

# NOTE TO USERS

This reproduction is the best copy available.

**UMI<sup>®</sup>**



RICE UNIVERSITY

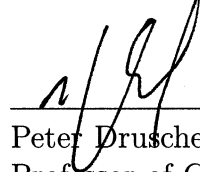
**POST: A Decentralized Platform for Reliable Collaborative  
Applications**

by

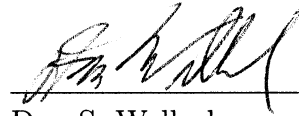
**Alan E. Mislove**

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE  
**MASTER OF SCIENCE**

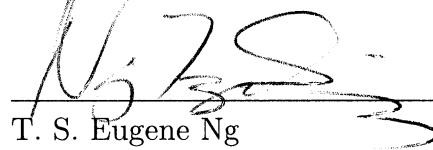
APPROVED, THESIS COMMITTEE:



Peter Druschel, Chair  
Professor of Computer Science



Dan S. Wallach  
Assistant Professor of Computer Science



T. S. Eugene Ng  
Assistant Professor of Computer Science

Houston, Texas

November, 2004

UMI Number: 1425850

## INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.



---

UMI Microform 1425850

Copyright 2005 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

## ABSTRACT

### POST: A Decentralized Platform for Reliable Collaborative Applications

by

Alan E. Mislove

Traditional collaborative applications, such as email and newsgroups, are among the most successful and widely distributed applications. However, such services are almost exclusively based on centralized servers, which inherently limits their scalability and fault tolerance. This thesis presents the design of POST, a platform for such collaborative applications which is completely decentralized and is based on the Pastry peer-to-peer overlay. To show that POST is sufficient to support even the most demanding applications, this thesis also presents ePOST, an email service built on POST. ePOST is in use as the primary email system for actual users, demonstrating an email system which is inherently more scalable and potentially more fault tolerant than existing systems. The success of POST shows that peer-to-peer technology is mature enough to support reliable applications, and that other traditionally client-server applications can be improved through the use of peer-to-peer technologies.

## Acknowledgments

I would first like to thank my thesis committee, whose insightful comments were invaluable. Specifically, I would also like to thank my advisor, Peter Druschel, for his guidance. Additionally, I would like to extend much thanks to all of the ePOST users who agreed to use our experimental email system - without them, none of this work would have been possible. I would also like to thank all of the other people who worked on the POST project, including Ansley Post, Andreas Haeberlen, Charles Reis, Jeff Hoyer, and Derek Ruths. I would like to thank Atul Singh and Animesh Nandi for their insightful discussions which shaped the design of POST and ePOST.

I owe an eternal debt of gratitude to my parents for their support and guidance throughout my life. Last, but certainly not least, I would like to thank Rebecca, whose friendship, understanding, and support always kept me motivated.

# Contents

Abstract	ii
Acknowledgments	iii
List of Figures	viii
List of Tables	ix
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	3
<b>2 Background</b>	<b>5</b>
2.1 Email Systems . . . . .	5
2.2 Peer-to-Peer (p2p) Overlays . . . . .	6
2.2.1 Pastry . . . . .	6
2.2.2 PAST . . . . .	7
2.2.3 Scribe . . . . .	9
<b>3 Scoped Overlays</b>	<b>10</b>
3.1 Design . . . . .	11
3.2 Ring structure . . . . .	12
3.3 Gateway nodes . . . . .	14
3.4 Routing . . . . .	14

3.5	Global lookup . . . . .	15
3.6	Multi-level ring hierarchies . . . . .	16
<b>4</b>	<b>POST Design</b>	<b>20</b>
4.1	User accounts . . . . .	21
4.2	Single-copy store . . . . .	22
4.3	Event notification . . . . .	24
4.4	Metadata . . . . .	26
4.5	Garbage collection . . . . .	27
4.6	POST Security . . . . .	29
4.6.1	Threat model . . . . .	30
4.6.2	Data privacy . . . . .	30
4.6.3	Data integrity . . . . .	31
4.6.4	Data durability . . . . .	32
4.6.5	Denial of service . . . . .	33
4.6.6	Freeloading . . . . .	33
<b>5</b>	<b>ePOST Design</b>	<b>35</b>
5.1	Email storage . . . . .	36
5.2	Email delivery . . . . .	37
5.3	Email folders . . . . .	38



5.4	Incremental Deployment . . . . .	39
5.5	Management . . . . .	40
5.5.1	Software . . . . .	41
5.5.2	Storage . . . . .	41
5.5.3	Access . . . . .	42
5.6	Discussion . . . . .	43
5.6.1	Feasibility . . . . .	43
5.6.2	Mailing Lists . . . . .	44
5.6.3	Spam . . . . .	45
<b>6</b>	<b>Evaluation</b>	<b>47</b>
6.1	Deployment Setup . . . . .	47
6.1.1	Timeline . . . . .	47
6.1.2	PAST . . . . .	49
6.1.3	ePOST . . . . .	50
6.2	Data Storage . . . . .	51
6.3	Message Traffic . . . . .	54
6.4	Single-Copy Store . . . . .	55
6.5	User-Perceivable Performance . . . . .	56
6.5.1	Email Delivery Time . . . . .	57
6.5.2	Folder Operations . . . . .	57

6.5.3	Availability . . . . .	60
6.6	Discussion . . . . .	60
<b>7</b>	<b>Related Work</b>	<b>62</b>
7.1	Collaborative Applications . . . . .	62
7.1.1	Scalability . . . . .	63
7.1.2	Security . . . . .	64
7.2	Peer-to-Peer Applications . . . . .	64
<b>8</b>	<b>Conclusions</b>	<b>66</b>
	<b>References</b>	<b>67</b>

## List of Figures

3.1	Diagram of application layers. . . . .	12
3.2	Example of a ring structure. . . . .	13
3.3	Pseudocode for routing between rings. . . . .	18
3.4	Diagram of a the routing process with multiple levels of hierarchy. . .	19
5.1	ePOST Stack . . . . .	36
5.2	Messages transmitted sending an email. . . . .	39
6.1	Number of participating machines . . . . .	48
6.2	Amount of churn in the ePOST ring. . . . .	49
6.3	Histogram of email sizes . . . . .	51
6.4	Total data stored in the ring, including PAST and Glacier. . . . .	52
6.5	Total data stored on each machine, including PAST and Glacier. . . .	53
6.6	Distribution of nodeIds in experimental ring. . . . .	53
6.7	Storage requirements if virtual nodes are used. . . . .	54
6.8	Cumulative number of messages sent by ePOST components. . . . .	55
6.9	Cumulative number of bytes sent by ePOST components. . . . .	55
6.10	Histogram of the delivery time for emails with online recipient. . . .	58
6.11	CDF of the folder write latency. . . . .	59

## List of Tables

4.1	POST API . . . . .	21
4.2	Modified PAST API . . . . .	28

# Chapter 1

## Introduction

Traditional email and news services, as well as newer collaborative applications like instant messaging, bulletin boards, shared calendars and whiteboards, are among the most successful and widely used distributed applications. Today, such services are implemented in the client-server model, with messages stored on and routed through dedicated servers, each hosting a set of user accounts. This partial centralization limits availability, since a failure or attack on a server denies service to the users it supports. Also, substantial infrastructure, maintenance, and administration costs are required to scale these services to large numbers of users. This is true in particular for the semantically richer and more complex messaging systems like Microsoft Exchange and Lotus Notes [47, 48].

The use of a decentralized approach, such as a peer-to-peer (p2p) based solution, seems like a natural fit for the above problems. For example, p2p overlays remove all single points of failure by distributing the services across all member nodes, providing the potential for a more highly available system. Additionally, p2p systems scale logarithmically with the number of participating users, removing the scalability bottleneck, which is one of the major problems with current collaborative systems.

However, it is not yet known whether p2p systems are mature enough to support

*mission-critical* applications, or ones which users rely on for their daily work. Current deployed p2p systems provide services such as file-sharing [35, 21], web server shielding [19], or a general purpose distributed hash table (DHT) [27], and while these applications demonstrate the benefits of p2p technology, none of them are relied upon by users for their daily work.

This thesis presents POST, a p2p system which can support mission-critical applications. POST offers a resilient, decentralized infrastructure that leverages the resources of users' desktop workstations to provide collaborative services. POST provides three basic, efficient services to applications: secure persistent single-copy storage, metadata based on single-writer logs, and event notification. As is demonstrated later in the thesis, a wide range of collaborative applications can be constructed on top of POST using just these services.

POST itself is built upon a structured p2p overlay network, lending it with scalability, resilience and self-organization. Users contribute resources to the POST system (CPU, disk space, network bandwidth), and in return, they are able to utilize its services. POST assumes that participating nodes can suffer Byzantine failures. Stronger failure assumptions, like simple crash failures, may be unrealistic even within a single organization, because a single compromised node may be able to disrupt critical messaging services or disclose confidential messages.

In this thesis, we present the design of the POST infrastructure, and show how it

can be used to support a variety of collaborative applications. In order to demonstrate the flexibility POST provides, we also detail the design of an email system, ePOST, which is built on top of POST. We chose email as an initial benchmark application because it is well understood, because users' expectations of availability, privacy, and durability place challenging demands on the underlying infrastructure. Finally, we evaluate POST and ePOST, using both simulations and the results of a deployment. The ePOST system was deployed with in the Computer Science department at Rice University to a real user base and experimental results were collected.

## 1.1 Contributions

Specifically, the contributions of this thesis are

- An extension to existing peer-to-peer overlays, called *scoped rings* which is necessary to deploy mission-critical decentralized applications.
- The design of POST, a decentralized platform for collaborative applications which is sufficient to support even the most demanding collaborative applications.
- The design of ePOST, an email system based on POST, showing that POST can easily handle complex applications.

- An evaluation of POST and ePOST based on simulations and a deployment of ePOST with real users relying on the system.

The remainder of this thesis is organized as follows. Chapter 2 provides background on peer-to-peer overlays and applications. Chapter 3 describes the scoping over overlays which was necessary to deploy POST. Chapter 4 presents the design of POST in detail and describes how POST meets the requirements of mission-critical applications. Chapter 5 details the design of ePOST, and Chapter 6 presents the results of an initial deployment of ePOST. Chapter 7 discusses related work and Chapter 8 concludes.



## Chapter 2

# Background

In this chapter, we discuss current email systems and protocols, which ePOST replaces, as well as p2p overlays, which we use to build POST and ePOST.

