity may be asymmetric as well, can be incorporated into the overlay.

Many deployed p2p systems, such as Skype, have built-in mechanisms to enable nodes behind NATs and firewalls to join the systems. These mechanisms serve a different purpose and are less general than our virtual links. They assume that a node behind a NAT or firewall can open a connection to any node in the open Internet, and that a connection, once established, allows two-way traffic for its entire duration.

4.2.3 Improvement

We incorporated virtual links and source routing into our ePOST implementation. As a result, routing anomalies no longer visibly affect the users of ePOST. In a deployment in PlanetLab, the redesigned system communicated between 59,104 distinct pairs of machines and found that 9.1% of these pairs were unable to establish a direct connection. We

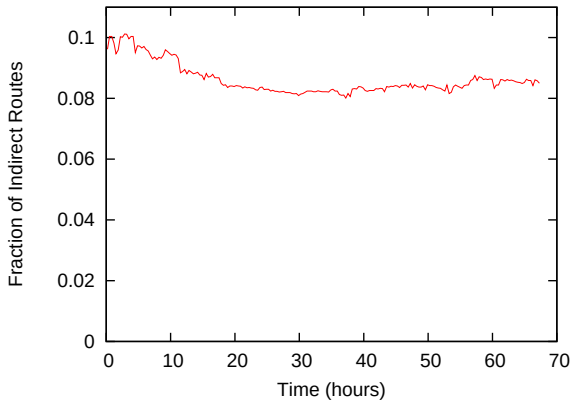

Figure 5: Percentage of indirect routes over a period of three days, March 17 to March 20, 2005.

monitored all of the routes in the network for a period of three days, and found that the percentage of indirect routes varied between 8% and 11%. The results of this experiment are shown in Figure 5. Source routes and virtual links allowed the system to route around network anomalies in all cases, as expected.

4.3 Overlay consistency

LESSON 3: *Reliable decentralized systems should ensure routing consistency in the absence of overlay partitions.*

Our experience showed that even in the absence of network partitions and with virtual links in place to handle routing anomalies, a user would still experience inconsistency. The reason is that due to network churn, nodes temporarily disagreed on the overlay membership and direct data lookups for the same key to different nodes. Although this effect is different from the routing anomalies described in the previous section, the symptoms are similar: For example, users in the PlanetLab-based ring would sometimes see headers of newly arrived messages, but were temporarily unable to retrieve the message bodies. Another problem is that log head updates could be directed to different nodes, creating the possibility of diverging logs. Thus, for ePOST to provide an acceptable user experience, we had to enhance Pastry to maintain a consistent view of the overlay in the presence of node churn.

To do this, we first define routing in overlays as consistent if, in the absence of network partitions, there is at most one node at any given time t that will accept messages for a key k. We then decided to redesign Pastry's leafset stabilization protocol to guarantee, with high probability, that only one node considers itself responsible for a given time at any time, thus providing routing consistency with high probability. MSPastry [6] provides the same property, but only in the absence of routing anomalies. An analytic framework for defining two levels of routing consistency has recently been proposed [9].

4.3.1 Consistency requirements

The mechanism for providing consistency must ensure the following two properties:

- When a new node N joins, the current members must stop accepting messages for the region of key space N is going to take over *before* N starts accepting messages.

- When a node N fails or leaves, N's neighbors may take over N's region of key space only *after* they have established that N no longer accepts messages.

Since the region of key space for which a node is responsible is determined by its two closest neighbors in identifier space, or more generally by its leafset, a consistency protocol must ensure that a node is in its neighbors' leafsets while it accepts messages. Our protocol is based on the assumption that leafsets are always *connected*, i.e., that for each pair of leafset members A and B, there exists a path of leafset nodes $A \rightarrow N_1 \rightarrow \ldots \rightarrow N_k \rightarrow B$ (where N_i is in the leafset) such that each is directly connected to the next, and A is therefore able to send messages to B along this path. Note that this assumption is stronger than the no-partitions assumption that is part of our definition of consistency; we will show in Section 4.3.4 that this stronger assumption is reasonable. Under this assumption, the source routing mechanism described in Section 4.2.2 discovers a route if one exists.