### 2.1 Email Systems

Email was first developed as messaging system on top of the ARPAnet. The Simple Mail Transfer Protocol (SMTP) for transmitting email between mail servers was formalized in 1982 as RFC 821 [37]. SMTP provides a simple, lightweight protocol for message transmission, but it does not directly support sender verification or encryption. This lack of security has lead to an explosion of bulk unsolicited commercial email, or spam [2], as senders of such email are easily able to forge the headers of their emails. Extensions to SMTP [28] were developed after email's popularity soared, but none of these extensions are used consistently and there is debate to this day over what is the best way to secure SMTP [42, 43].

Protocols for accessing and managing email were developed as email gained in popularity. The first of such protocols, the Post Office Protocol (POP3) was formalized in 1988 [32], and provides a simple way of downloading newly arrived email. A semantically richer protocol, the Internet Message Access Protocol (IMAP) was created in 1994 [13], and is today the de-facto standard for client-program-based email

message access.

Web-based email or *webmail* is an alternative method of email access which has recently gained in popularity. Webmail providers, including Hotmail [25] and Google [22], typically present the user with a set of web pages containing their email, which allows the user to access their email by using just a web browser.

## 2.2 Peer-to-Peer (p2p) Overlays

POST relies on *Pastry*, a structured overlay network, as well as two basic services built upon Pastry: *PAST*, a distributed storage system and *Scribe*, a group communication system. POST could easily be layered above similar systems like Chord/CFS, or Tapestry/OceanStore [45, 15, 29, 49] which are compatible with the key-based routing (KBR) API [16].

### 2.2.1 Pastry

Pastry [39] is a structured p2p overlay network designed to be self-organizing, highly scalable, and fault tolerant. In Pastry, every node and every object is assigned a unique identifier randomly chosen from a 160-bit id 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 in less than  $\log_{2^b} N$  hops, where  $N$  is the number of nodes in the network and  $b$  is a configuration parameter which is usually 4. Eventual delivery is guaranteed unless  $\lfloor l/2 \rfloor$  nodes with *adjacent*

nodeIds fail simultaneously, where  $l$  is a configuration parameter with typical value 12.

### 2.2.2 PAST

PAST [40] is a storage system built on top of a structured overlay and can be viewed as a distributed hash table (DHT). Each stored item in PAST is given a 160 bit key (hereafter referred to as the *handle*), and replicas of an object are stored at the  $k$  live nodes whose nodeIds are the numerically closest to the object’s handle. PAST maintains the invariant that the object is replicated on  $k$  nodes, regardless of node addition or failure.

Since nodeId assignment is random, these  $k$  nodes are unlikely to suffer correlated failures. PAST relies on Pastry’s secure routing [7] to ensure that  $k$  replicas are stored on the correct nodes, despite the presence of malicious nodes who may attempt to prevent this. Throughout this thesis, we assume that at most  $k - 1$  nodes are faulty in any replica set. Section 4.6 discusses the basis for this assumption.

PAST is used in POST to store three types of data: *content-hash blocks*, *certificate blocks*, and *public-key blocks*.

#### Content Hash Blocks

Content-hash blocks are stored using the cryptographic hash of the block’s contents as the handle and are immutable after being created. Content-hash blocks can

be authenticated by obtaining a single replica and verifying that its contents match the handle; because they are immutable after creation, any violation of this can be easily detected when the hash of the block does not match its handle.

### **Certificate Blocks**

Certificate blocks are signed by a trusted third party and bind a public key to a name (for instance, an email address). Certificate blocks are stored using the cryptographic hash of the name as the handle and are also immutable after creation. Certificate blocks, as with content-hash blocks, can be verified easily based on the digital signature of the third-party trusted authority who initially signed them.

### **Public-Key Blocks**

Public-key blocks contain monotonically increasing timestamps, are signed with a private key, and are stored using corresponding public key as the handle, allowing for block mutation after creation. Public key blocks, however, require more elaborate mechanisms to support mutation. First, to prevent an impostor from trying to post a forged update, the nodes maintaining the block must verify that the signature on the update matches the already-known public key. Likewise, to prevent an attacker from trying to roll the block back to an earlier valid state, the nodes maintaining the block verify that the timestamps are advancing forward. Finally, to prevent a man-in-the-middle attack or a malicious replica node when fetching a block, the object

requester must obtain all  $k$  replicas, verify their signatures, and discard any with older timestamps. So long as at least one of the  $k$  nodes returns the desired block, the requester will get the most recent version of the object.

### 2.2.3 Scribe

Scribe [9] is a highly scalable group communication system built on top of Pastry. Each Scribe group has a 160 bit *groupId* which serves as the address of the group. The nodes subscribed to each group form a multicast tree, consisting of the union of Pastry routes from all group members to the node with *nodeId* numerically closest to the *groupId*. Since membership maintenance is distributed throughout the tree, Scribe can handle highly dynamic groups. Scribe also supports an *anycast* primitive, which allows any node in the overlay to efficiently locate a nearby member of a given group in a decentralized fashion.

Scribe takes advantage of proximity-based neighbor selection in Pastry to provide trees which are efficient in terms of network proximity. Since Pastry routes from close nodes are likely to converge soon, Scribe trees will send a small number of copies of each message to each organization with high probability.

## Chapter 3

# Scoped Overlays

Structured peer-to-peer (p2p) overlay networks provide a self-organizing, decentralized substrate for distributed applications and support powerful abstractions such as distributed hash tables (DHTs) and group communication [24, 38, 39, 45, 49, 30]. Most of these systems use randomized object keys and node identifiers, which yields good load balancing and robustness to failures. However, in such overlays, applications cannot ensure that a key is stored in the inserter’s own organization, a property known as *content locality*. Likewise, one cannot ensure that a routing path stays entirely within an organization when possible, a property known as *path locality*. In an open system where participating organizations have conflicting interests, this lack of control can raise concerns about autonomy and accountability [24]. This is particularly a problem when deploying mission-critical services, as organizations may desire that organizational data always remain within the organization.

Moreover, participants in a conventional overlay must agree on a set of protocols and parameter settings like the routing base, the size of the neighbor set, failure detection intervals, and replication strategy. Optimal settings for these parameters depend on factors like the expected churn rate, node failure probabilities, and failure correlation probability. These factors may not be uniform across different organizations

and may be difficult to assess or estimate in a Internet-wide system. The choice of parameters also depends on the required availability and durability of data, which is likely to differ between participating organizations. Yet, conventional overlays require global agreement on protocols and parameter settings among all participants.

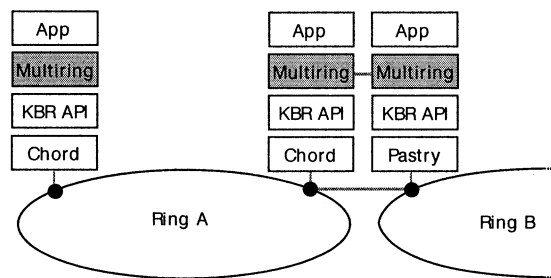
In POST, the p2p overlay is divided into a hierarchy of overlay instances with separate identifier spaces. The hierarchy reflects administrative and organizational domains, and naturally respects connectivity constraints. This technique leaves participating organizations in control over local resources, choice of protocols and parameters, and provides content and path locality. Each organization can run a different overlay protocol and use parameter settings appropriate for the organization’s network characteristics and requirements. This generalizes existing protocols with a single id space, thus leveraging prior work on all aspects of structured p2p overlays, including secure routing [7].

### 3.1 Design

A *multi-ring* protocol stitches together the rings and implements global routing and lookup. To applications, the entire hierarchy appears as a single instance of a structured overlay network that spans multiple organizations and networks. The rings can use any structured overlay protocol that supports the key-based routing (KBR) API defined in Dabek et al. [16].

Figure 3.1 shows how our multi-ring protocol is layered above the KBR API of the

overlay protocols that implement the individual rings. Shown at the right is a node that acts as a gateway between the rings, and is therefore two instances of the same node in the different rings. The instances of structured overlays that run in each ring are completely independent. In fact, different protocols can run in the different rings, as long as they support the KBR API.



**Figure 3.1** Diagram of application layers.

## 3.2 Ring structure

The system forms a tree of rings. Typically, the tree consists of just two layers, namely a *global ring* as the root and *organizational rings* at the lower level. Each ring has a globally unique *ringId*, which is known to all members of the ring. The global ring has a well-known *ringId* consisting of all zeroes. It is assumed that all members of a given ring are fully connected in the underlying physical network, i.e., they are not separated by firewalls or NAT boxes.

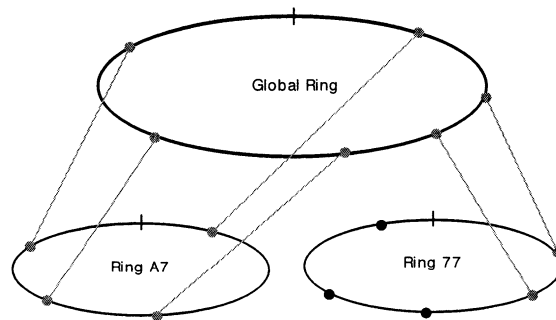
All nodes in the entire system join the global ring, unless they are connected behind a firewall or a NAT. In addition, each node joins a ring consisting of all the nodes that belong to a given organization. A node is permitted to route messages



and perform other operations only in rings in which it is a member.

The global ring is used primarily to route inter-organizational queries and to enable global lookup of keys, while application keys are stored in the organizational rings. Each organizational ring defines a set of nodes that use a common set of protocols and parameter settings; they enjoy content and path locality for keys that they insert into the overlay. In addition, an organizational ring may also define a set of nodes that are connected to the Internet through a firewall or NAT box.

An example configuration is shown in Figure 3.2, where nodes shown in gray are instances of the same node in multiple rings and nodes in black are only in a single ring due to a firewall. The nodes connected by lines are actually instances of the same node, running in different rings. Ring A7 consists of nodes in an organization that are fully connected to the Internet. Thus, each node is also a member of the global ring. Ring 77 represents a set of nodes behind a firewall. Here, only two nodes can join the global ring, namely the firewall gateway nodes.



**Figure 3.2** Example of a ring structure.

### 3.3 Gateway nodes

A node that is a member of more than one ring is a *gateway node*. Such a node supports multiple virtual overlay nodes, one in each ring, but uses the same `nodeId` in each ring. Gateway nodes can forward messages between rings, as described in the next section. In Figure 3.2 above, all of the nodes in ring *A7* are gateway nodes between the global ring and ring *A7*. To maximize load balance and fault tolerance, all nodes are expected to serve as gateway nodes, unless connectivity limitations (firewalls and NAT boxes) prevent it.