4.3.2 Key ownership

To prevent overlaps between the responsible region of a newly joined node and that of its neighbors, we introduce the concept of *key ownership*. Each node has a range of keys that it owns, and it is not allowed to accept messages for keys outside of this range. When the first node N_1 with nodeId k_1 in the network starts up, it automatically has ownership of the entire key range. As nodes join, they request range transfers from existing nodes. For example, when the next node N_2 with nodeId k_2 joins, it will ask N_1 to transfer ownership of the range

$$\left[\frac{k_1+k_2}{2} \bmod 2^{160}, \frac{k_1+k_2+2^{160}}{2} \bmod 2^{160}\right)$$

Note that once a node releases ownership over a range of keys, it is no longer able to accept messages for those keys. Additionally, each node must obtain ownership transfers from *both* of its neighbors before accepting any messages. During the interim time period messages may be dropped in the interest of preserving consistency. In practice, this is not a concern, since requests can be retried.

If we assume for the moment that no nodes ever leave the overlay (i.e., all nodes stay forever), it is clear that routing consistency is maintained. Since the key space is repeatedly partitioned up, nodes never conflict in their owned ranges. However, as churn is common, we must show that reclaiming transferred key space does not break routing consistency. In order to do so, we impose the rule that a node may reclaim its neighbor's owned keys only if the node is declared dead (as discussed in Section 4.3.3). If a neighbor is declared dead, then, by the assumption that leafsets are connected, we know that no leafset member is able to reach the neighbor, and we can therefore assert that the neighbor is dead. In this case, the surviving node may reclaim its portion of the neighbor's key space.

4.3.3 Liveness checks

Direct neighbors in the key space monitor each other's liveness. For this purpose, they make sure that they receive at least one message from each other within a configurable time period T_P, on the order of 30 seconds. When there is no overlay traffic to send, they may send a `Ping` message instead. Each `Ping` must be answered immediately by a `Pong`, which also must include the source route used in the corresponding `Ping` as a payload.

Should a node A not receive any traffic from B after T_P, A starts sending periodic `Pings` to B over the best known virtual link. Should A still not receive any messages after time $2T_P$, A starts sending `Ping` to B via the best known route to each of its leafset members. If a `Pong` arrives now, A changes B's virtual link to the source route listed in the `Pong`.

However, if $3T_P$ expires before any route is found, A has established that *none* of the other leafset members can directly reach B any more, so under our assumption that leafsets are always connected, B cannot be alive. Hence, A declares B dead. However, A does not remove B from its leafset until another T_P has passed, to ensure that all other leafset nodes declare B dead. Once this total of $4T_P$ has expired, A can remove B from the leafset and then reclaim its keys.

The parameter T_P directly influences the bandwidth required for maintenance. Higher values result in lower bandwidth, but also increase the latency between a node failure and the time when its portion of key space is taken over by the other nodes.

4.3.4 Justification

We assumed above that leafsets were connected, or that in a given node N's leafset $[N_1, N_2, ..., N, ..., N_k]$ there always existed a path between N and every N_i. In this section, we provide a brief justification of why this is reasonable.