Gateway nodes announce themselves to other members of the rings in which they participate by subscribing to a group in each of the rings. The group identifiers of these groups are the `ringIds` of the associated rings. In Figure 3.2 for instance, a node *M* that is a member of both the global ring and *A7*, joins the Scribe groups:

Scribe group *A700...0* in the global ring

Scribe group *0000...0* in ringId *A7*

### 3.4 Routing

Next, we describe how messages are routed in the system. We assume that each message carries, in addition to a target key, the `ringId` of the ring in which the key is stored. In the subsequent section, we will show how to obtain these `ringIds`.

Recall that each node knows the `ringIds` of all rings in which it is a member. If

the target ringId of a message equals one of these ringIds, the node simply forwards the message to the corresponding ring. From that point on, the message is routed according to the structured overlay protocol within that target ring.

Otherwise, the node needs to locate a gateway node to the target ring, which is accomplished via anycast. If the node is a member of the global ring, it then forwards the message via anycast in the global ring to the group that corresponds to the desired ringId. The message will be delivered to a gateway node for the target ring that is close in the physical network, among all such gateway nodes. This gateway node then forwards the data into the target ring, and routing proceeds as before.

If the node is not a member of the global ring, then it forwards the message into the global ring via a gateway node by anycasting to the group whose identifier corresponds to the ringId of the global ring. Routing then proceeds as described above.

As an optimization, it is possible for nodes to cache the IP addresses of gateway nodes they have previously obtained. Should the cached information prove stale, a new gateway node can be located via anycast. This optimization drastically reduces the need for anycast messages during routing.

### 3.5 Global lookup

In the previous discussion, we assumed that messages carry both a key and the ringId of the ring in which the key is stored. In practice, however, applications often

wish to look up a key without knowledge of where the key is stored. For instance, keys are often derived from the hash of a textual name provided by a human user. In this case, the ring in which the key is stored may be unknown.

The following mechanism is designed to enable the global lookup of keys. When a key is inserted into a organizational ring and that key should be visible at global scope, a special indirection record is inserted into the global ring that associates the key with the ringId(s) of the organizational ring(s) where (replicas of) the key is(are) stored. The ringId(s) of a key can now be looked up in the global ring. Note that indirection records are the only data that should be stored in the global ring. Only legitimate indirection records are accepted by members of the global ring to prevent space-filling attacks.

### 3.6 Multi-level ring hierarchies

We believe that a two-level ring hierarchy is sufficient in the majority of cases. Nevertheless, there may be situations where more levels of hierarchy are useful. For instance, a world-wide organization with multiple campuses may wish to create multiple rings for each of its locations in order to achieve more fine-grained content locality. In these cases, it may be advantageous to group these machines into subrings of the organization's ring, further scoping content and path locality.

In order to provide for such extensions, the ring hierarchy described above can be naturally extended. To do so, we view ringIds as a sequence of digits in a configurable

base  $b$ , and each level of ring hierarchy will append an extra digit onto the parent ring's ringId. Thus, organizations which own a given ringId can dynamically create new rings by appending digits to their ringId.

The routing algorithm can be generalized to work in a multi-level hierarchy as follows. When routing to a remote ring  $R$ , the node first checks to see if it is a member of  $R$ . If so, it simply routes the message in  $R$  using the normal overlay routing.

If the node is not a member of  $R$ , it must forward the message to a gateway. If the node is a member of multiple rings, it must choose one of these rings in which to forward the message. This is done by comparing the shared prefix length of each local ringId and  $R$  and picking the ring with the longest shared prefix. In the case of multiple ringIds with the longest prefix, the node should pick the shortest one in total length. This process guarantees that the node picks the local ring which is “closest” to the destination ring  $R$ .

Once the node has chosen in which local ring  $L$  to send the message, it must determine if it should route the message up (towards the global ring), or down. This is an easy computation, as it is dependent only upon the length of the shared prefix of  $L$  and  $R$ . If  $R$  has  $L$  as a prefix, the node should route the message downwards since  $R$  is “below” this ring. Thus, the node should forward the message via an anycast to the Scribe group rooted at  $substring(R, length(L) + 1)$ . The gateway node which

receives the message can then use the routing algorithm again in the other ring.

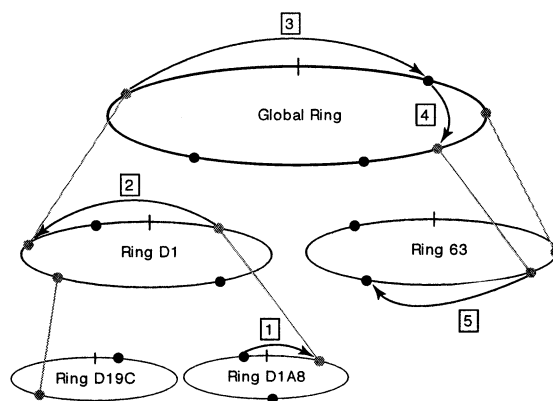
```

(1)  route(dst, msg) {
(2)    if (local == dst) {
(3)      route_normally(msg)
(4)    } else {
(5)      len = length(local)
(6)
(7)      if (dst.hasPrefix(local))
(8)        forward(substring(dst, len+1), msg)
(9)      else
(10)       forward(substring(local, len-1), msg)
(11)    }
(12) }
```

**Figure 3.3** Pseudocode for routing between rings.

If  $R$  does not have  $L$  as a prefix, the node should route the message upwards, towards the global ring. This is done by routing the message to the parent ring, or to a ring with ringId  $substring(L, length(L) - 1)$ . Clearly, messages are routed efficiently by forwarding the message until a ring is found whose id is a prefix of the destination ring, and then routing the message downwards towards the destination ring.

The pseudo-code for routing a message  $msg$  to the ringId  $dst$  at a node in ringId  $local$  is shown in Figure 3.3. Figure 3.4, below, shows an example a node in ring  $D1A8$  routing to a location in the ring  $63$ . In the figure, gray nodes are gateways, which exist in multiple rings and route between them and the numbers 1-5 denote the steps in routing.



**Figure 3.4** Diagram of a the routing process with multiple levels of hierarchy.

## Chapter 4

# POST Design

POST provides three generic services: (i) a shared, secure single-copy message store, (ii) metadata based on single-writer logs, and (iii) event notification. These services can be combined to implement a variety of collaborative applications, like email, news, instant messaging, shared calendars and whiteboards.

In a typical pattern of use, users create messages that are inserted in encrypted form into the secure store. To send a message to another user or group, the event notification service is used to provide the recipient(s) with the necessary information to locate and decrypt the message. The recipients may then modify their personal, application-specific metadata to incorporate the message into their view (e.g., into a private mail folder, a shared bulletin board, or a calendar view).

POST assumes the existence of a certificate authority. This authority signs identity certificates binding a user's unique name (e.g., her email address) to her public key. The same authority issues the `nodeId` certificates required for secure routing in Pastry [7]. Users can access the system from any participating node, but it is assumed that the user trusts her local node, hereafter referred to as the trusted node, with her private key.

Figure 4 shows psuedocode detailing the POST API which is presented to ap-



plications. The `store` and `fetch` methods comprise the single-copy message store. Similarly, the `readTopEntry`, `readPreviousEntry`, and `writeEntry` methods provide the metadata service, and the `notify` method comprises the event notification service.

The most complicated of these APIs is the metadata service, and we describe it in more detail here. Each of the user's logs is given a name unique to the user, denoted below by `LogName`. Applications can scan through a log in reverse order by first calling `readTopEntry`, followed by successive invocations of `readPreviousEntry`. Similarly, applications can write to the log by simply calling `writeLog` with the desired target log's name.

```
// these two methods provide the single-copy message store
Handle store(Object)
Object fetch(Handle)

// and these methods provide metadata service
LogEntry readTopEntry(LogName)
LogEntry readPreviousEntry(LogEntry)
void writeEntry(LogName, LogEntry)

// lastly, this method provides the notification service
void notify(User, Message)
```

Table 4.1 POST API

## 4.1 User accounts

Each user in the POST system possesses an account, which is associated with an identity certificate. The certificate is stored as a certificate block, using the secure

hash of the user's name as the handle. Also associated with each account is a user identity block, which contains a description of the user, the contact address of the user's current trusted node, and any references to public metadata associated with the account. The identity block is stored as a public-key block, and signed with the user's private key. Finally, each user account has an associated Scribe group used for event notification, with a groupId equal to the cryptographic hash of the user's public key.

The immutable identity certificate, combined with the mutable public-key block, provides a secure means for a trusted authority to bind names to keys, while giving users the ability to change their personal contact data without requiring subsequent interactions with the certificate authority. The Scribe group allows anybody waiting for news from that user, or anybody wishing to notify the user that new data is available, to have a common rendezvous point.

## 4.2 Single-copy store

POST stores potentially sensitive user data on nodes throughout the network. While we could require application-layer cryptography, such as S/MIME or PGP for email, we wish to provide an expectation of privacy comparable to maintaining data purely on a local disk. POST uses a technique called convergent encryption [17], which allows a message to be disclosed to selected recipients, while ensuring that copies of a given plaintext message inserted by different users or different applications map to

the same ciphertext, thus requiring only a single copy of the message to be stored.

When an application wishes to store message  $X$ , POST first computes the cryptographic  $Hash(X)$ , uses this hash as a key to encrypt  $X$  using an efficient symmetric cipher, and then stores the resulting ciphertext with the handle

$$Hash\left(Encrypt_{Hash(X)}(X)\right)$$

which is the secure hash of the ciphertext. To decrypt the message, a user must know the hash of the plaintext.

Convergent encryption reduces the storage requirements when multiple copies of the same content tend to be inserted into the store independently. This happens commonly in cooperative applications, for example, when a give popular document is sent as an email attachment or posted on bulletin boards by different users.

In certain scenarios, however, it may be undesirable to use convergent encryption, as is discussed in Section 4.6.2. In these cases, the use of convergent encryption is completely optional and the single-copy store can be set to use a more traditional encryption function, such as AES with randomly generated keys. The single-copy property of the store is lost though, as inserts of the same data by different users will no longer produce the same ciphertext. If this can be tolerated by the users of the system, the rest of POST still works as expected.

### 4.3 Event notification

The notification service is used to alert users and groups of users to certain events, such as the availability of a message, a change in the state of a user, or a change in the state of a shared object.

For instance, after a new message was inserted into POST as part of an email or newsgroup, the intended receiver(s) must be alerted to the availability of the message and provided with the appropriate decryption key. Commonly, this type of notification requires obtaining the contact address from the recipient's identity block. (This may require a lookup of the recipient's certificate block, if the certificate is not already cached by the sender). Then, a notification message is sent to the recipient's contact address, containing the secure hash of the message's ciphertext and its decryption key, encrypted with the recipient's public key and signed by the sender.