As mentioned earlier, we assume that each path $N_i \rightarrow N_j$ in a given leafset fails independently with probability p. For simplicity, we consider virtual links with at most two hops. Then N cannot reach N_i if the direct path $N \rightarrow N_i$ fails *and* for every leafset member L_i, either the path $N \rightarrow L_i$ or the path $L_i \rightarrow N_i$ fails. If the leafset contains l nodes on each side, this occurs with probability

$$P_1 = p \cdot \left(1 - (1-p)^2\right)^m$$

where m is the number of nodes in the shared leafset of N and N_i, which ranges from $l-1$, when N and N_i are far apart, to $2(l-1)$, when they are adjacent. We consider N and N_i disconnected if either of them cannot reach the other one. The probability of this is

$$P_2 = 1 - (1 - P_1)^2$$

As stated above, routing consistency is violated only if N is disconnected from either its left or its right neighbor. This happens with probability

$$P_3 = 1 - (1 - P_2)^2$$

If we assume small leafsets ($l = 8$) and a massive failure rate of $p = 0.1$, then $P_3 \approx 6.072 \cdot 10^{-12}$, so even in a network with $N = 10000$ nodes, the probability of finding a single disconnected node is less than $6.1 \cdot 10^{-8}$. If we consider virtual links with more than two hops, the resulting probability is even lower. For comparison, in a protocol that requires a physical link between each node and its right and left neighbors, the probability of an inconsistency is

$$P_3' = 1 - (1-p)^4$$

For the parameters mentioned earlier, $P_3' \approx 0.3439$, so about one-third of the nodes would be disconnected. In practice, after adding the new leafset stabilization protocol we have not observed any routing inconsistencies.

4.4 Correlated failures

LESSON 4: *Reliable decentralized systems must be built to withstand large-scale failures.*

In order to provide high availability based on unreliable end hosts, a decentralized system must be able to tolerate many of the nodes failing, possibly simultaneously. We initially assumed that the ePOST node population would be highly diverse, so failures would be independent. However, our actual node population was diverse only in some aspects, and not in others. In our initial Rice University ePOST ring, we deployed the network on 10 Red Hat, 14 Debian, 2 Fedora Core, 6 Windows, and 3 OS X machines. These machines were connected over a range of subnets, but they were all inside a single building. In the PlanetLab-based ring, the nodes were connected over a wide range of subnets and power grid, but all shared the same operating system and had similar hardware. As a result, we sometimes observed large-scale correlated failures that affected a large fraction of the nodes. Since early versions of ePOST were not designed to withstand failures of this magnitude, some ePOST user data was lost.

4.4.1 Causes of correlated failures

While deploying ePOST, we found that correlated failures were caused by a range of problems. We experienced a few network failures, causing data to become temporarily unavailable. However, there was no actual data loss from these failures, since the failed nodes re-joined the ring later with their state intact. A few ePOST software failures we experienced, on the other hand, were not always so benign. Occasionally, bugs caused data loss in the overlay.

The first type of correlated failures we experienced did not result in data loss or service unavailability, but caused a partition. For example, at one point, a member node was accidentally configured to claim the IP address of the local router, leading all traffic destined for addresses outside of the subnet to be dropped. This effectively partitioned the ring in roughly two halves, causing many email messages to be unavailable to users.

A second type of failures led to DHT data being temporarily unavailable. For example, the PlanetLab ring suffered two large-scale correlated failures: First, after an upgraded kernel was deployed, 89% of the nodes failed to boot and returned slowly over the following week. The week before that, a separate upgrade had caused 50% of our nodes to fail. Figure 6 shows the number of live nodes in our PlanetLab deployment over time; both of these correlated failures are clearly visible.

The third type of failures was the most severe, because it resulted in actual data loss. First, a bug caused some nodes to crash during periodic data lease extensions (discussed in Section 5.2). Since the nodes never completed the lease extensions, some of the data was lost before we no-

ticed the problem. Second, in the Rice ring, a bug in ePOST lead nodes to incorrectly calculate their range of responsible keys. As a result, the nodes started deleting keys and removed 80% of the data in the network before the bug was caught. Third, in the early stages of the PlanetLab deployment, our PlanetLab slice was accidentally deleted, which caused a 100% correlated failure with complete data loss.