In practice, the notification can be more complicated if the sender and the recipient are not on-line at the same time. To handle this case, the sender may delegate the responsibility of delivering the notification message to a set of  $k$  random nodes. When a user  $A$  wishes to send a notification message to a user  $B$  whose trusted node is off-line,  $A$  first sends a notification request message to the  $k$  nodes numerically closest to a random Pastry key  $C$ . This message is encrypted for  $B$ , and separately contains  $A$ 's signature indicating the message is valid. The  $k$  nodes are then responsible for delivering the notification message (contained within the notification request message)

to  $B$ . Each of these nodes stores the message and then subscribes to the Scribe group rooted at the hash of  $B$ 's public key. Additionally, the nodes periodically check the recipient's identity block for an updated contact address, and ping the address.

Whenever user  $B$  is on-line, his trusted node periodically publishes a message to the Scribe group rooted at the hash of his public key, notifying any subscribers of his presence and current contact address. This presence message may contain additional application-specific data about the state of the user. Upon receipt of this message, subscribers deliver the notification by sending it to the contact address. Since, by assumption, at most  $k - 1$  of these nodes can be faulty, the notification is guaranteed to be delivered. POST relies on Scribe only for timely delivery – if Scribe messages are lost due to failures, the notification will eventually be delivered due to periodic pings and checks of the recipient's identity block.

To guarantee confidentiality, each notification message is encrypted using a symmetric cipher such as AES with a unique session key, and the session key itself is then encrypted using the recipient's public key. Thus, only the recipient can decrypt the session key (i.e., with his private key) in order to decrypt the remainder of the message. Each notification message is also signed with the sender's private key, allowing the recipient to verify its authenticity. Finally, each notification message also includes a timestamp to prevent the message from being replayed by malicious users. Note that, unlike most traditional user messaging infrastructures, everything in POST is

digitally signed and encrypted, by default, from the ground up. This proves useful when implementing secure higher-level services like email, instant messaging, and so forth.

## 4.4 Metadata

POST provides single-writer logs that allow applications to maintain metadata. Typically, a log encodes a view of a specific user or group of users and refers to stored messages. For instance, a log may represent updates to a user's private email folder, or the history of a public newsgroup. An email or news application would then use a log consisting of insert, update, and delete records to keep track of the state of the folder or newsgroup.

In general, logs can be used to track the state of a chatroom, a newsgroup, a shared calendar, or an arbitrary data structure. POST represents logs using self-authenticating blocks. This is similar to, and was inspired by, the logs used in the Ivy p2p filesystem [33].

The log head is stored as a public-key block and contains the location of the most recent log record. Handles for log heads may be stored in the user's identity block, in a log record, or in a message. Each log record is stored as a content-hash block and contains application-specific metadata and the handle of the next recent record in the log. Applications optionally encrypt the contents of log records depending on the intended set of readers.

In the original implementation used in Ivy, the log head and each log record are stored at a different set of nodes. To allow for more efficient log traversal, POST aggregates together clusters of  $M$  consecutive log records in a single PAST object. Partially filled clusters are buffered in the log head object itself, and are actually pushed out as single objects once they are full. This reduces the number of keys in PAST by a factor of  $M$ , which is a significant savings in terms of overhead.

Other optimizations are provided to reduce the overhead of log traversals, including caching of log records at clients and the use of snapshots. Like Ivy, POST applications may periodically insert snapshots of their metadata into the store. In this case, log traversals always terminate at the most recent snapshot.

Single-writer logs are the only mechanism used to maintain mutable state in POST. Their use avoids the cost and complexity of a general Byzantine fault-tolerant replicated state machine. As we will show, POST's restricted mechanism for mutable state is flexible and efficient enough for a variety of collaborative applications including email, instant messaging, shared calendars and bulletin boards.

## 4.5 Garbage collection

In order to make a DHT practical for use in a mission-critical application, we found it necessary to introduce a mechanism for removing objects from the DHT. While the rapid growth in hard disk size may make storing all inserted data *ad infinitum* possible in terms of storage overhead, the extra maintenance overhead quickly becomes

overwhelming. This maintenance overhead includes ensuring that there always are  $k$  live replicas of each stored object, and re-replicating each object as necessary.

The obvious solution in this case is to simply add a `delete` operation to PAST which will remove the object associated with the given key. However, this is not a realistic solution for a number of reasons. First, a delete method is unsafe under most circumstances when arbitrary nodes must be able to delete any object in the DHT. In this scenario, a single compromised node could issue delete commands for all objects in the DHT, removing the all of the data. Second, even if the assumption is made that all nodes are trusted, multiple users may be interested in a given object. A complicated system such as a reference-counting scheme is necessary in order to make sure that an object is only deleted once all users are no longer interested.

As an alternative solution, POST is based on a lease-based version of PAST. Each object inserted into the DHT is given a lifetime by the inserting node. Once the expiration time for a given object has passed, the replica nodes have completed their storage contract and are free to delete the object. Clients are also allowed to extend the lifetime of existing objects; the users must periodically do this to all data which the user is still interested in. The modified PAST API is shown in Table 4.5.

<pre>void put(Key, Object, Expiration) Object get(Key) void refresh(Key, Expiration)</pre>
--

**Table 4.2** Modified PAST API



Other slight modifications are necessary to PAST in order to fully implement this feature. Specifically, the replication protocol must now exchange tuples (*key*, *expiration*). When a node is told to fetch a key which it already stored with a different lifetime, it simply extends the lifetime of the stored key if the new lifetime is longer.

Additionally, it is unrealistic to assume more than a loose synchronization between the clocks at each storage node. Therefore, expired objects are not deleted immediately; instead, they are kept for an additional *grace period*  $T_G$ . During this time, the objects are still available for queries, but they are no longer advertised to other nodes during maintenance. Thus, nodes that have already deleted their objects do not attempt to recover them.

## 4.6 POST Security

POST must be designed to face a variety of threats, ranging from nodes that simply fail to operate to attackers determined to read or modify sensitive information. POST must likewise be robust against freeloading behavior, including users consuming more resources than they contribute, and to application-specific resource consumption issues, such as the space consumed by spam.

#### 4.6.1 Threat model

Our threat model for POST consists of attacks from both within and outside of POST. Internal attacks can be broken down into two classes: freeloading and malicious behavior. Freeloading, discussed below in Section 4.6.5, consists of either selfish behavior or simple denial of service. Malicious behavior, however, can consist of nodes attempting to read confidential data, modify existing data, or delete data from the ePOST system. How POST handles these attacks is described below in the following sections.

Additionally, we have not made specific efforts to make POST immune to traffic analysis attacks. For example, nodes may be able to learn whether or not two users are sending messages back and forth, although they are not able to discover the content of these messages. Additionally, attackers may be able to determine if a given user is reading a certain ciphertext by overhearing a DHT request. However, we do not believe that these traffic analysis attacks are a large threat, as they require a local attacker and current collaborative systems are just as vulnerable.

#### 4.6.2 Data privacy

While convergent encryption provides the benefit of a single-copy store, it is known to be vulnerable to known plaintext attacks. An attacker who is able to guess that plaintext of a message can verify its existence in the store, and may or may not be able

to determine whether any given node has requested that particular message. This may be a particular concern for short messages, messages that are highly structured, or generally any messages with low entropy. To partially address these concerns, POST uses traditional cryptographic techniques (such as simple AES encryption with a random key), to protect data that is never intended to be shared, such as the logs and other metadata maintained for every user of the system.

#### 4.6.3 Data integrity

The single-writer property and the content-hash chaining [31] of the logs make it computationally infeasible for a malicious user or storage node to insert a new log record or to modify an existing log record without the change being detected. This is due to the choice of a collision-resistant secure hash function for the log entries and the use of signatures based on public key encryption in the log heads.

To prevent version rollback attacks by malicious storage nodes, public-key blocks contain version timestamps. When reading a public-key block (e.g., a log-head) from the store, it is necessary to read all  $k$  replicas of the log, and use the authentic replica with the most recent timestamp. When reading content-hash blocks or certificate blocks, it is sufficient to use any authentic replica.

#### 4.6.4 Data durability

Of great concern is the durability of stored messages. For a message to become unavailable, every member of the message's replica nodes must independently fail before the object can be re-replicated. The use of local rings allows more local replication, increasing an administrator's control over the replica nodes and also reducing the latency for fetching a given document.

Organizations that run a local ring should ensure that nodes are spread over different buildings, if not different locations. Furthermore, correlated failures may be caused by viruses, worms, or other attacks that take advantage of most organization's monoculture approach to systems administration. In addition to using a variety of host platforms, administrators can run the POST daemon with reduced system privileges under its own user id, partially isolating the POST daemon and its file store from other issues that may effect any given system. Likewise, if the POST daemon itself were compromised, its effect on the rest of the system could be more easily contained.

In order to provide data durability even in the face of a massively correlated failure, POST employs the distributed data-backup system Glacier [23]. Glacier provides data durability by erasure encoding objects and efficiently storing and maintaining object fragments. Glacier has been shown to provide 99.9999% durability even under an 60% correlated failure with modest overhead in both bandwidth and storage. Glacier

is discussed in more detail in Chapter 6.

#### 4.6.5 Denial of service

A variety of denial of service (DoS) attacks may be attempted against p2p networks. A common DoS strategy might be to control enough nodes to effectively partition the overlay network, or even to control all of the outgoing routes from a given node. Likewise, DoS attacks may be aimed at controlling all of the replicas of a given document, allowing the attacker to effectively censor any desired document. Pastry's secure routing mechanism provide an effective defense against DoS attacks, both from within and outside the overlay [7]. When using secure routing, an attacker would need to control over 25% of the overlay nodes to mount an effective DoS attack.

#### 4.6.6 Freeloading

Freeloading is a similarly pressing concern in p2p networks. Nodes within the network may wish to consume more remote storage than they provide to the network. Likewise, nodes may wish to fetch objects more often than they serve objects to other nodes. If bandwidth or storage are a scarce resource, users will have an incentive to modify their POST software to behave selfishly. Nodes can generally be coerced into behaving correctly when other nodes observe their behavior and, if they determine a node to be a freeloader, will refuse to give it service [34, 12]. Such mechanisms can guarantee that it is rational for nodes to behave correctly. On the other hand,

when nodes stop servicing their peers because, for whatever reason, they would find it undesirable, this can have a negative effect on the integrity and durability of the system.

POST, in its present form, does not include any mechanisms which make it immediately compatible with existing incentives-based systems [34, 12]. Instead, our current focus is taking advantage of the local ring, which resides within a single administrative domain. By scoping the rings in this manner, abuses of the POST ring can be detected and corrected completely within a single organization. For example, organizations can periodically monitor the POST ring and if malicious behavior is detected, they are able to revoke the user's access since the admission policy is specific to the organization. If the network were simply one large global ring, it would be non-trivial to track down an abuser or freeloader, much less correct the malicious behavior.

## Chapter 5

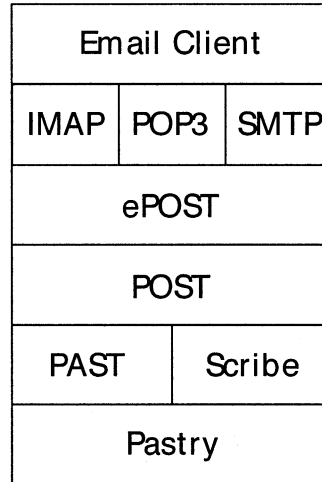
# ePOST Design

In this chapter, we describe the design of a serverless email system, ePOST, on top of the POST infrastructure. The goal is to show how POST can support a secure, scalable and highly resilient email system that leverages the resources of participating desktop computers.

While a system like ePOST promises increased resilience, greater scalability and lower cost, it remains an open question whether these advantages will be sufficient to completely displace the existing, server-based email infrastructure. Nevertheless, we chose to pursue ePOST for several reasons. First, ePOST is designed so that it can be deployed incrementally, thus allowing individual organizations to adopt it while still relying on existing standards and infrastructure for communication across organizations. Second, unlike most existing p2p applications, email is mission-critical and demands high reliability, security, and availability. Thus, it is a challenging driver for the development of POST and, more generally, the underlying p2p infrastructure.

Each ePOST user is expected to run a daemon program on his desktop computer that implements ePOST, and contributes some CPU, network bandwidth and disk storage to the system. The daemon also acts as a SMTP and IMAP server, thus allowing the user to utilize conventional email client programs. The daemon is assumed

to be trusted by the user and holds the user's private key. No other participating nodes in the system are assumed to be trusted by the user. An outline of the hierarchy used in ePOST is shown in Figure 5.1.



**Figure 5.1** ePOST Stack

## 5.1 Email storage

In ePOST, email messages received from an email client program are parsed and the MIME components of the message (message body and any attachments) are stored as separate objects in POST's single-copy store. Thus, frequently circulated attachments are stored in the system only once.

The message components are first inserted into POST by the sender's ePOST daemon; then, a notification message is sent to the recipient. Sending a message or attachment to a large number of recipients requires very little additional storage overhead beyond sending to a single recipient, as the data is only inserted once.



Additionally, if messages are forwarded or sent by different users, the original message data does not need to be stored again; the message reference is reused.

The convergent encryption used in POST is known to be less secure when encrypting short messages and highly structured content (e.g., text), as it is vulnerable to known plaintext attacks (see Section 4.6.2). To avoid a loss of confidentiality, small message bodies inserted by ePOST are inserted into the store using a normal encryption function with a random key, as is discussed in Section 4.2. This measure defeats the single-copy storage, but this is not a concern given the small average size of message bodies.

## 5.2 Email delivery

The delivery of new email is accomplished using POST's notification service. A sender first constructs a notification message containing basic header information, such as the names of the sender and recipients, a timestamp, and a reference to the body and attachments of the message. The sender then requests the local POST service to deliver this notification to each of the recipients. This message is signed by the sender and encrypted using the receiver's public key in the usual fashion, combining RSA public key cryptography with a fast symmetric cipher like AES.

If the recipient of an email is in a different ring than the sender, the recipient has the option of referencing the received email body and attachments in the ring of their originator, or to fetch and insert copies into his own local ring. The latter approach

leads to higher availability and greater confidence in message durability, due to the greater replication and confidence in the recipient's local ring. Therefore, our ePOST implementation replicates all incoming mail in the recipient's local ring by default.

### 5.3 Email folders

Each mail folder is represented by an encrypted POST log. Each log entry represents a change to the state of the associated folder, such as the addition or deletion of a message. Furthermore, since the log can only be written by its owner and its content are encrypted, ePOST preserves the expected privacy and integrity semantics of current email systems with storage on trusted servers.

Next, we describe a log record representing an insertion of a email message into a user's folder, such as her inbox. Other types of log records are analogous. An email insertion record contains the content of the message's MIME header, the message's handle and its decryption key, and a signature of all this information, taken from the sender's original notification message. All of this data is then encrypted with a unique session key, using a low-cost symmetric cipher like AES. As these insertion records need only be legible to the original sender, the session key is encrypted with a master key, also using the cheap symmetric cipher. This symmetric master key is maintained with the same care as the user's private key. This allows the owner of the folder, and none other, to read messages in the inbox and verify their authenticity without performing expensive public key operations. The exact messages are shown

$$\begin{aligned}
\textit{EncryptedEmail} &= \textit{Encrypt}_{\textit{Hash}(X)}(X) \\
\textit{MessageHeader} &= (A, B, T, \textit{Hash}(\textit{EncryptedEmail}), \textit{Hash}(X)) \\
\textit{Notification} &= \textit{Encrypt}_{K_B}(\textit{MessageHeader}, \textit{Sign}_{K_A}(\textit{MessageHeader}))
\end{aligned}$$

**Figure 5.2** Messages transmitted sending an email.

in Figure 5.2.

## 5.4 Incremental Deployment

In this section, we discuss integration issues in the specific context of ePOST. To allow an organization to adopt ePOST as its email infrastructure, ePOST must be able to interoperate with the existing, server-based email infrastructure. We describe here how ePOST is deployed in a single organization and interoperate with email services in the general Internet.

For inbound email, the organization's DNS server provides MX records referring to a set of trusted POST nodes within the local organization. These nodes act as incoming SMTP mail gateways, accepting messages, inserting them into POST, and notifying the recipient's node. Suitable headers are generated such that the receiver is aware the message may have been transmitted on the Internet unencrypted. If no identity block can be found for the recipient in the local ring, then the email "bounces" as in server-based systems.

Sending email to the outside world first requires determining that the desired email

address is not already available inside ePOST. At that point, there may be a gateway service that can provide the appropriate certificate material to generate a standard cryptographic email in S/MIME or PGP format. This encryption is performed in the sending user's trusted local node, before the data goes onto the network. If the recipient does not support secure email, then the email must ultimately be transmitted in the clear, so the ePOST proxy server can speak regular SMTP to the recipient's mail server.

The inbound proxy nodes need to be trusted to the extent that they receive plaintext email messages for local users. Typically, the desktop workstations of an organization's system administrators can be used for this purpose. These administrators own root passwords that allows them to access incoming email in conventional, server-based systems. Thus, ePOST provides the same privacy for incoming email from non-POST senders as existing systems, and provides stronger confidentiality for email transmitted within ePOST.

## 5.5 Management

If ePOST is to replace existing reliable email systems, there must be a viable management strategy for organizations to adopt when deploying ePOST. The management tasks in ePOST can be broken down into three categories: software distribution, storage, and access. In the paragraphs below, we discuss these tasks in detail and show how they can be minimized in the context of ePOST.

### 5.5.1 Software

The first management task incurred with ePOST is maintaining the proxy software which runs on users desktops. This software will need to be kept running and up-to-date as bugs are fixed and features are added. Organizations can do this in a straightforward manner by including the proxy as part of their standard machine images. The ePOST proxy can be configured as a service which runs and is restarted if it fails. Including the software in the images also allows the software to be pre-configured for the organization. Additionally, upgrades can be handled by signing updated code and having users' proxies periodically check and download authentic updates.

To allow administrators to efficiently monitor the ePOST software, a monitoring application can easily be built. Such an application can run in the background and periodically check the status of all of the member nodes. Any error conditions or unusual behavior can be forwarded to an administrator who is able to take the appropriate action. In fact, we built such a monitoring application on top of our ePOST implementation (described in Chapter 6) to aid us in monitoring and debugging.

### 5.5.2 Storage

In a distributed storage system such as ePOST, certain management overhead is necessary to monitor the storage pool. For example, administrators need to ensure

that space-filling attacks are not taking place and that nodes which are running out of disk space are promptly serviced. Such monitoring can easily be done automatically using the monitoring tool described in the above section. Machines which are close to their disk space limit can be forwarded to the administrators, who can then service the machines.

### 5.5.3 Access

Controlling access to ePOST can be broken down into two related tasks: trust and naming. Trust is based on organization-provided certificates, as each user must obtain a certificate to participate in the system. This is no different from current email systems, where each user is required to obtain an account on an email server, and it can be accomplished in a similar manner. For example, in our experimental deployment, we have simply provided a web page where users can sign up and download certificates. In more realistic settings, however, the process could require approval by various administrators before the new certificate is produced.

Naming in ePOST can also be accomplished in a manner similar to current systems. Organizations only need to ensure that each email address is only bound to one public key, meaning that each email address is only give to one user. This is easy to accomplish, since each user must obtain a certificate from their organization anyway. Similar to current systems, the organization can simply keep a list of all assigned email addresses and reject any applications for names which already exist.

In summary, the overhead of managing a ePOST system is comparable to current systems. Administrators are required to set up mechanisms for access and trust, as well as monitor the running software and storage systems. However, in the case of ePOST, the overhead has the potential to be lower, since the self-organizing properties of the underlying peer-to-peer substrate can mask the effect of node failures. Additionally, the organic scalability granted to ePOST from the overlay has the potential to significantly reduce the overhead associated with scaling an existing email server to more users.

## 5.6 Discussion

By default, ePOST provides strong confidentiality, authentication and message integrity. The system is able to tolerate up to  $k - 1$  faulty nodes with Byzantine faults in any replica set of  $k$  POST nodes without loss of data or service, where  $k$  is the degree of message replication. It relies on Pastry's secure routing facilities [7], data replication, and cryptographic techniques to achieve robustness under a wide range of attacks, including denial-of-service, and Byzantine failures.

### 5.6.1 Feasibility

More analysis and experimentation will be necessary to determine appropriate assumptions about the fraction of faulty nodes in various environments, and appropriate levels of replication. However, results of a prior study on p2p filesystems in

corporate environments indicate that modest levels of replication can yield very high availability [6]. At the same time, our own results indicate that even relatively high degrees of replication (e.g.,  $k = 10$ ) are possible within the disk space budget provided by ordinary desktop computers. Furthermore, the use of Glacier [23] allows for less replication, since full-object replication is only necessary for efficient short-term availability. If the data is lost due to a correlated failure, it can simply be reconstituted from the Glacier fragments.

Since ePOST inserts all incoming messages into the local ring, only the node failure probability and failure independence within a user's local ring determines the durability of the messages that the user references. Therefore, a user's organization can take appropriate steps to ensure failure independence and determine a degree of replication commensurate with the expected node failure probability within the organization, as discussed in Section 4.6.