Other interesting failures we experienced included a lightning strike, which caused a campus-wide power outage. We were fortunate enough to never observe a major worm-related failure, probably because most of our early adopters use less common machines such as Apple and Linux workstations. A fast worm such as Slammer [36] could affect a large fraction of the node population within minutes. Some worms have been known to gain administrator privileges, so they could easily erase the hard drive if they wanted to.

4.4.2 Dealing with correlated failures

To handle such failures, we revised our initial design in four areas. Our first response was to simply add more heterogeneity to the system by introducing additional nodes. Several users agreed to run ePOST nodes at home, therefore eliminating the single shared network and power sources. Second, we changed the storage layer to be much more conservative when deleting files. Specifically, we modified the implementation such that it would keep deleted objects in a special 'trashcan' for some time. This allowed us to recover lost objects when the failure was caused by a software bug, and would have prevent some of the failures we experienced from causing actual data loss. Third, we added mechanisms to tolerate and recover from overlay partitions, as described in Section 4.1.

Fourth, and most importantly, we added a durable storage layer called Glacier [23] that stores data with sufficient redundancy to survive large-scale correlated failures of up to 80% of the nodes. Since Glacier has been presented elsewhere, we only briefly describe it here. Glacier uses erasure codes to reduce the storage cost of this redundancy, and it uses aggregation and a loosely coupled fragment maintenance protocol to reduce the maintenance costs. The replicas in the DHT are now used only to provide efficient data access and to mask individual node failures. Since the DHT is no longer required to guarantee data durability, we reduced the number of replicas for immutable objects from five to just three.

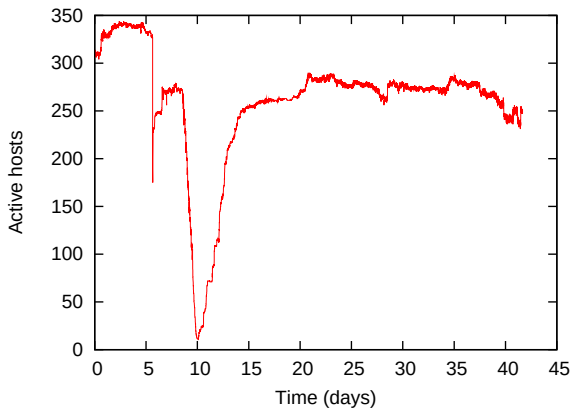

Figure 6: Number of live PlanetLab hosts over a period of 43 days.

Since some of the failures we observed were instantaneous, we decided against a reactive system that would monitor data availability and repair damages as they occur (our underlying DHT, PAST, is an example of a reactive system). Glacier is *proactive* in the sense that it always maintains enough redundancy to sustain an instantaneous worst-case failure, at the expense of a higher storage overhead. Also, due to the difficulty of capturing all sources of correlated failures, we decided against the introspective approach used in Phoenix [27] or OceanStore [30]. Instead, Glacier provides *stochastic* guarantees and assumes only an upper bound on the magnitude of a worst-case failure (e.g., up to 80% of the nodes may fail at the same time).

Since Glacier provides statistical guarantees, Glacier's operational parameters are entirely dependent on the expected maximal fraction of nodes that may suffer a correlated failure. With a single global overlay, this quantity would be difficult to assess, as the make-up of the overlay may not be known. However, since each organization runs its own independent ring, the failure probabilities of such individual rings can be controlled and estimated more easily, allowing for a practical deployment of Glacier with different parameters across different organizations.

4.4.3 Improvement

With Glacier in place, the data in our ePOST deployment can now survive large-scale instantaneous and permanent failures with high probability. Since we introduced Glacier, we have not observed any symptoms of data loss.

5. PITFALLS

In this section, we briefly describe a number of pitfalls we encountered while designing, deploying and running ePOST. Most of these pitfalls have been encountered in other systems and been reported elsewhere. Had we managed to bring to bear on our original design the collective experience of the systems community, we probably could have avoided them. Nevertheless, we think they are important enough to repeat them here as a guide to others, particularly since some of the issues appeared in new disguises.