### 5.6.2 Mailing Lists

Mailing lists are supported in POST by maintaining the list as an additional log and storing the log head reference at the list maintainer's user identity block. Only the maintainer is allowed to modify the membership. When delivering a message, the sender notices the list and expands the recipient list appropriately.



### 5.6.3 Spam

Spam can be seen as an additional form of freeloading. Spam may potentially originate from within ePOST or may be injected by a mail gateway. If spam were to originate purely within ePOST, then the spam messages would need to be stored and maintained in the *sender's* local ring. Only notifications would be transmitted to the recipient, saving the recipient the bandwidth cost of downloading an entire spam message. This makes it easier to track the origin of a spam message and to punish the origin. Likewise, because all of the message copies are within a single local ring, it becomes easier to delete the spam after it has been detected. ePOST gives the recipient the option to make a local copy of a message, but this would clearly not be done for spam. Furthermore, if an incentive-based storage mechanism is being used [34], then one of the major goals of anti-spam researchers, to push the costs of spam back onto the spammers, can be achieved in a straightforward manner.

If spam originates from a mail gateway, it gets mixed in with other messages in the local ring. Lately, spammers have been customizing their messages for every recipient, meaning that techniques like convergent encryption will be unlikely to collapse multiple spam messages to a single POST object. To deal with spam coming through the gateway, ePOST offers no new mechanisms, but ePOST does not preclude the full variety of spam filtering techniques already in use.

Additional spam prevention techniques are possible when using ePOST. For ex-

ample, users can easily share a 'Junk' folder containing all of their received spam, which can be used by a Bayesian filter as a training set. However, such collaborative spam filtering systems open the door for additional security problems, such as users maliciously marking non-spam as spam in hopes of polluting the spam archive. For these reasons and others, the design of a collaborative spam filter is part of our future work.

## Chapter 6

# Evaluation

In this section, we present results of our deployment of POST and ePOST within the Computer Science department at Rice University. We implemented a version of POST and ePOST on top of FreePastry [20], an open-source implementation of Pastry, PAST and Scribe, and the POST and ePOST code will be released with FreePastry 1.4.

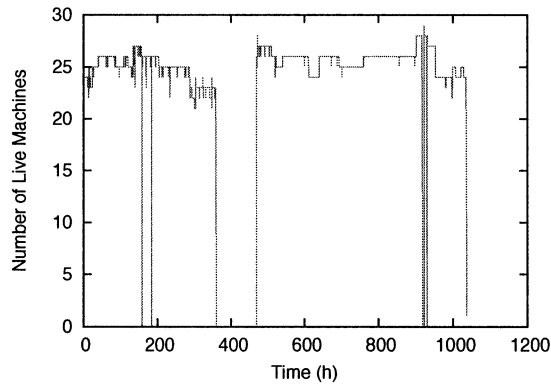
### 6.1 Deployment Setup

Our initial deployment of ePOST began in January of 2004 with very few users. As confidence in the system grew, we expanded our userbase and incorporated new features. The results presented in this section reflect only the latest statistics from the deployment - data from before these results does not reflect the same setup, so it is omitted for clarity.

#### 6.1.1 Timeline

Each POST node in the system ran a Pastry node, with PAST, Scribe, POST, and ePOST services, as well as IMAP, POP3, and SMTP servers for the local user. We started the experiment on September 19, 2004, and the results presented below cover data until November 12, 2004, a span of 53 days. The ring consisted of on average 26 nodes, running various versions of Linux and Windows.

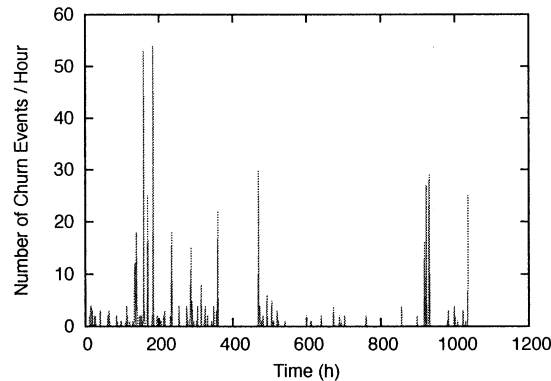
Also, the ring was restarted a few times throughout the deployment in order to fix bugs and add additional features. These restart events occurred three times throughout the trace, on October 6, 2004; October 7, 2004; and November 6, 2004. Additionally, the ring was taken down for maintenance from October 14, 2004 through October 18, 2004. All of these events are shown on Figure 6.1 below, which contains graph of the number of live machines in the POST ring over the 53 day trace.



**Figure 6.1** Number of participating machines

As is stated before in this paper, the target environment of POST and ePOST is a large organization. Correspondingly, we expect that such an environment will have a much lower churn rate than that of many of the common p2p file-sharing utilities. The churn rate for our POST and ePOST deployment is shown in Figure 6.2, as the number of join and leave events per hour. As can be seen, the network was relatively stable, but there were a few times where a significant portion of the ring was under churn. Even under fairly heavy churn, the network managed to recover and continue operation and while we don't expect this to happen frequently, POST should be able

to degrade gracefully.



**Figure 6.2** Amount of churn in the ePOST ring.

### 6.1.2 PAST

The results presented reflect two different parameters for the ePOST ring. From September 19, 2004 until October 14, 2004, the ring was run with a very aggressive garbage collection lease of just 4 days, which was done in order to simulate multiple rounds of object lifetimes. The ring was then taken down through October 18, 2004, when it was restarted with a more realistic garbage collection lease of 30 days. Additionally, the number of primary PAST replicas was 4 in the former deployment, and 3 in the latter - this was reduced once we gained confidence in Glacier for durability.

PAST requires that a periodic protocol be run to ensure that  $k$  replicas exist for every object. Due to our expected low rate of churn, we have liberally set the protocol frequency is set to every 10 minutes. As an optimization, we use Bloom filters to exchange lists of keys which guarantees eventual consistency. Additionally,

in order to provide for easy upgrades and disaster recovery, data was stored on-disk in a GZipped XML format. This represented an approximately 30% increase over the storage requirements for a simple byte stream, and this overhead is reflected in these results.

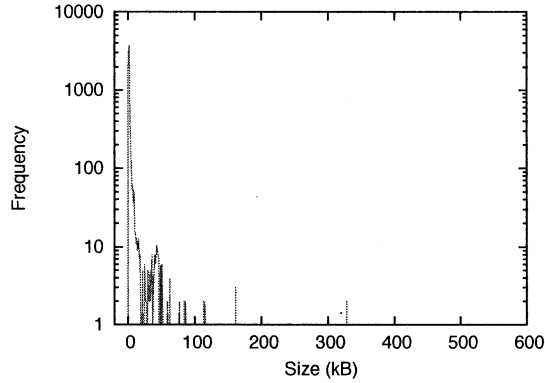
ePOST uses Glacier in order to provide durable storage. Glacier is configured to provide 99.9999% reliability under a 60% correlated failure, resulting in an approximately 10-fold increase in the storage overhead. Additionally, since the ePOST workload mainly consists of small objects (such as email headers and bodies), Glacier is configured to aggregate such small objects into larger *aggregates*, which reduces the overhead. These aggregates hold, on average, around 25 objects and are pushed out minimally every 12 hours.

### 6.1.3 ePOST

In our deployment, 4 machines served as SMTP gateways, accepting normal SMTP traffic for ePOST users. Additionally, we provided 6 other “seed” machines which participated in the ePOST ring but were not proxies for any user. The ePOST ring was used by 4 *active* users, who used ePOST as their primary method of accessing email. The ring was also used by 12 *passive* users, who forwarded their mail into ePOST and had a running proxy, but did not use ePOST as their primary email system.

Throughout the deployment, the ePOST ring delivered 31,094 individual email

messages, which translates to 1.8 messages per user per hour. The sizes of the emails were highly bimodal - the vast majority of the messages were under 10 KB, while a very small number of messages ranged up to 5 MB in size. Figure 6.3 shows a histogram of the distribution of email sizes.



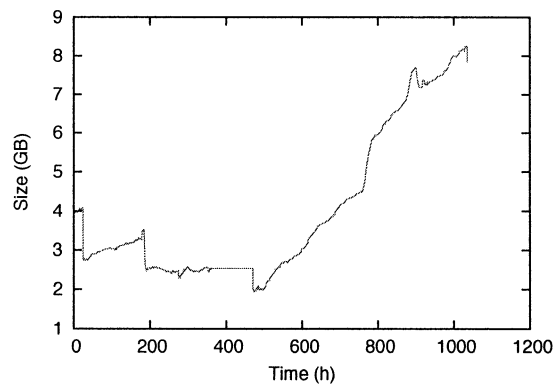
**Figure 6.3** Histogram of email sizes

## 6.2 Data Storage

In this section, we present results concerning the storage load on the member nodes in the ePOST ring. Figure 6.4 shows a graph of the total data stored in the ePOST ring, including the overhead from the replication in both PAST and Glacier. This graph shows the storage increasing slowly during the first phase of the experiment, until 375 hours, and then increasing at a faster rate once the ring is restarted at 450 hours. This difference is the effect of garbage collection: during the former time-span, objects were leased for only 4 days, so deleted objects were quickly collected. However, in the later time-span, the lease was granted for 30 days, which postpones

the effect of garbage collection for 30 days, or longer than the results are shown for.

The rapid increases and drops at 20 hours and 200 hours correspond to bugs in the code which were detected and corrected. Also, the significant drop in the storage requirements at 450 hours is from changing PAST to maintain 3 primary replicas instead of 4, resulting in a 25% savings.



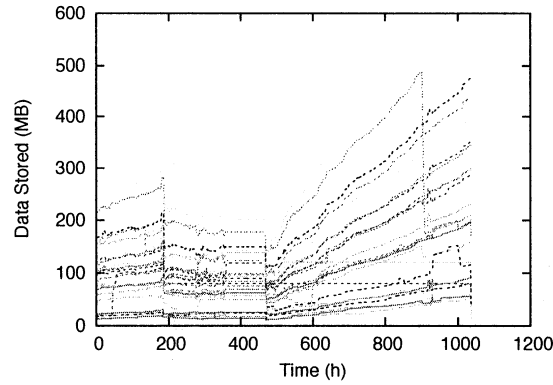
**Figure 6.4** Total data stored in the ring, including PAST and Glacier.

At the end of the experiment, with 26 machines participating in the ring, the average amount of storage required on each node was 303 MB. This is a very modest amount of storage for a commodity desktop, even as this increases it should increase at a rate much slower than the increase in overall hard drive capacity. Furthermore, once garbage collection resumes in the ring, the rate of increase will slow as deleted objects begin to be collected.