5.1 Do not rely on fail-stop failures

Like many other designers of p2p systems, we believed that nodes would fail in predictable and detectable ways. Experience showed that even in the absence of security attacks, nodes sometimes failed in complex ways, and the overlay must be prepared to handle such failures.

Bugs in the ePOST software, the libraries, or language runtime would occasionally cause nodes to stall but not crash. In this state, nodes maintained their TCP connections and sometimes accepted incoming connections, but messages sent to the nodes were never delivered. This resulted in insert and lookup failures in PAST, and consequently prevented emails from being delivered and logs from being written. To fix this problem, we added a *watchdog* process, which periodically pings the ePOST process locally and automatically restarts it if no response is received.

DHT objects inserted in PAST are stored on the local disk of the ePOST nodes that are responsible for their key. Stored with the key is node-specific metadata such as the lifetime of the object in PAST. When a node fails, such data

cannot be recovered from other nodes and must be protected on disk. To prevent occasional corruption of this metadata due to node failures, we had to add an atomic disk write mechanism.

5.2 Avoid strong assumptions regarding resource consumption

Like many other designers of p2p systems we believed that the underutilized resources of the participating nodes could provide all of the resources needed by the system. During the initial design of ePOST, we studied the growth of consumer hard disk capacity and measured the amount of data users typically store in their mail folders. We found that the disk growth rate far exceeded the storage requirements over time. This led us to conclude that we could easily retain all ePOST data indefinitely. To maintain PAST keys among replica nodes, our replication protocol simply exchanged the complete list. We also initially stored the PAST data on disk by creating one file per key in a single directory.

In practice, we found that while the amount of disk storage required was acceptable, the number of keys we were forced to manage grew very large. First, we had not anticipated the large increase in spam email since the initial design of ePOST, which led to a large increase in the number of objects stored in the DHT. Second, the addition of Glacier introduced 48 additional fragments for each DHT object, significantly increasing the number of keys ePOST needed to manage.

Key management became a burden on the memory, CPU, and network resources. On average, over $1,700$ objects were inserted daily, which resulted in approximately $86,000$ new keys generated daily or $2,600$ new keys per machine per day. This large number of keys was due to log entries, data replication, and Glacier fragments. Many of these keys were no longer referenced by any objects, but they could not be deleted. This *garbage* came from deleted email data and old log entries in ePOST. In fact, we found that after running ePOST for only 6 months, 75% of the keys stored in the DHT were garbage.

To handle this explosion of keys, we were forced to change the ePOST system in several ways. The replication began to incur a significant overhead due to the large key lists, so we instead used Bloom filters [4] to optimize key list exchange. These allow smaller messages to be exchanged during replication, and do not require the full key list to be sent. Second, we implemented a log coalescing scheme which collapsed 50 log entries into a single entry, thereby reducing the number of new keys.

Third, to reduce the large key increase from Glacier fragments, we implemented an *aggregation* layer that bundles small objects and inserts them into Glacier as a single object. Aggregation reduced the number of keys required by Glacier in our deployment by a factor of 30, greatly aiding the scalability of the storage layer.

Fourth, to reduce the number of keys referencing garbage objects, we implemented a lease-based version of the PAST DHT, which stores objects for limited time intervals. When inserting an object, an expiration time is specified, and user proxies can later extend the expiration time if they are still interested in the object. Therefore, objects that are no longer referenced by any user are eventually garbage collected and deleted. This has the secondary effect of reducing the storage needed as these garbage objects are deleted. The use of leases to control the lifetime of stored objects obviates the need for a delete operation, which would present a severe security risk. Leases are commonly used in distributed storage systems; for example, they have also been used in Tapestry [57] and CFS [13].

5.3 Consider using a database library to maintain persistent data

Storing a large number of keys also became a problem on machines using the FAT and NTFS filesystems, which limit the total number of files in a directory. Moreover, in ext3, a bug surfaced once more than 64,000 files were stored in a directory and the NTFS file system became increasingly slow as the number of keys increased.

To work around the file system scalability problems, we implemented a directory-splitting mechanism that maintains a directory hierarchy and limits the number of keys in a single directory. Once this limit is exceeded, subdirectories are created, and the keys are redistributed into the newly created directories. With hindsight, we may have been better off using a database library like BerkeleyDB [39] to manage persistent data on disk, as was done for instance, in DHash [13], for instance. This would have avoided the scalability problems in file systems and have freed us from having to implement our own atomic disk write mechanism, though the use of a complex black-box system like a database comes with risks as well.

5.4 Anticipate slow nodes and congested links

Like most academic research projects, ePOST was initially designed and tested on a LAN with well-maintained machines. It did not consider what would happen when older nodes or nodes with slow network connections participated in the system. The original design of the routing mechanism ignored overloaded nodes and severely congested network links; the next hop of an overlay route was always chosen based solely on its nodeId and the periodically measured, smoothed RTT. Thus, intermittently overloaded nodes or ones whose network links were congested were chosen alongside other nodes.

Experience with a deployment 'in the wild' (e.g., in PlanetLab, the open Internet, and with DSL and cable modems), quickly showed that such nodes introduced significant delays. This affected the user experience, as the interactive response time of folder operations and email retrieval suffered.

To work around this problem, we implemented fast rerouting, in which nodes constantly monitor the forwarding queue to a given node, and quickly re-route messages destined for slow nodes. Additionally, the nodes use the length of the queue as a hint to check on the liveness of the remote node using UDP pings, in case the remote node has stalled or is dead. Similar mechanisms are used in MS-Pastry [6] and Bamboo [2].

Additionally, in the initial implementation of Glacier, data objects were pushed to newly joining nodes. We found that this caused congestion on nodes connected via relatively slow links, e.g., those connected via DSL or cable modem. To fix this, we added configurable traffic shaping based on token buckets, which allows a node to control the speed at which data is downloaded.

5.5 Watch for hidden single points of failure

One of the strengths of p2p systems is the absence of a single point of failure. In practice, however, not all sources of failures are obvious, and great care must be taken to prevent single points of failure from creeping in unexpectedly.

In the initial ePOST design, the set of bootstrap nodes was specified by their domain names, which all ended in 'epostmail.org'. Since DNS is a highly available, replicated name service, the implicit dependence on DNS translations seemed safe. However, at one point, our domain name provider misconfigured the DNS entry for 'epostmail.org', making it impossible for nodes to join the ePOST ring, thus causing the ePOST ring to slowly fade away as a result of normal churn. The problem was solved trivially by storing a cached IP address as a hint with the domain name.

Despite replication and Glacier, there is a non-zero probability that a data item will eventually get lost. Moreover, a software error may cause a corrupted data item to be inserted into the PAST store. An extremely rare loss of a single data item seems tolerable and is ultimately unavoidable in any real storage system. As it turns out, however, if the lost data item is a log entry and the log is organized as a singly linked list, the entire log preceding the lost entry becomes inaccessible. A trivial solution is to store some skip pointers into each entry, greatly reducing the probability of a loss of the log.

Lastly, there is a single point of failure that is actually desirable from a security standpoint, but requires awareness and care on the part of users. If a user loses their ePOST key material, their email account becomes inaccessible. There is no system administrator who can reset their key and recover the data.

6. DIRECTIONS FOR FUTURE RESEARCH

We have identified four major interesting directions for future work on reliable decentralized systems. First, the ePOST solution for dealing with inconsistent data after network partitions is specific to the class of data that ePOST stores: single-writer logs with few mutable log heads and mostly immutable data. A more general solution for other types of applications and data remains an interesting open problem.