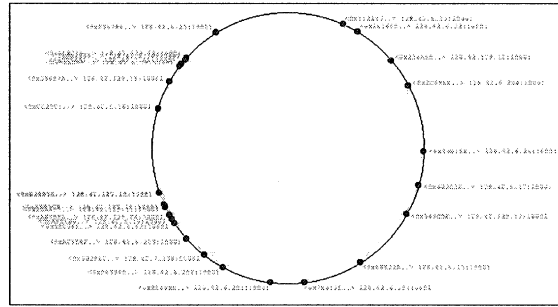
Figure 6.5 shows the same graph as Figure 6.4, but includes separate plots for each of the machines, showing a variance in storage requirements of about one order of magnitude. While machines are storing similar amounts of data, the imbalance



in the amount of storage is caused by random deviations in the nodeIds assigned to the machines. Figure 6.6 shows a plot of the nodeId assignment, which clearly shows the cause of the variance in storage requirements. As more machines are added this imbalance should decrease, as the variance between identifiers will drop.



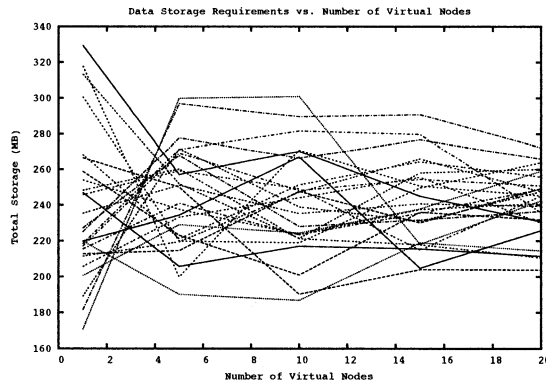
**Figure 6.5** Total data stored on each machine, including PAST and Glacier.



**Figure 6.6** Distribution of nodeIds in experimental ring.

Reducing the variance in the storage requirements can also be done by making each node run  $\log N$  virtual nodes, instead of just one node [45]. Figure 6.7 shows the storage requirements for each physical node as the number of virtual nodes grows. As can be seen from the graph, the variance in storage requirements drops as the number

of virtual nodes increases. However, using more virtual nodes increases the overlay maintenance traffic overhead, so this must be taken into account when deciding on an appropriate number of virtual nodes.

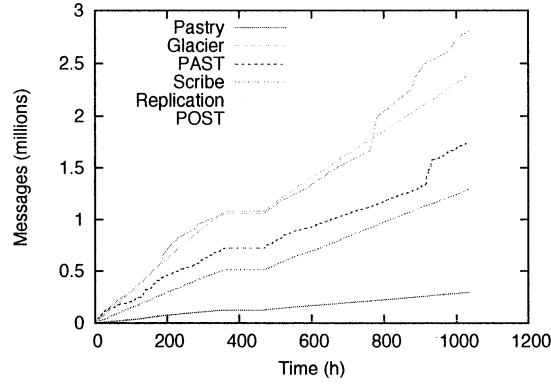


**Figure 6.7** Storage requirements if virtual nodes are used.

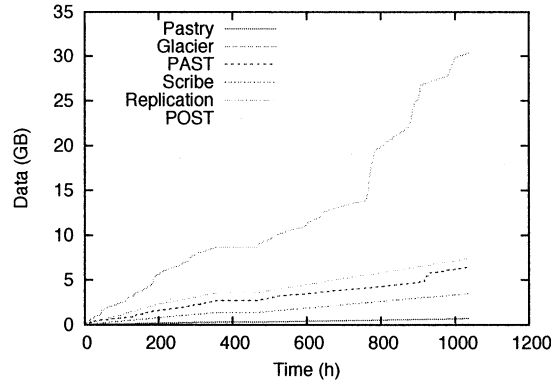
### 6.3 Message Traffic

Next, we take a look at message traffic generated by ePOST. Messages exchanged in Pastry are sent using standard Java serialization, the overhead from which is included in the results below. First, Figure 6.8 shows the cumulative number of messages sent by the various components of ePOST, and Figure 6.9 shows the cumulative number of bytes sent. Unsurprisingly, Glacier was responsible for the majority (62%) of the data sent, however, it did this efficiently, as it only sent 32% of the messages. For comparison, PAST sent only 13% of the bytes, but did this with 20% of the messages.

Overall, the ePOST ring sent a total of 8.5 million messages containing 48GB of



**Figure 6.8** Cumulative number of messages sent by ePOST components.



**Figure 6.9** Cumulative number of bytes sent by ePOST components.

data over the 53 day trace. This works out to an average of 5 messages per node per minute, or 30 KB per node per minute. This is completely acceptable amount of bandwidth, especially when considering that the majority of this bandwidth is internal to an organization.

## 6.4 Single-Copy Store

As mentioned in Section 4.2, ePOST uses PAST to provide a single-copy store to applications. During our experiment, we found that the single-copy store was

able to reduce the storage load by 6.1%. While this amount is not significant when considering the replication required by both PAST and Glacier, we believe that as the system grows, the percentage of overlapping content will also increase.

To test this hypothesis, we collected statistics of the email contained within the email folders of 30 IMAP users in our department. The 30 users volunteered for this study and included students, administrative and research staff, and professors. We recorded the average size, number of attachments, and the number of times a message body or an attachment with the same content appeared in several folders.

The folders contained approximately 300,000 email messages, totally 3.8 GB. Using the POST single-copy message store, this total would be reduced to 3.2 GB of unique data, representing a savings of 15.5%. Thus, we observed that increasing the number of observed users from 16 to 30 gave a 254% increase in the effect of the single-copy store. Moreover, in business environments, users already tend to exchange large attachments more than is likely reflected in our academic department workload.

## 6.5 User-Perceivable Performance

Even though we have demonstrated that ePOST is efficient in terms of bandwidth and storage, arguably the most important metric is the ePOST performance which a user perceives - if ePOST is significantly slower than current centralized systems, many users will not use it. We break the user-perceivable performance metrics into three classes: email delivery time, speed of folder operations, and availability.

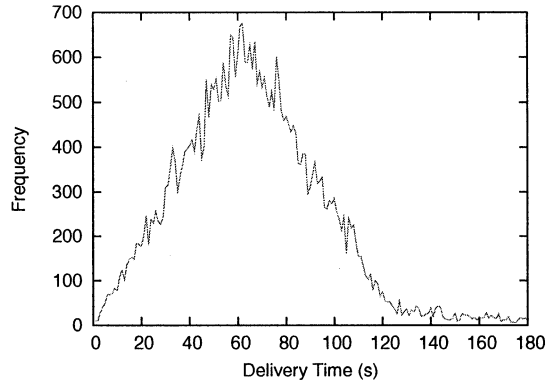
### 6.5.1 Email Delivery Time

We calculate the *delivery time* of an email as the time between when the message is received by an ePOST SMTP server and the time when the recipient's proxy has added the message to the Inbox. In our implementation of ePOST, we did not add the optimization to directly deliver messages to online users. Thus, every email is first handed off to a random set of nodes who then join the recipient's group and eventually deliver the message. The periodic joining of groups by the random nodes is done once per minute, and the periodic publishing to one's group is also done once per minute. Hence, we expect an average duration of 1 to 2 minutes for delivery time with these settings.

Figure 6.10 below shows a histogram of the delivery times for emails during our experiment. In our deployment, over half of the emails were delivered in under 82 seconds and 67% were delivered within 2 minutes. Additionally, since the recipient's proxy may be offline, a number of emails were necessarily postponed until the recipient's proxy came back online. However, in the common case when the recipient was online, the average delivery time was just 66 seconds.

### 6.5.2 Folder Operations

The duration of folder operations is dominated by the underlying DHT operations - for example, writing to a folder incurs one DHT insert in the common case. In this



**Figure 6.10** Histogram of the delivery time for emails with online recipient.

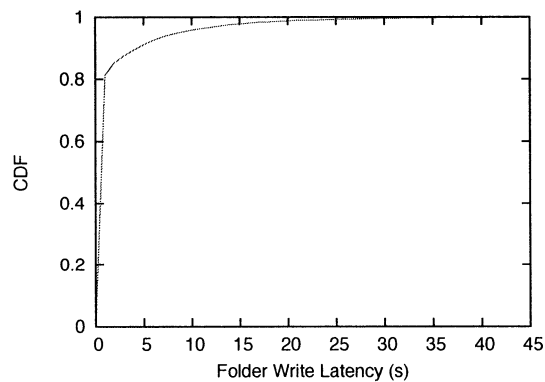
section, we present the performance of DHT read and write operations. In order to deal with Byzantine failures, POST declares an insert of data into the DHT successful once a majority of the replicas have responded successfully.

As is expected, we found that folder writes were more expensive than reads. Specifically, we found that folder reads took, on average, 1.4 seconds to complete. However, this is heavily dominated by a few large object reads - disregarding the longest 5% of these reads lowers the average to just 554 ms. Additionally, caching in PAST has significant benefits here: we found that 75% of the client fetch requests were completed in under 200 ms and were therefore likely cache hits.

We found that folder writes took an average of 2.0 seconds to complete. However, we saw the same behavior here as we did with folder reads - a few large writes dominated the average. In the case of the writes, removing the longest 5% of the writes results in an average write time of 1.2 seconds. Comparing these results to conventional servers is nontrivial, as they are dependent on a large number of factors

such as machine speed, load, and mailbox size. However, the consensus is that from the end user's standpoint, ePOST is faster at folder manipulation operations than our Computer Science department's mail server.

The variance of folder writes is slightly higher than expected, mainly due to two implementation details. In POST, log entries are collapsed into groups of 50 entries, which are pushed out together once the group of entries is filled. Additionally, in ePOST, folder snapshots are inserted every 500 entries, which significantly increases the cost of inserting the 500th entry, as snapshots of large folders are typically large. Both of these features cause every 50th and 500th log entry insertion to be much longer than average, which skews the latency distribution. A CDF of the folder write time is shown below in Figure 6.11. In general, we are still investigating mechanisms to reduce the variance in folder inserts. However, the variance in ePOST is predictable and does not increase as the system scales, unlike centralized systems where the variance increases as the load on the server grows.



**Figure 6.11** CDF of the folder write latency.

### 6.5.3 Availability

The last user-perceivable performance metric we discuss is the availability of ePOST. This metric, however, is the hardest to quantify as it is highly dependent on the characteristics of the deployment as well as the external events which occur during the deployment. For our experimental deployment, reporting availability figures is not particularly useful, as we rebooted the ring a few times to upgrade node software and we also took down the ring for a few days for a major software upgrade. In general, though, we found the overall availability to be high - only one unplanned ring failure occurred (and it was a direct result of a bug introduced in a software upgrade). Per node availability was relatively high, too, which is reflected in Figure 6.1.