Second, while ePOST was designed to be secure, and we are not aware of any vulnerabilities in the ePOST layer itself, our deployment has not been large enough to be subjected to any security or denial-of-service attacks on either ePOST or the components on which it is built. Studying the security and robustness of decentralized systems like ePOST "in the wild" is an important subject for future research.

Third, one of the goals of ePOST was to reduce the amount of systems management humans were required to perform on reliable systems. We hoped that the system would be self-managing to a large extent. Given that the system is fundamentally different from a server-based email system, that it has been under active development until recently, and we do not have a large enough deployment to do a meaningful study of manageability, the results are still inconclusive. An interesting question is to what extent decentralized systems can be made to be self-managing, and which tasks fundamentally require human intervention.

Fourth, a broader question is what the class of reliable user applications can decentralized systems support. ePOST benefits from the time scale of operations and the consistency requirements that are typical of collaboration applications: relatively infrequent events requiring completion times that are acceptable to human users, with relatively weak consistency requirements and little contention. We have demonstrated that reliable applications with these characteristics can be built today. However, what other classes of applications with challenging reliability demands can reasonably be built on top of decentralized, cooperative systems?

7. CONCLUSION

In this paper, we have reported our experience in designing, deploying and operating ePOST, a decentralized email service that is based on p2p technology. ePOST was intended as a 'proving ground' to study whether p2p systems can actually deliver the high reliability they promise. We ran into challenges providing the reliability and data durability required for such a system, especially in the case of network partitions, routing anomalies, node churn, and correlated failures. We were able to solve some of these issues by applying known principles of system design, and others by devising new techniques and tools. Beyond ePOST, a number of important challenges still remain, such as a general solution for handling network partitions and creating more self-managing reliable systems.

8. ACKNOWLEDGMENTS

This research was supported in part by Texas ATP (003604-0079-2001), by NSF (CNS-0509297 and ANI-0225660), and by Microsoft Research. We wish to thank Dan Wallach, Eugene Ng, Atul Singh, Animesh Nandi, and Dan Sandler for their ideas, advice, and support. We also wish to thank all of the users who agreed to run ePOST, especially while it was still being developed. Lastly, we thank our shepherd, Christof Fetzer, and the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of SOSP'01*, Banff, Canada, Oct. 2001.

[2] Bamboo DHT. `http://bamboo-dht.org/`.

[3] BitTorrent protocol. `http://www.bittorrent.com/protocol`.

[4] B. Bloom. Space-time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of SIGMETRICS'00*, Santa Clara, CA, 2000.

[6] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of DSN'04*, Florence, Italy, June 2004.

[7] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of OSDI'02*, Boston, MA, December 2002.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

[9] W. Chen and X. Liu. Enforcing routing consistency in structured peer-to-peer overlays: Should we and could we? In *Proceedings of IPTPS'06*, Santa Barbara, CA, February 2006.

[10] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 20(8), 2002.

[11] Coral content distribution network. `http://www.scs.cs.nyu.edu/coral/`.

[12] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of OSDI'02*, Boston, MA, December 2002.

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

[14] J. R. Douceur. The Sybil attack. In *Proceedings of IPTPS'02*, Cambridge, MA, Mar. 2002.

[15] eDonkey file sharing network. `http://www.edonkey2000.com/`.

[16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of NSDI'04*, San Francisco, CA, 2004.

[17] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proceedings of WORLDS'05*, San Francisco, CA, Dec 2005.

[18] Freepastry. `http://freepastry.rice.edu/`.

[19] R. Friedman and K. Birman. Trading consistency for availability in distributed systems. Technical Report TR95-1579, Cornell University, Apr 1996.

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

[21] S. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer voip system. In *Proceedings of IPTPS'06*, Santa Barbara, CA, February 2006.

[22] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, and H. M. Levy. Improving the reliability of internet paths with one-hop source routing. In *Proceedings of OSDI'04*, San Francisco, CA, Dec. 2004.

[23] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of NSDI'05*, Boston, MA, May 2005.

[24] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of IPTPS'02*, Cambridge, MA, Mar. 2002.

[25] Y. hua Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an internet broadcast system based on overlay multicast. In *Proceedings of USENIX'04*, Boston, MA, June 2004.

[26] D. B. Johnson, D. A. Maltz, and J. Broch. The dynamic source routing protocol for multihop wireless ad hoc networks. In *Ad Hoc Networking*, edited by Charles E. Perkins, Chapter 5, pages 139–172. Addison-Wesley, 2001.

[27] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving internet catastrophes. In *Proceedings of USENIX'05*, Anaheim, CA, Apr 2005.

[28] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of IPTPS'04*, San Diego, CA, 2004.

[29] Kazaa. `http://www.kazaa.com/us/index.htm`.

[30] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS'00*, Cambridge, MA, Nov 2000.

[31] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li. An empirical study of collusion behavior in the Maze p2p file-sharing system. Technical Report MSR-TR-2006-14, Microsoft Research, February 2006.

[32] R. C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy*, page 122, 1980.

[33] A. Mislove. "POST: A Decentralized Platform for Reliable Collaborative Applications". Master's thesis, Rice University, Houston, TX, 2004.

[34] A. Mislove and P. Druschel. Providing administrative control and autonomy in structured overlays. In *Proceedings of IPTPS'04*, San Diego, CA, February 2004.

[35] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. S. Wallach, X. Bonnaire, P. Sens, J.-M. Busca, and L. Arantes-Bezerra. POST: A secure, resilient, cooperative messaging system. In *Proceedings of HotOS'03*, Lihue, HI, May 2003.

[36] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[37] E. T. Mueller, J. D. Moore, and G. J. Popek. A nested transaction mechanism for LOCUS. In *Proceedings of SOSP'83*, Breton Woods, NH, Oct 1983.

[38] P. Murray. The Anubis service. Technical Report HPL-2005-72, HP Labs, June 2005.

[39] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track of USENIX'99*, Monterey, CA, Jun 1999.

[40] eDonkey 2000 - OverNet. `http://www.edonkey2000.com`.

[41] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proceedings of OSDI'04*, San Francisco, CA, Dec. 2004.

[42] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of SIGCOMM'97*, Cannes, France, September 1997.

[43] PlanetLab. `http://www.planet-lab.org/`.

[44] V. Ramasubramanian and E. Sirer. The design and implementation of a next generation name service for the Internet. In *Proceedings of SIGCOMM'04*, Portland, OR, Aug 2004.

[45] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of SIGCOMM'05*, Philadelphia, PA, Aug. 2005.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware'01*, Heidelberg, Germany, Nov. 2001.

[47] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP'01*, Banff, Canada, Oct. 2001.

[48] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[49] K. Sripanidkulchai, A. Ganjam, B. Maggsand, and H. Zhang. The feasability of supporting large-scale live streaming applications with dynamic application end-points. In *Proceedings of SIGCOMM'04*, Portland, OR, Aug. 2004.

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

[51] J. Stribling. All pairs ping results for PlanetLab. http://www.pdos.lcs.mit.edu/∼strib/pl_app/.

[52] J. Stribling, I. Councill, J. Li, M. F. Kaashoek, D. R. Karger, R. Morris, and S. Shenker. OverCite: A cooperative digital research library. In *Proceedings of IPTPS'04*, Ithaca, NY, Feb. 2005.

[53] R. van Renesse, K. P. Birman, D. Dumitriu, and W. Vogel. Scalable management and data mining using Astrolabe. In *Proceedings of IPTPS'02*, Cambridge, MA, Mar. 2002.

[54] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from DNS. In *Proceedings of NSDI'04*, San Francisco, CA, 2004.

[55] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of USENIX'04*, Boston, MA, June 2004.

[56] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of SIGCOMM'04*, Portland, OR, Aug. 2004.

[57] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2001.