## 6.6 Discussion

From our experimental deployment, we have shown that both the storage and bandwidth requirements of ePOST are practical. Without any garbage collection, the storage load on the entire network works out to be 7 MB per user per day, and we believe that garbage collection should reduce this figure significantly. The bandwidth required for ePOST works out to be approximately 500 bytes per node per second, a completely feasible number for our targeted environment of a large organizational LAN.

Throughout our testing and deployment, we encountered a number of challenges



and failures, and were forced to modify our Pastry and ePOST implementation to be robust against such failures. For example, we experienced a number of correlated failures including power failures, code bugs, and router misconfigurations. In fact, one computer misconfiguration caused the ePOST to become partitioned, a failure which ePOST handled without any data loss. We also experienced problems which were less correlated, such as Java Virtual Machine (JVM) bugs and OS-level machine hangs. ePOST (and Pastry where necessary) has benefited from this experience and are now more robust to such Byzantine failures.

We intend to further verify the scalability and practicability of ePOST by expanding our user base both within and outside of Rice. We initially plan to start other ePOST rings at the other institutions, as well as attract more internal Rice users.

## Chapter 7

### Related Work

In this Chapter, we discuss related work in both collaborative applications and peer-to-peer systems and applications.

#### 7.1 Collaborative Applications

Electronic mail (email) was the first major decentralized collaborative application. It was designed as an extension to the SENDMSG program to allow users to send messages to users on other machines connected to the ARPAnet. Email was initially sent using an extension to the File Transfer Protocol (FTP) [4]. However, the popularity of email necessitated a new protocol for email transmission, which was introduced in 1982 as the Simple Mail Transfer Protocol (SMTP) [37] and is still in use today. Additionally, other early architectures were built for the delivery of email messages, most notably the Grapevine system [5]. However, the management overhead of systems like Grapevine limits their scalability.

Current email protocols, including SMTP [37], POP3 [32], and IMAP [13], are tailored towards an infrastructure based on dedicated servers. These protocols are based on email, and do not provide the more generic support for collaborative applications that POST offers. Lotus Notes [47] and Microsoft Exchange [48] provide a general, secure messaging infrastructure based on the client-server model, providing

the ability to transfer email, personal contacts, calendars, and tasks. POST aims to provide similar functionality based on a serverless, decentralized and cooperative p2p architecture.

However, two major problems have become apparent with current email systems: scalability and security.

#### **7.1.1 Scalability**

There has been much work to allow email services to scale more effectively through the use of cluster-based servers. The most notable of such approaches include the Porcupine System [41] as well as Hotmail's [25] and Google's [22] mail services. Porcupine provides email services to a single organization by using a cluster of workstations to handle up to a billion messages per day, while webmail solutions use highly administered private clusters of machines. While these systems achieve massive scalability, they come at a high cost with respect to both equipment and maintenance overhead.

ePOST instead utilizes a completely decentralized, self-scaling architecture. ePOST therefore eliminates the need to purchase dedicated powerful mail servers or clusters of mail servers. Additionally, ePOST has the potential to reduce the management overhead, as it can take advantage of the self-organizing properties of the peer-to-peer overlay.

### 7.1.2 Security

Security has become a major problem for email as it has gained in popularity. The protocol for sending email, SMTP, has no verification built-in, which has lead to an explosion of bulk unsolicited commercial email (or *spam*) and, more recently, forged emails known as *phishing scams* [1]. Spam has quickly become the major problem for both email users and providers - AOL, for example, reports that up to 80% of its inbound email is spam [3].

Extensions to email, such as PGP [50] and GPG [46], provide secure, verifiable email but are not widely used. Other approaches to securing email from spoofing involve reverse DNS tricks [43, 42] but such proposals are only just now being finalized and implemented. ePOST has the advantage that encryption and verification are built-in from the beginning, removing most of these vulnerabilities.

## 7.2 Peer-to-Peer Applications

Peer-to-peer systems were first used as the basis for file-sharing and were based on unstructured overlays such as Gnutella [21]. A number of structured peer-to-peer overlays were created shortly after these unstructured systems were introduced [45, 38, 39, 49]. Applications designed for structured overlays range from basic distributed hash tables (DHTs) [15, 40, 29] and end-system multicast [9] to multimedia content distribution [8] and Usenet-style news services [44].

The use of a single-writer, self-authenticating log in POST was inspired by the use of similar logs in the Ivy decentralized filesystem [33]. The signed loghead is the root of a Merkle hash tree [31], which allows the log to be stored on untrusted nodes, while ensuring that the authenticity of each log entry can be verified locally. This allows POST to avoid more complex Byzantine state machine protocols [10].

Another proposed serverless email system [26] shares many of the goals of ePOST. Unlike POST, it focuses on email service only, and unlike ePOST, it is not compatible with the existing email infrastructure. Providing email services on top of a p2p storage system has also been explored in the OceanStore project [14]. The use of single-writer logs allows POST to achieve similar functionality with significantly less complexity, while allowing more general support for collaborative applications.

A few other structured peer-to-peer applications have been deployed on a wide scale. The most notable of these include Kademlia [30], which is used as the backbone of the eDonkey2000 [18] file sharing network. Also, the Coral [19] DHT has been used to provide a distributed web cache, Coral-CDN [11], and the OpenDHT [27] project aims to provide a general-purpose deployed DHT. However, the both of the latter systems are centrally administered on the Planetlab [36] network, and they do not address adding potentially untrusted entities.

## Chapter 8

### Conclusions

In this thesis, we have presented POST, a decentralized, serverless messaging system that leverages the resources of participating desktop computers. POST provides highly resilient and scalable messaging services, while ensuring confidentiality, data integrity, and authentication. Three general, lean services provided by POST can be used to support a variety of collaborative applications.

To demonstrate the feasibility of POST, we have also presented the design of ePOST, a system providing email services built using POST. We deployed ePOST within the Rice Computer Science department with real users relying on ePOST as their primary email account. The results show that that POST is efficient enough to support a demanding collaborative application, and that the overhead incurred with respect to storage and bandwidth are acceptable for our target environment.

Generally, we see that the success of POST as a bellwether for peer-to-peer applications. Since email has been shown to be practically provided in a completely decentralized setting, we believe that other demanding collaborative applications are possible. Using the POST architecture, for example, one can easily provide instant messaging, newsgroups, calendars, and shared whiteboards.

## References

1. Anti-Phishing scam working group. <http://www.antiphishing.org/> .
2. Anti-Spam reference site. <http://spam.abuse.net/> .
3. AOL spam press release, Nov. 2004. [http://media.aoltimewarner.com/media/newmedia/cb\\_press\\_view.cfm?release\\_num=55253692](http://media.aoltimewarner.com/media/newmedia/cb_press_view.cfm?release_num=55253692) .
4. A. K. Bhushan. RFC 385: Comments on the file transfer protocol, Aug. 1972.
5. A. Birrell, R. Levin, M. Schroeder, and R. M. Needham. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, Apr. 1982.
6. 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 the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*, Santa Clara, CA, 2000.
7. M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, Boston, MA, December 2002.
8. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, Oct. 2003.
9. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8), 2002.
10. M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, 1999.
11. Coral content distribution network. <http://www.scs.cs.nyu.edu/coral/> .
12. L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, Oct. 2003.

13. M. Crispin. RFC 3501: Internet message access protocol, version 4rev1, Dec. 1996.
14. S. Czerwinski, A. Joseph, and J. Kubiawicz. Designing a global email repository using OceanStore, June 2002. UC Berkeley summer retreat.
15. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Canada, 2001.
16. F. Dabek, B. Zhao, P. Druschel, J. Kubiawicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, California, 2003.
17. J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, 2002.
18. eDonkey file sharing network. <http://www.edonkey2000.com/> .
19. M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, California, 2004.
20. Freepastry. <http://www.cs.rice.edu/Systems/Pastry> .
21. Gnutella website. <http://www.gnutella.com> .
22. Google mail website. <http://www.gmail.com> .
23. A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Ensuring high data durability in peer-to-peer storage systems, Oct. 2004. Submitted for publication.
24. N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, Washington, Mar. 2003.
25. Hotmail website. <http://www.hotmail.com> .
26. J. Kangasharju, K. W. Ross, and D. A. Turner. Secure and resilient p2p e-mail: Design and implementation. In *Proceedings of the Third International Conference on P2P Computing*, Linkoping, Sweden, Sept. 2003.



27. B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, California, 2004.
28. J. Kinsin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. RFC 1869: SMTP service extensions, Nov. 1995.
29. J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gum-madi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proceedings of Ninth Inter-national Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, Cambridge, MA, 2000.
30. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings for the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, Massachusetts, Mar. 2002.
31. R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO'87 (LNCS, vol. 293)*, 1987.
32. J. Meyers and M. Rose. RFC 1939: Post office protocol, version 3, May 1996.
33. A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI'02)*, Boston, MA, December 2002.
34. T. Ngan, P. Druschel, and D. S. Wallach. Enforcing fair sharing of peer-to-peer resources. In *Proceedings for the Second International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, Feb. 2003.
35. Overnet website. <http://www.overnet.com> .
36. PlanetLab. <http://www.planet-lab.org/> .
37. J. Postel. RFC 821: Simple mail transfer protocol, Aug. 1982.
38. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the Third International Workshop on Networked Group Communication (NGC'01)*, Nov. 2001.
39. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM In-ternational Conference on Distributed Systems Platforms (Middleware'01)*, Hei-delberg, Germany, Nov. 2001.

40. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighteenth ACM Symposium on Operating System Principles (SOSP'01)*, Banff, Canada, 2001.
41. Y. Saito and B. B. H. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP'99)*, Charleston, South Carolina, Dec. 1999.
42. Sender-ID framework overview. <http://www.microsoft.com/senderid> .
43. Sender policy framework. <http://spf.pobox.com/> .
44. E. Sit, F. Dabek, and J. Robertson. UsenetDHT: A low overhead usenet server. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, CA, February 2004.
45. 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 the ACM Symposium on Communications Architectures and Protocols (SIGCOMM'01)*, San Diego, CA, 2001.
46. The GNU privacy guard. <http://www.gnupg.org/> .
47. S. Thomas, B. Hoyt, and B. J. Hoyt. *Lotus Notes & Domino 4.5 Architecture, Administration, and Security*. Computing McGraw-Hill, 1997.
48. J. Woodcock. *Introducing Microsoft Exchange 2000 Server*. Microsoft Press, 2000.
49. B. Zhao, J. Kubiawicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.
50. P. Zimmerman. PGP user's guide, Dec. 1992.